Лекции по алгоритмам и структурам данных.

Бабичев С. Л.

4 февраля 2021 г.

Оглавление

Предисловие

Данная книга написана по материалам лекций, прочитанных автором, по курсу «Алгоритмы и структуры данных» в Московском государственном университете в рамках Техносферы, совместного образовательного проекта `Mail.ru Group` и МГУ. Этот курс серьёзно отличается от традиционных учебных курсов, читаемых в высших учебных заведениях по двум аспектам:

1. Традиционные курсы алгоритмов и структур данных обычно весьма академичны. Большая часть из них основана на прекрасной книге Кормена с соавторами [4]. Ни эта книга, ни академические курсы не привязываются к какому-либо языку программирования. Написания программ, реализующих какой-либо алгоритм, не требуется, а ведь знания, не подтверждённые практикой их использования, имеют тенденцию к затуханию.
2. Другая группа курсов обычно ориентирована на подробное изучение деталей конкретного языка программирования — и алгоритмы в ней рассматриваются только как иллюстрация к свойствам языка.

Курс «Алгоритмы и структуры данных» Техносферы ориентирован на студентов и аспирантов, уже имеющих представление как о языках программирования `Си` и `C++`, так и уже писавших какие-то программы. Его цель — создать прочный программистско-алгоритмический базис, основываясь на котором, можно решать достаточно сложные алгоритмические задачи. В данной книге каждый из разобранных алгоритмов иллюстрируется примерами на языках `Си` и `C++`. При прохождении курса студенты должны решить несколько десятков задач, каждая из которых иллюстрирует либо отдельные алгоритмы, либо их композицию.

Основной принцип, проводимый в курсе — необходимость применения принципа разделения задачи на более простые подзадачи — классическую триаду «анализ, декомпозиция, синтез». Для успешного разбиения задачи на подзадачи необходимо представлять, какие подзадачи могут быть ре-

шены в ограниченное время, а какие — нет, какие подзадачи удобны для дальнейшей декомпозиции, а какие дальше разбивать не получится. Для этого у обучающегося создаётся некий набор типичных подзадач, фиксируются их основные характеристики и свойства — то, что в книге называется *абстракциями*. Опыт автора, как профессионального программиста с 30-летним стажем написания программ, относящихся к разделам системного программирования и математического моделирования, показывает, что при правильном подходе к декомпозиции даже сложные задачи можно свести к более простым, укоротив тем самым их разработку и реализацию[1].

Изучение «абстракций» создаёт у обучающихся некие «маячки», направления, по которым можно двигаться для достижения результата. На лекциях поясняется, почему именно эти «маячки» полезны для дальнейшего применения, и, решая задаваемые домашние и контрольные задачи, студенты учатся двигаться к цели, их используя.

Строгое доказательство корректности всех алгоритмов и их сложности, конечно, необходимо. К сожалению, эта строгость часто может быть достигнута только применением достаточно продвинутого математического аппарата. Ориентация на практическую сторону данного курса не позволяет приводить все необходимые доказательства, для этих целей есть такие замечательные книги, как [1, 2, 3, 4]. Однако полностью отказаться от понимания того, как доказывается корректность и сложность алгоритмов, ни в коем случае нельзя. Разработчик алгоритма, который не способен хотя бы в общих чертах представить характеристики разработанного алгоритма, заслуживает сожаления — и, в сущности, профессионально непригоден к этой деятельности. Поэтому мы обязательно будем для каждого рассмотренного алгоритма оценивать, возможно, нестрого, его сложность и корректность. Опора на уже решённые подзадачи здесь может очень пригодиться.

Материал разбит по лекциям, каждая из которых занимает четыре академических часа. После каждой из лекций приведено по пять домашних задач в том виде, как они были предоставлены студентам. Одна из целей курса — научить студентов определять нужные алгоритмы и использовать их. При этом требуется создать эффективную программу, которая должна вписаться в ограничения по времени и памяти.

Полное решение задачи засчитывалось только при прохождении всех тестов с указанными ограничениями и за указанное время. В качестве предпочтительного языка программирования использовались языки `Си` и `C++`, в

[1] Например, один из проектов, на разработку и реализацию которого планировалось три года, удалось сдать в промышленную эксплуатацию менее, чем за один год.

качестве дополнительных предлагались языки `Go` и `Rust`. Достаточно жёсткие временны́е ограничения делают невозможным прохождение тестов на языке `Python`[2]. От языка `C++` нужно лишь небольшое подмножество: возможность создавать свои классы, реализуя абстракции, и возможность использования уже готовых абстракций из стандартной библиотеки шаблонов `STL`. Некоторые задачи для решения требуют творческого осмысления алгоритмов, их небольшой модификации и адаптации к задаче.

Благодарности
Автор благодарит:

свою жену Наталью и своих детей Татьяну и Дмитрия, которые помогали находить интересные задачи, интересные темы и оказывали моральную и методическую поддержку;

Московский государственный университет и его прекрасных студентов, которые не стеснялись сообщать мне о замеченных неточностях в задачах и изложении материала;

Владимира Геннадьевича Абрамова, прочитавшего черновик рукописи и давшего ряд ценных замечаний;

коллектив `Mail.Ru group` и персонально Оксану Озёрную, без которых проект «Техносфера» не состоялся бы;

Дмитрия Гущина, тщательно прочитавшего первое издание этой книги и обратившего моё внимание на ряд описок, опечаток и сложных в понимании мест;

студентов Московского физико-технического института, которым я предлагал решать отдельные задачи из курса и благодаря которым был исправлен ряд неточностей в условиях.

[2] В связи с рядом особенностей языка `Python` он исполняет программы, алгоритмически эквивалентные программам на `C++`, во много раз медленнее.

Лекция 1

Тема наших занятий — алгоритмы и структуры данных. Эти понятия тесно связаны. Перефразируя классика, можно сказать: мы говорим «алгоритмы» — подразумеваем «структуры данных», мы говорим «структуры данных» — подразумеваем «алгоритмы». Невозможно создать хороший алгоритм, опираясь на неподходящие структуры данных.

Можно дать такое неформальное определение: алгоритм — это последовательность команд для *исполнителя*, обладающая свойствами:

- **полезности**, то есть умения решать поставленную задачу;
- **детерминированности**, то есть строгой определённости каждого шага во всех возможных ситуациях;
- **конечности**, то есть способности завершаться для любого множества входных данных;
- **массовости**, то есть применимости к разнообразным входным данным;
- **корректности**, то есть получения верных результатов для всех допустимых входных данных.

Список свойств алгоритмов можно продолжать и далее, но сейчас мы отметили те свойства, на которые будем обращать в дальнейшем особое внимание.

Каждый алгоритм для своего исполнения (ещё говорят — *вычисления*) требует от исполнителя некоторых *ресурсов*. *Программа* есть запись алгоритма на формальном языке.

Одну и ту же задачу зачастую можно решить несколькими способами, несколькими алгоритмами, которые могут отличаться использованием ресурсов, таких, как *элементарные действия* и *элементарные объекты*. Например, исполнитель алгоритма «компьютер» использует устройство *центральный процессор* для исполнения таких элементарных действий, как сложение, умножение, сравнение, переход и других, и устройство *оператив-*

ная память как хранителя элементарных объектов (целых и вещественных чисел). Способность алгоритма использовать ограниченное количество ресурсов называется *эффективностью*.

1.1 Сложность алгоритма

Если мы спросим у специалиста по алгоритмам, какая сложность у предложенного им алгоритма, он задаст встречный вопрос: а какую сложность вы имеете в виду?

Если требуется реализовать алгоритм в виде схемы вычислительного устройства, реализующего конкретную функцию, то *комбинационная сложность* определит минимальное число конструктивных элементов для реализации этого алгоритма. *Описательная сложность* есть длина описания алгоритма на некотором формальном языке. Один и тот же алгоритм на различных языках может иметь различную описательную сложность. Например, одна строка описания алгоритма на языке `Python` может быть эквивалентна нескольким десяткам строк алгоритма на языке `Pascal`. Нас, как составителей алгоритма, больше всего будет интересовать *вычислительная сложность*, определяющая количество элементарных операций, исполняемых алгоритмом для каких-то входных данных. Для алгоритмов, не содержащих циклов, описательная сложность примерно коррелирует с вычислительной. Если алгоритмы содержат циклы, то прямой корреляции нет, и нас интересует другая корреляция — времени вычисления от входных данных, причём обычно интересна именно асимптотика этой зависимости.

Давайте введём понятие *главный параметр* (мы его будем обычно обозначать буквой N), наиболее сильно влияющий на скорость исполнения алгоритма. Это может быть, например, размер массива при его обработке, количество символов в строке, количество бит в записи числа. Если нам приходится выделять несколько таких параметров, то постараемся создать функцию от них, определяющую один обобщённый параметр. Для определения вычислительной сложности алгоритма (а здесь и далее под термином *сложность* будет подразумеваться именно *вычислительная сложность*) введена специальная нотация. Мы опустим здесь строгие математические определения используемых символов O и Θ и дадим их неформально.

В дальнейшем изложении материала мы будем под термином *сложность* понимать именно вычислительную сложность, если об этом не будет сказано особо.

Определение 1. *Функция $f(N)$ имеет порядок сложности $\Theta(g(N))$, если*

существуют постоянные c_1, c_2 *и* N_1*, такие, что для всех* $N > N_1$

$$0 \leqslant c_1 g(N) \leqslant f(N) \leqslant c_2 g(N).$$

$\Theta(f(n))$ — *класс функций, примерно пропорциональных* $f(n)$.

На графике это выглядит таким образом: коэффициенты c_1 и c_2, умноженные на функцию $g(n)$, приводят к тому, что график функции $f(n)$ оказывается зажат между графиками функций $c_1 g(n)$ и $c_2 g(n)$.

Например, если мы о каком-то алгоритме сказали, что он имеет сложность $\Theta(N^2)$, где-то при больших N функция сложности будет неотличима от функции cN^2, где c — константа, которую ещё называют *коэффициент амортизации*.

Определение 2. *Функция* $f(N)$ *имеет порядок сложности* $O(g(N))$*, если существуют постоянные* c_1 *и* N_1*, такие, что для всех* $N > N_1$

$$f(N) \leqslant c_1 g(N)$$

$O(f(n))$ — *класс функций, ограниченных сверху* $cf(n)$.

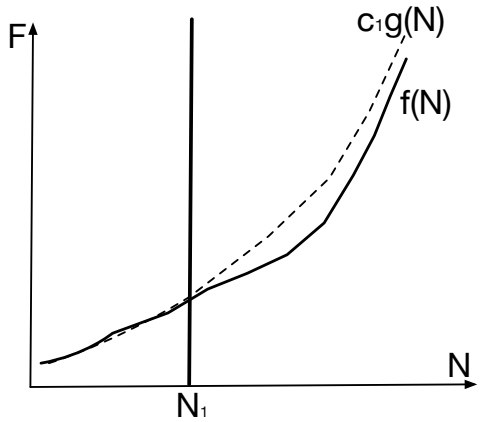


Рис. 1.1. Сложность O(N)

Пусть $f(N)$ — функция сложности алгоритма в зависимости от N.

Тогда если существует такая функция $g(N)$ (асимптотическая функция) и константа C, что

$$\lim_{N\to\infty} \frac{f(N)}{g(N)} = C,$$

то сложность алгоритма $f(N)$ определяется функцией $g(N)$ с коэффициентом амортизации C.

Говорят, что символ $\Theta(f(n))$ определяет класс функций, примерно пропорциональных $f(n)$, а символ $O(f(n))$ — класс функций, ограниченных сверху $cf(n)$.

Класс сложности алгоритма определяется по асимптотической зависимости *a(N)*.

- Экспонента с любым коэффициентом превосходит любую степень.
- Степень с любым коэффициентом, большим единицы, превосходит логарифм по любому основанию, большему единицы.
- Логарифм по любому основанию, большему единицы, превосходит 1.

Можно привести несколько примеров:

$F(N) = N^3 + 7N^2 - 14N = \Theta(N^3)$.

$F(N) = 1.01^N + N^{10} = \Theta(1.01^N)$.

$F(N) = N^{1.3} + 10\log_2 N = \Theta(N^{1.3})$.

Однако, не стоит делать поспешные выводы о том, что алгоритм A_1 с асимптотической сложностью $\Theta(N^2)$ заведомо хуже алгоритма A_2 с асимптотической сложностью $\Theta(N \log N)$. Вполне может оказаться так, что для небольших значениях N количество операций, требуемых для исполнения алгоритма A_1 может оказаться меньше (и существенно), чем для алгоритма A_2.

Время исполнения алгоритма, исчисляемое в элементарных операциях, может отличаться для различных входных данных. Рассмотрим элементарный пример.

1.1.1 Пример: поиск в массиве

Задача. Пусть имеется массив A длиной N элементов. Найти номер первого вхождения элемента со значением P.

Сколько операций потребуется, чтобы обнаружить искомый номер с помощью алгоритма, заключающегося в последовательном просмотре элементов массива?

Самый первый элемент массива может оказаться P, следовательно, минимальное количество операций поиска будет $K_{min} = 1$. Элемента в массиве может не быть совсем, и тогда для поиска потребуется ровно $K_{max} = N$ операций. А какое среднее значение количества поисков? Предполагая, что количество итераций алгоритма. требуемых для поиска, равномерно распределено по всем числам от 1 до N, получаем:

$$K_{avg} = \frac{\sum_{i=1}^{N} i}{N} = \frac{N \times (N+1)}{2N} = \frac{N+1}{2}$$

Можно ли сказать, что алгоритм имеет сложность порядка $\Theta(N)$ в общем случае?

Нет, в наилучшем случае $f(n) = 1$ совсем не зависит от N. Мы можем сказать, что в лучшем случае алгоритм имеет сложность $\Theta(1)$, в среднем и в худшем — сложность $\Theta(N)$. Но для данного алгоритма нам проще будет использовать O-нотацию: $f(N) = O(N)$.

Все ли задачи можно решить за полиномиальное время? Как выясняется, отнюдь не все. Вот очень простая в формулировке и тем не менее очень сложная задача.

1.1.2 Задача о наполнении рюкзака

Задача. Пусть имеется N предметов, каждый из которых имеет объём V_i и стоимость C_i, предметы неделимы. Имеется рюкзак вместимостью V. Требуется поместить в рюкзак набор предметов максимальной стоимости, суммарный объём которых не превышает объёма рюкзака.

Решение задачи. Как оказывается, задача не имеет решения с полиномиальной сложностью. Один из простых в реализации неполиномиальных по сложности алгоритмов заключается в следующем:

1. Перенумеруем все предметы.
2. Установим максимум достигнутой стоимости M в 0.
3. Составим двоичное число с N разрядами, в котором единица в разряде будет означать, что предмет выбран для укладки в рюкзак. Это число однозначно определяет расстановку предметов.
4. Рассмотрим все расстановки, начиная от $000\ldots000$ до $111\ldots111$. Для каждой из них подсчитаем значение суммарного объёма V_M.
   (a) Если суммарный объём расстановки V_M не превосходит объёма рюкзака V, то подсчитывается суммарная стоимость W_M и сравнивается с достигнутым ранее максимумом стоимости M.

(b) Если вычисленная суммарная стоимость превосходит максимум M, то максимум M устанавливается в вычисленную стоимость W_M и запоминается текущая конфигурация.

Алгоритм, как нетрудно убедиться, обладает всеми требуемыми свойствами: он *детерминирован*, так как его поведение зависит исключительно от входных данных; он *конечен*, так как его исполнение неизбежно прекратится, как только будут исчерпаны все расстановки; он *массовый*, так как он решает все задачи этого класса, и он *полезный*, так как даёт нам решение конкретной задачи.

Его сложность пропорциональна 2^N, так как требуется перебрать все возможные перестановки (мы не рассматриваем тривиальный вариант, когда все N предметов помещаются в рюкзак).

Много ли времени потребуется на решение задачи для $N = 128$?

Предположим, на подсчёт одного решения потребуется 10^{-9} секунд, то есть, одна наносекунда.

Предположим, задачу будет решать триллион компьютеров (10^{12}).

Тогда общее время решения задачи будет составлять

$$\frac{2^{128} \times 10^{-9}}{10^{12}} \text{секунд} \approx 10.8 \times 10^9 \text{лет}.$$

Это — пример задачи, которая имеет решение не полиномиальной сложности (NP), но до сих пор не найдено решение полиномиальной сложности (P). Мало того, не доказано, что она может иметь решение полиномиальной сложности. Однако проверить, удовлетворяет ли какое-либо предложенное решение условию корректности, можно за полиномиальное время. Имеется понятие *сертификат* решения, который явным образом определяет предложенное решение. Например, в рассмотренной нами задаче с рюкзаком, сертификатом может быть значение последовательности из N двоичных цифр. Точное решение подобных задач (а задача о рюкзаке относится к классу *NP-сложных* задач[3]) требует времени, превышающего все мыслимые значения. Мы должны понимать, что такие задачи существуют, и что для них лучше искать *приближённое* решение, которое может оказаться не таким сложным, пусть и не таким хорошим.

[3]Обсуждение NP-полноты выходит за рамки нашего курса, можно сказать лишь, что если будет найдено эффективное решение одной из NP-полных задач, то этим самым будет найдено эффективное решение и для остальных NP-полных задач.

1.2 Исполнитель

В нашем практическом курсе мы хотим решать практические задачи — и в качестве универсального исполнителя будем использовать язык C++. Этот язык, как и почти все остальные языки программирования, достаточно мощен для того, чтобы быть исполнителем любых алгоритмов. В дальнейшем изложении материала мы будем явным образом различать языки Си и C++ только в тех местах, где это является необходимым.

Под элементарными типами данных мы будем понимать отображаемые на вычислительную систему (аппаратный исполнитель) типы, такие как char, int, double.

Под элементарными операциями аппаратного исполнителя мы будем понимать операции над элементарными типами и операции передачи управления.

Во многих языках программирования, в том числе в C++, имеются элементарные типы данных, которые отображаются на элементарные типы аппаратного исполнителя. Другие же типы данных комбинируются из элементарных. Операции языка есть комбинация элементарных операций.

В других языках программирования, например, в языке Python, типы данных языка не отображаются явным образом на аппаратные типы данных, этим занимается ещё один слой — *интерпретатор* языка Python. Если в C++ операция сложения двух целых чисел часто переводится в одну-две операции аппаратного исполнителя, то в языке Python на это могут потребоваться сотни операций. Именно по этой причине мы не включили Python в языки, на которых можно сдавать домашние задания.

Давайте с этого момента под термином *язык программирования* иметь в виду язык программирования C++.

Рассмотрим к примеру цикл for — неэлементарную операцию языка.

```
int a[10];
int s = 0;
for (int i = 0; i < 10 && a[i] % 10 != 5; i++) {
  s += a[i];
}
```

Здесь имеется неэлементарный тип *массив*, представителем которого является a и элементарный тип int, представителем которого является s, элементарная операция присваивания (инициализации) s = 0, неэлементарная операция for, состоящая из операций присваивания i = 0, двух операций сравнения, и т. д.

Целые числа на современных компьютерах имеют двоичное представление, и этот факт можно использовать для понижения сложности операций. Пока производителям компьютеров не удалось исполнять операции целочисленного деления и умножения так же быстро, как побитовые операции и операции сложения/вычитания. Не блещут быстротой и явные операторы перехода. Операторы присвоения значения логической операции быстрее, чем операторы сравнения. Операции сдвига битов быстрее, чем эквивалентные явные операции умножения и деления на степень двойки.

Во всех современных вычислительных системах целые числа имеют двоичное внутреннее представление, что даёт нам возможности использовать это.

Условно разобьём все доступные нашему исполнителю элементарные операции на несколько групп:

— простые элементарные операции: сложение, вычитание, присваивание, побитовые;
— немного более сложные элементарные операции: сравнение;
— сложные элементарные операции: целочисленное умножение, деление (не на степень двойки);
— самые сложные элементарные операции: условный и безусловный переходы.

Например, машинный код, соответствующий

```
if (x > 0) t = 1;
else t = 0;
```

требует для аппаратуры исполнения операций перехода – и поэтому требует в несколько раз больше времени, чем эквивалентный код

```
t = x > 0;
```

который операций перехода не содержит. К счастью для программистов, современные оптимизирующие компиляторы тоже знают о сложности элементарных операций и стараются понижать их сложность в генерируемом машинном коде без изменения корректности алгоритма.

Побитовые операции можно иногда представить как параллельные операции над массивом битов. В ряде алгоритмов это оказывается полезным.

1.2.1 Аппаратные исполнители

Имеется много архитектур аппаратных средств. Мы перечислим те, с которыми вам, скорее всего, придётся столкнуться на практике.

- `X86` — изобретена `Intel`, лицензирована и производится в том числе и `AMD`. И тип данных `int`, и указатели занимают 32 бита. 32 бита занимает и максимально быстро обрабатываемый аппаратно целочисленный формат.
- `X64` — изобретена `AMD`, лицензирована и производится в том числе и `Intel`. `int` — 32 бита, но максимально обрабатываемый аппаратно и быстро целочисленный формат — 64 бита.
- `ARM` схожа с `X86`, `ARM64` — с `X64`. Почти все телефоны, большинство планшетных компьютеров, редко сервера, используют эту архитектуру. Почти не используется для ноутбуков и настольных компьютеров, но её доля растёт и в этом сегменте.

1.2.2 Модулярная арифметика

Задача. Найти последнюю цифру значения 3^{7^8}.

Решение задачи. Заметим, что последние цифры степени тройки образуют период.

$$\begin{aligned} 3^0 &\pmod{10} = 1 \\ 3^1 &\pmod{10} = 3 \\ 3^2 &\pmod{10} = 9 \\ 3^3 &\pmod{10} = 7 \\ 3^4 &\pmod{10} = 1 \\ 3^5 &\pmod{10} = 3 \\ &\dots \end{aligned}$$

Таким образом, последняя цифра результата определяется остатком от деления 7^8 на 4.

Аналогично:

$$\begin{aligned} 7^0 &\pmod{4} = 1 \\ 7^1 &\pmod{4} = 3 \\ 7^2 &\pmod{4} = 1 \\ 7^3 &\pmod{4} = 3 \\ &\dots \end{aligned}$$

$7^8 \pmod{4} = 1 \rightarrow 3^{7^8} \pmod{10} = 3$

Все аппаратные исполнители производят действия, используя модулярную арифметику. Вся компьютерная арифметика основана на тождествах:

$$
\begin{array}{llll}
(a+b) & (\mathrm{mod}\ m) = (a & (\mathrm{mod}\ m) + b & (\mathrm{mod}\ m)) \quad (\mathrm{mod}\ m) \\
(a-b) & (\mathrm{mod}\ m) = (a & (\mathrm{mod}\ m) - b & (\mathrm{mod}\ m)) \quad (\mathrm{mod}\ m) \\
(a \times b) & (\mathrm{mod}\ m) = (a & (\mathrm{mod}\ m) \times b & (\mathrm{mod}\ m)) \quad (\mathrm{mod}\ m) \\
\ldots & & &
\end{array}
$$

В качестве m при двоичном представлении выступают числа 2^8, 2^{16}, 2^{32}, 2^{64}. Например, при умножении двух 32-битных целых чисел `X` и `Y` без знака результат `Z=X*Y` есть остаток по модулю 2^{32} от результата математически точной операции умножения.

1.3 Инварианты. Индуктивное программирование

Часто при создании алгоритмов мы не задумываемся о математике, которая лежит в их основе, например, пользуемся индуктивными функциями, не подозревая этого.

1.3.1 Индуктивные функции

Пусть имеется множество M. Пусть аргументами функции f будут последовательности элементов множества M, а значениями — элементы множества N. Тогда, если значение функции f на последовательности $x_1, x_2, \ldots x_n$ можно восстановить по её значению на последовательности $x_1, x_2, \ldots, x_{n-1}$ и по элементу x_n, то такая функция называется *индуктивной.*

Если мы хотим найти наибольшее значение из всех элементов последовательности, то функция $maximum$ — индуктивна, так как

$$maximum(x_1, x_2, \ldots, x_n) = max(maximum(x_1, x_2, \ldots, x_{n-1}), x_n).$$

Определение 3. ***Предикат*** *— логическое утверждение, содержащее переменную величину.*

Определение 4. ***Инвариант*** *— предикат, сохраняющий своё значение после исполнения заданных шагов алгоритма.*

В алгоритме нахождения наибольшего элемента массива `a[N]` мы неявно пользуемся такими предикатами:

```
int m = a[0];
for (int i = 1; i < N; i++) {
  if (a[i] > m) {
    m = a[i];
  }
}
```

Предикат гласит, что для любого `i < N` переменная `m` содержит наибольшее значение из элементов `a[0]...a[i]`, то есть `m` всегда равна значению уже рассмотренной индуктивной функции `maximum`.

Рассмотрим один из простейших алгоритмов — найти сумму элементов массива.

```
// Вход: массив a[n]
// Выход: сумма его элементов
int sum = 0;
for (int i = 0; i < n; i++) {
  sum += a[i];
}
```

Применяемая нами операция — использование индуктивной функции.

Утверждение, что значение переменной `sum` для подмножества, состоящего из первых `i` элементов, есть сумма этих элементов, является инвариантом, то есть предикатом, значение которого всегда истинно.

1.3.2 Доказательство корректности алгоритмов

Инвариант — важнейшее понятие при доказательстве корректности алгоритмов или его фрагмента.

Путь доказательства корректности фрагмента алгоритма:

1. выбираем предикат (или группу предикатов), значение которого истинно до начала исполнения фрагмента;
2. исполняем фрагмент, наблюдая за поведением предиката;
3. если после исполнения предикат остался истинным при любых путях прохождения фрагмента, алгоритм корректен относительно значения этого предиката.

1.4 Автоматы

Определение 5. ***Автомат*** — *множество пар, состоящих из состояний P и переходов T.*

Имеются ***начальное состояние автомата*** *и* ***заключительное состояние****.*

Определение 6. ***Конечный автомат*** — *автомат с ограниченными множествами состояний и переходов.*

Определение 7. ***Вход автомата*** — *события, вызывающие переходы.*

Определение 8. ***Детерминированный конечный автомат*** — *конечный автомат, в котором одна и та же последовательность входных данных приводит при одном и том же начальном состоянии к одному и тому же заключительному.*

Задача. На вход алгоритма подаётся последовательность символов. Назовём *строкой* любую подпоследовательность символов, начинающуюся знаком одиночной или двойной кавычки и заканчивающуюся им же. Внутри строк могут находиться любые символы, кроме завершающего. Нужно определить корректность входной последовательности.

```
'abracadabra'-корректно.
'abra"shvabra cadabra' - корректно.
"" - корректно.
"abra'shravra' - некорректно.
```

Решение задачи. Задача смотрится весьма простой, однако прямолинейное решение наталкивается на неприятности: требуется рассмотреть несколько частных случаев, аккуратно избегая выхода за конец строки. Доказательство корректности данного алгоритма может оказаться не таким уж и простым.

```
bool check(string s) {
  int ps = 0;
  if (s.size() == 0) return true;
  while (ps < s.size()) {
    if (s[ps] == '\'') {
      while (++ps < s.size() && s[ps] != '\'')
        ;
      if (s[ps] == '\'') ps++;
```

```
      else if (ps >= s.size()) return false;
    } else if (s[ps] == '"') {
      while (++ps < s.size() && s[ps] != '"')
        ;
      if (s[ps] == '"') ps++;
      else if (ps >= s.size()) return false;
    } else {
      ps++;
    }
  }
  return true;
}
```

Для проектирования детерминированного конечного автомата, решающего данную задачу, требуется определить все состояния автомата и действия при возможных переходах.

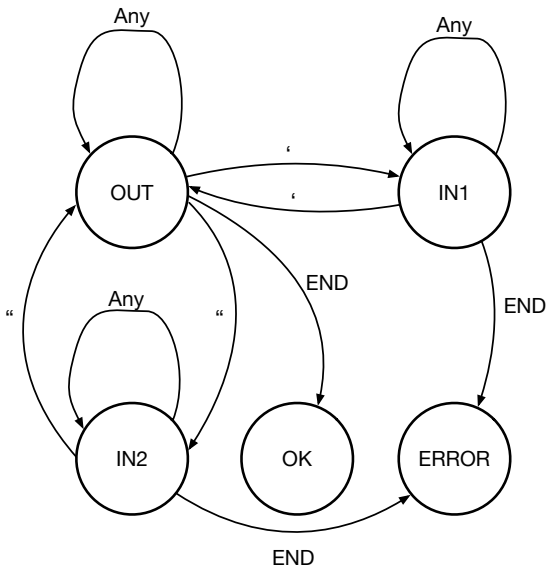


Рис. 1.2. Конечный автомат для задачи

Состояний, как оказывается, не столь уж и много. Это два заключительных состояния — `OK` и `ERROR`, одно начальное состояние `OUT`, к которому мы всё время возвращаемся, и два промежуточных — находится ли автомат в процессе разбора одиночной кавычки (`IN1`) или двойной кавычки (`IN2`).

Разбор теперь заключается в подаче очередного символа в автомат. Над каждой стрелкой перехода состояния написан символ, вызывающий этот переход.

Оказывается, что программа, реализующая этот автомат, не только более компактная, но и более понятная. Доказать её корректность достаточно просто.

```
bool DFA(string const &s) {
  enum {OUT, IN1, IN2} state = OUT;
  for (auto c: s) {
    if (state == IN1 && c == '\'') state = OUT;
    else if (state == IN2 && c == '"')  state = OUT;
    else if (state == OUT && c == '\'') state = IN1;
    else if (state == OUT && c == '"')  state = IN2;
  }
  return state == OUT;
}
```

1.5 Понятие интерфейс абстракции.

Как только появляются *объекты*, появляются *абстракции* — механизмы разделения сложных объектов на более простые, без деталировки подробностей разделения.

Функциональная абстракция — разделение функций, *методов*, которые манипулируют с объектами, с их реализацией.

Интерфейс абстракции — набор методов, характерных для данной абстракции.

При реализации неэлементарных операций алгоритма мы широко используем готовые конструкции языка.

1.5.1 Абстракция *Последовательность*

Одной из первых абстракций, с которыми мы встречаемся — абстракция *последовательность элементов*. Первые операции в любом языке, с

которых мы начинаем изучение, это операции ввода с клавиатуры и вывода на экран. В простых применениях этих операций у нас нет возможности вернуться назад и перечитать произвольное число, как и нет возможности вывести какое-то число, потом вернуться назад и вставить что-то перед ним.

Можно сказать, что эта абстракция реализует следующие операции:

- Создать объект *последовательность*[4]. Операционная система обычно предоставляет нам такие объекты при старте программы — стандартные ввод и вывод. Мы и сами можем создавать такие объекты (операция *открыть файл*), добавляя к ним атрибуты: объект предназначен для чтения или объект предназначен для записи.
- Удалить объект *последовательность*. Это может быть, например, операция `fclose`.
- Получить очередной элемент последовательности. Например, это может быть операция `getchar()`.
- Добавить элемент в последовательность, например, операцией `putchar(c)`.

В классической последовательности уже полученный однажды элемент второй раз получить из последовательности нельзя, можно только запомнить.

Интересно, что для обработки информации в файлах, содержащих терабайты, иногда вполне хватает единственной операции получения одного элемента. Алгоритмы, рассчитанные на обработку последовательностей, могут иметь сложность по памяти $O(1)$ и по времени $O(N)$.

1.5.2 Абстракция *массив*

Вторая замечательная абстракция — массив. Она имеется практически во всех языках программирования[5]. В интересующем нас языке `C++` эту абстракцию можно реализовать несколькими способами:

- Создать массив. У нас есть выбор — создавать массив статически, указывая его размер в квадратных скобках, или динамически, через операции заказа памяти `malloc`,`calloc` или `new` (в этом случае приходится явно или неявно пользоваться указателями).

[4]Философы могут возразить, что мы создаём не объекты, а экземпляры объектов — и они будут правы. Мы в дальнейшем изложении для краткости опускаем слово *экземпляр* в понятии *экземпляр объекта*.

[5]Почему *почти*? Поинтересуйтесь языком *LISP* в его классическом варианте. Есть и другие языки программирования, обходящиеся без массивов.

```
int a[100];
int *b = calloc(100, sizeof(int));
int *c = new int[100];
vector<int> d(100); // из библиотеки STL
```

- Удалить массив. При статически созданном массиве нужды в удалении массива нет. При динамически созданных массивах их нужно удалять явно, если массив создан при помощи указателей, или неявно, если он создаётся средствами `STL`.

```
free(b);
delete c;
```

- Обратиться к элементу массива.

```
int q1 = a[i];
int q2 = b[i];
int q3 = c[i];
int q4 = d[i];
```

Операции копирования обычных массивов, изменения их размеров, требуют вызовов соответствующих функций или комбинации более мелких операций, например, цикла и доступа к элементам. Например, если мы используем `STL`, то допустима операция `vector<int> e = d;`.

При реализации алгоритмов часто удобно абстрагироваться от того, каким именно образом представлен объект. Например, основная операция — доступ к элементу массива — выглядит одинаково для всех представлений.

Давайте введём ещё несколько полезных абстракций. При проектировании алгоритма мы можем их использовать, при более детальной реализации мы уже выберем, каким именно образом эти абстракции будут реализованы.

Для удобства мы будем представлять методы абстракций в *точечной нотации* после их объекта, например, `s.push`.

1.5.3 Интерфейс абстракции *стек*

Ещё одна удобная абстракция — *стек*. Стек должен предоставлять нам методы:

- **create** — создание стека. Может быть, потребуется аргумент, определяющий максимальный размер стека.
- **push** — занесение элемента в стек. Размер стека увеличивается на единицу. Занесённый элемент становится *вершиной стека*.

- **pop** — извлечь элемент, являющийся вершиной стека и уменьшить размер стека на единицу. Если стек пуст, то значение операции не определено.
- **peek** — получить значение элемента, находящегося на вершине стека, не изменяя стека. Если стек пуст, значение операции не определено.
- **empty** — предикат истинен, когда стек пуст.
- **destroy** — уничтожить стек.

Как стек представлять в программе? Можно парой массив/указатель стека. Можно классом C++. На семинарах рассматривается несколько способов реализации этой абстракции.

1.5.4 Интерфейс абстракции *множество*

Ещё очень важная абстракция — *множество* однотипных уникальных элементов, например, целых чисел. Между элементами должна быть определена операция сравнения на идентичность, чтобы мы могли эти элементы различать.

Мы будем обозначать множество списком значений внутри фигурных скобок. Например, пустое множество $s = \{\}$.

Эта абстракция реализует следующие методы:

- **insert** — добавление элемента в множество.
```
{1,2,3}.insert(5) -> {1,2,3,5}
{1,2,3}.insert{2} -> {1,2,3}
```

- **remove** — удалить элемент из множества.
```
{1,2,3}.remove(3) -> {1,2}
{1,2,3}.remove(5) -> или {1,2,3} или не определено.
```

- **in** — определить принадлежность множеству.
```
{1,2,3}.in(2) -> true
{1,2,3}.in(5) -> false
```

- **size** — определить количество элементов в множестве

Реализацию абстракции *множество* через побитовые операции мы подробно рассмотрим на семинарских занятиях.

1.6 Рекурсия. «Разделяй и властвуй»

С XII века известны числа Фибоначчи $\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots\}$ и способ их получения по правилу

$$F_n = \begin{cases} 0, & \text{если n} = 0, \\ 1, & \text{если n} = 1, \\ F_{n-1} + F_{n-2}, & \text{если } n > 1. \end{cases}$$

Это — рекуррентный способ записи последовательностей. Часто последовательности задаются только таким способом.

Такой алгоритм добывания элементов последовательности Фибоначчи очень легко запрограммировать:

```
int fibo(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibo(n-1) + fibo(n-2);
}
```

Корректен ли этот алгоритм? Да, мы просто реализовали его определение.

Каково время его работы? Этот вопрос сложнее.

Взглянем на *дерево вызовов* этой функции (рис 1.3):

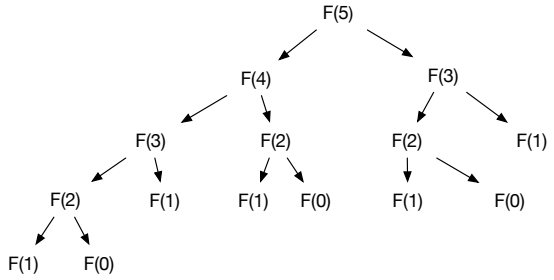


Рис. 1.3. Дерево вызовов для F(5)

Попытаемся оценить количество вызовов этой функции. Количество вызовов функции для $n = 0$ и $n = 1$ равно одному. Обозначив количе-

ство вызовов через $t(n)$, получаем $t(0) > F(0)$ и $t(1) = F(1)$. Для $n > 1$ $t(n) = t(n-1) + t(n-2) \geqslant F(n)$. Следовательно, при рекурсивной реализации алгоритма количество вызовов превосходит число Фибоначчи для соответствующего n.

Сами же числа Фибоначчи удовлетворяют отношению

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \Phi,$$

где $\Phi = \dfrac{\sqrt{5}+1}{2}$, то есть, $F_n \approx C \times \Phi^n$. Сложность этого алгоритма есть $\Theta(\Phi^N)$.

Как же так, алгоритм прост, но почему так медленно исполняется?

Проблема в том, что мы много раз повторно вычисляем значение функции от одних и тех же аргументов.

Требуемая для исполнения память характеризует *сложность алгоритма по памяти.*

- Каждый вызов функции создаёт новый *контекст функции* или *фрейм вызова.*
- Каждый фрейм вызова содержит все аргументы, локальные переменные и служебную информацию.
- Максимальное количество фреймов, которое создаётся, равно глубине рекурсии.
- Сложность алгоритма по занимаемой памяти равна $O(N)$.

Третий вопрос: можно ли ускорить алгоритм? Да, и достаточно просто. Если мы введём добавочный массив для сохранения предыдущих значений функции, дело пойдёт на лад и уже вычисленные значения функции не будут вычисляться повторно:

```
int fibo(int n) {
  const int MAXN = 1000;
  static int c[MAXN];
  if (n == 0) return 0;
  if (n == 1) return 1;
  if (c[n] > 0) return c[n];
  return c[n] = fibo(n-1) + fibo(n-2);
}
```

Алгоритм остаётся рекурсивным, но дерево рекурсии становится вырожденным (рис 1.4).

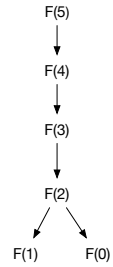


Рис. 1.4. Вырожденное дерево вызовов для F(5)

Обратите внимание на серьёзную проблему: пока мы реализовали только алгоритмическую часть решения нашей проблемы, то есть установили верный порядок исполнения операций. А вот с данными всё плохо. Значение функции растёт слишком быстро — и уже при небольших значениях n число выйдет за пределы разрядной сетки.

Мы здесь столкнулись с тем, что в реальных программах, исполняющихся на реальных компьютерах, существуют ограничения на операнды машинных команд. Их размер зависит и от архитектуры компьютера, и от исполняющейся операционной системы. Самые популярные пока 32-битные операционные системы стремительно уступают место 64-битным. Согласно установившемуся жаргону, 32-битная архитектура на `Intel`-совместимых компьютерах называется `X86` или `IA32`. 64-битных архитектур две, одна уже малопопулярна, разработка её прекращена — это архитектура `IA64`, процессоры `Itanium`. Вторая, как ни странно, была разработана конкурентом `Intel`, компанией `AMD`, и она так и называется — `AMD64` (правда, `Intel` называет её `EMT64`), другое её название `X64`. В языках `Си` и `C++` на архитектурах `X86` и `X64` стандартный тип данных `int` занимает ровно 32 бита. Чтобы воспользоваться 64-разрядной арифметикой, достаточно использовать тип данных `long long` (с префиксом `unsigned` или без него). Как говорят математики, не умаляя общности, мы будем в дальнейшем полагать, что основной единицей данных для вычисления является `int`, и будем оценивать сложность алгоритмов, исходя из этого.

Когда мы говорим про числа в алгоритмах, нас не очень заботит тот факт, что при исполнении алгоритма на каком-либо исполнителе некоторые

числа, получающиеся при вычислениях, оказываются не представимы на этом исполнителе. Мы должны ответить себе на вопрос: *что есть число в алгоритме*?

Мы видим, что значение функции `fibo(n)` растёт слишком быстро и уже при небольших значениях n число выйдет за пределы разрядной сетки, какую бы архитектуру мы ни использовали, то есть станет непредставимо на исполнителе алгоритма. Поэтому будем иметь в виду следующее:

- Алгоритм `fibo` оперирует с числами.
- Программа, реализующая алгоритм `fibo`, имеет дело с *представлениями* чисел.

> Любой исполнитель алгоритма имеет дело не с числами, а с их представлениями.

1.7 Представление чисел в алгоритмах

Оценим, насколько затратен такой переход от абстрактных чисел к их представлениям.

В реальных программах имеются ограничения на операнды машинных команд. `X86`,`X64` → `int` есть 32 бита, `long long` есть 64 бита.

На 32-битной архитектуре сложение двух 64-разрядных → сложение младших разрядов и прибавление бита переноса к сумме старших разрядов. Две или три машинных команды.

`X86`: сложение: 32-битных ≈ 1 такт; 64-битных ≈ 3 такта.

`X64`: сложение: 32-битных ≈ 1 такт; 64-битных ≈ 1 такт.

`X86`: умножение: 32-битных ≈ 3-4 такта; 64-битных ≈ 15-50 тактов.

`X64`: умножение: 32-битных ≈ 3-4 такта; 64-битных ≈ 4-5 тактов.

Команды деления целых чисел и нахождения целочисленного остатка на современных компьютерах весьма долго исполняются. Мы будем их использовать только в тех случаях, когда без этого в алгоритме не обойтись.

1.7.1 Представление длинных чисел

Для решения проблемы представления всевозможных целых чисел для исполнителя введём абстрактную структуру данных «длинное число», над которой определены все те же операции, что и над элементарно представимыми целыми числами — сложение, вычитание, умножение, деление, нахождение остатка от деления, сравнение.

Представлять такие числа можно многими способами, но, для удобства вычислений, мы воспользуемся привычной нам позиционной системой счисления, правда, с необычным основанием.

Так как длинные числа имеют представление в виде *цифр*, представляющих удобный для нас тип данных в позиционной системе счисления, то все операции будут производиться в этой системе счисления.

Мы привыкли использовать по одному знаку на десятичную цифру, то есть использовать десятичную систему. Аппаратному исполнителю удобнее работать с длинными числами в системе счисления по основаниям, большим 10 ($2^8, 2^{16}, 2^{32}, 2^{64}$).

Определение 9. *`(n)-числа` — те числа, которые требуют не более `n` элементов элементарных типов (цифр) в своём представлении.*

Например, если за элементарный тип данных для представления на 32-битной архитектуре мы выберем тип `int`, то `long long` будут `(2)-числами`.

`(n)-числа`, где `n>2` в исполнительной системе языков `Си` и `C++` уже представления не имеют[6]. Представление длинных чисел требует массивов элементарных типов. Основание системы счисления `R` для каждой из цифр представления должно быть представимо элементарным типом данных аппаратного исполнителя.

Сколько операций потребуется для сложения двух `(n)-чисел`?

Похоже, что оптимальнее школьного алгоритма сложения «в столбик» придумать что-либо трудно.

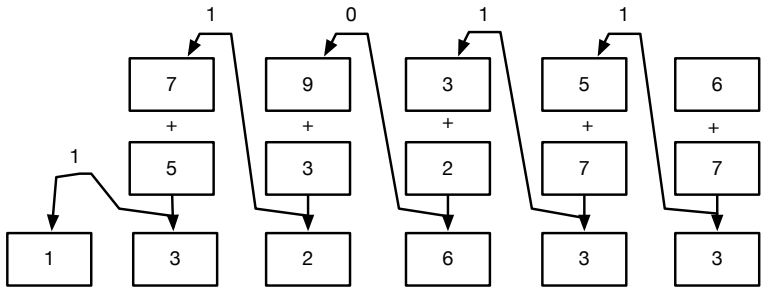


Рис. 1.5. Школьный алгоритм сложения

[6] в последних версиях `gcc/g++` появился тип данных `int128_t`, что представляет `(4)-числа`, но на этом всё заканчивается.

Каждую цифру первого числа нужно сложить с соответствующей цифрой второго и учесть перенос из соседнего разряда. Сложность алгоритма составляет $O(n)$.

Школьный алгоритм умножения длинных чисел тоже на первый взгляд кажется оптимальным: мы умножаем первое число на каждую из цифр второго, на что требуется n операций умножения, затем складываем все n промежуточных результатов, что даёт нам $O(n^2)$ операций умножения и $O(n^2)$ операций сложения.

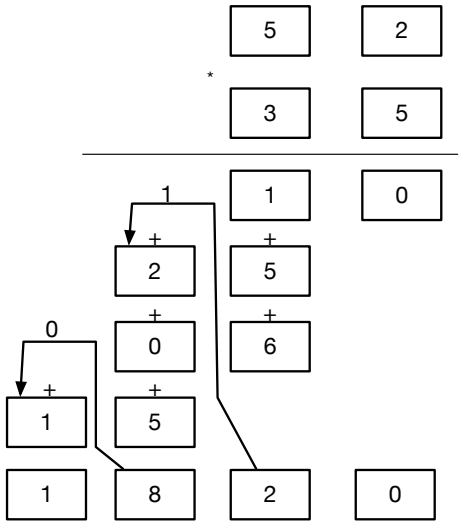


Рис. 1.6. Школьный алгоритм умножения

Можно ли умножать быстрее?

Мы будем полагать, что существует операция умножения двух 32-битных чисел, дающая 64-битное число, и это — одна операция. Именно поэтому мы не переходим к 64-разрядным числам как к элементарным единицам, потому что нам тогда потребуется операция умножения двух 64-разрядных чисел с получением 128-разрядного результата. Некоторые компиляторы имеют такую встроенную функцию, некоторые — нет.

Будем ориентироваться на более слабые компиляторы и оставим использование 64-битной арифметики как резерв для дальнейшей оптимизации.

Итак, наивный алгоритм умножения длинных чисел (n) имеет сложность $O(N^2)$. К счастью, имеется более быстрый алгоритм. Он изобретён в 1960-х годах аспирантом А. Н. Колмогорова Анатолием Карацубой и с тех пор является неизменным участником любых библиотек работы с большими числами. Нас он интересует постольку, поскольку реализует принцип Цезаря — *разделяй и властвуй*. Пусть нам требуется перемножить два (2n)-числа. Введём константу T, на единицу большую максимального числа, представляемого (n)-числом. Тогда любое (2n)-число X можно представить в виде суммы $Tx_u + x_l$. Это разложение имеет сложность $O(n)$, так как оно заключается просто в копировании соответствующих разрядов (n)-чисел.

$$N_1 = Tx_1 + y_1$$
$$N_2 = Tx_2 + y_2$$

При умножении в столбик

$$N_1 \times N_2 = T^2 x_1 x_2 + T((x_1 y_2 + x_2 y_1) + y_1 y_2.$$

Это — четыре операции умножения и три операции сложения. Число T определяет, сколько нулей нужно добавить к концу числа в соответствующей системе счисления, и мы полагаем сложность этой операции равной $O(1)$.

Алгоритм Карацубы находит произведение по другой формуле:
$N_1 \times N_2 = T^2 x_1 x_2 + T((x_1 + y_1)(x_2 + y_2) - x_1 x_2 - y_1 y_2) + y_1 y_2$.

Рассмотрим алгоритм на примере произведения чисел 56 и 78, приняв T за 10 (мы хотим получить ответ в десятичной системе счисления).

$x_1 = 5, y_1 = 6$
$x_2 = 7, y_2 = 8$
$x_1 x_2 = 5 \times 7 = 35$
$(x_1 + y_1)(x_2 + y_2) = (5 + 6)(7 + 8) = 11 * 15 = 165$
$y_1 y_2 = 6 \times 8 = 48$
$N_1 \times N_2 = 35 * 100 + (165 - 35 - 48) * 10 + 48 = 3500 + 920 + 48 = 4368$

Мы уменьшили число операций умножения за счёт увеличения операций сложения. Принесёт ли это нам выгоду? На этот вопрос нам поможет ответить *основная теорема о рекурсии*.

1.8 Основная теорема о рекурсии

А всё-таки, как определить, какой порядок сложности будет иметь рекурсивная функция, не проводя вычислительных экспериментов?

Так как рекурсия есть разбиение задачи на подзадачи с последующей *консолидацией* результата, обозначим количество подзадач, на которые разбивается задача за a. Пусть размер каждой подзадачи уменьшается в b раз и становится $\left\lceil \frac{n}{b} \right\rceil$.

Пусть сложность консолидации после решения подзадач есть $O(n^d)$. Тогда сложность такого алгоритма, выраженная рекуррентно, есть

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d).$$

Упрощённый вариант основной теоремы о рекурсии (в [4] она сформулирована немного сложнее и строже), которую мы даём здесь без доказательств, звучит так:

Теорема 1. *Пусть* $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ *для некоторых* $a > 0, b > 1, d \geqslant 0$. *Тогда*

$$T(n) = \begin{cases} O(n^d), & \text{если } d > \log_b a, \\ O(n^d \log n), & \text{если } d = \log_b a, \\ O(n^{\log_b a}), & \text{если } d < \log_b a. \end{cases}$$

Рассуждая неформально, можно заметить, что общая сложность алгоритма есть сумма членов геометрической прогрессии с знаменателем $q = \frac{a}{b^d}$. При $q < 1$ она сходится и её сумма оценивается через первый член, при $q > 1$ она расходится и её сумма оценивается через её последний член, а при $q = 1$ все члены (а их $O(\log n)$) равны.

Оценим сложность алгоритма Карацубы по этой теореме.

Коэффициент a порождения задач здесь равен трём, так как одна первичная операция «большого» умножения требует трёх операций «маленького» умножения. Коэффициент уменьшения размера подзадачи $b = 2$, мы делим число на две примерно равные части. Консолидация решения производится за время $O(n)$, так как она заключается в операциях сложения и вычитания (которые имеют сложность $O(n)$ и их строго определённое количество), следовательно, $d = 1$. Так как $1 < \log_2 3$, то в действие вступает третий случай теоремы и, следовательно, сложность алгоритма есть $O(N^{\frac{3}{2}})$.

Операция умножения чисел (n) при умножении в столбик имеет порядок сложности $O(n^2)$.

Наличие большого количества операций сложения и вычитания говорит о том, что для малых (n) алгоритм может исполняться дольше, чем «школьный», и поэтому в реальных программах рекурсию стоит ограничить. Например, в большинстве библиотек длинной арифметики алгоритм Карацубы используется при достаточно больших значениях n.

1.9 Ещё немного о сложности

При вычислении больших чисел Фибоначчи можно воспользоваться аппаратом линейной алгебры. Введём вектор-столбец $\begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$, состоящий из двух элементов последовательности Фибоначчи, и умножим на него матрицу $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ так, чтобы получился новый вектор:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}.$$

Для вектора-столбца из элементов F_{n-1} и F_n умножение на ту же матрицу даст:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} + F_n \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}.$$

Таким образом,

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}.$$

Это означает, что для нахождения n-го числа Фибоначчи достаточно возвести матрицу $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ в n-ю степень.

1.10 Быстрое возведение в степень

Можно ли возвести число в n-ю степень за число операций, меньших $n - 1$?

Возведение числа в квадрат есть умножение числа на себя, и достаточно быстро приходит в голову, что, например, возведение в 16-ю степень можно

произвести не за 15 операций умножения, а всего за 4:

$$x^{16} = (x^8)^2 = ((x^4)^2)^2 = (((x^2)^2)^2)^2.$$

С показателями степеней, не равными степеням двойки, вроде бы всё не так просто. Эксперименты над числом 18 покажут нам, что и здесь всё хорошо:

$$x^{18} = (x^9)^2 = (x^8 \cdot x)^2 = (((x^2)^2)^2 \cdot x)^2.$$

Таким образом, можно вывести следующую рекуррентную формулу возведения в степень:

$$x^n = \begin{cases} 1, & \text{если } n = 0 \\ (x^{\frac{n}{2}})^2 & \text{если } n \neq 0 \wedge n \pmod 2 = 0 \\ (x^{n-1}) \times x & \text{если } n \neq 0 \wedge n \pmod 2 \neq 0 \end{cases}$$

Имея рекуррентную формулу, легко получить первичный код на любом языке программирования, допускающем рекурсивные функции, например, на `C++`:

```
SomeType pow(SomeType x, int n) {
  if (n == 0) return (SomeType)1;
  if (n & 1) return pow(x, n-1);
  SomeType y = pow(x, n/2);
  return y*y;
}
```

Мы полагаем, что в приведённом коде имеется тип данных `SomeType`, для которого определены все необходимые операции.

Как оценить сложность этого алгоритма? Представим степень, в которую мы возводим, в виде двоичного числа, например, степень 25 в виде 11001. Тогда нечётная степень будет означать, что последний разряд в двоичном представлении степени есть единица и операция $n-1$ есть её обнуление. Чётная же степень будет означать, что последний разряд равен нулю и деление такого числа на два есть вычёркивание этого разряда. Каждую из единиц требуется уничтожить, не изменяя количества разрядов, и каждый из разрядов требуется уничтожить, не изменяя количества единиц. Таким образом, сложность алгоритма равна $O(\log N)$.

1.11 Домашние задания

Задача 1. Симметрическая разность

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

На вход подаётся множество чисел в диапазоне от 1 до 20000, разделённых пробелом. Они образуют множество A. Затем идёт разделитель — число 0, и на вход подаётся множество чисел B, разделённых пробелом. 0 — признак конца описания множества (во множество не входит). Необходимо вывести множество $A \Delta B$ — симметрическую разность множеств A и B в порядке возрастания элементов. В качестве разделителя используйте пробел. В случае, если множество пусто, вывести 0.

Формат входных данных

1 2 3 4 5 0 1 7 5 8 0

Формат выходных данных

2 3 4 7 8

Примеры

стандартный ввод	стандартный вывод
1 2 6 8 7 3 0 4 1 6 2 3 9 0	4 7 8 9

Замечание

Для вывода можно использовать любой алгоритм сортировки.

Задача 2. Два массива

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Даны два упорядоченных по неубыванию массива. Требуется найти количество таких элементов, которые присутствуют в обоих массивах. Например, в массивах (0, 0, 1, 1, 2, 3) и (0, 1, 1, 2) имеется четыре общих элемента — (0, 1, 1, 2).

Первая строка содержит размеры массивов $N1$ и $N2$. В следующих $N1$ строках содержатся элементы первого массива, в следующих за ними $N2$ строках — элементы второго массива.

Программа должна вывести ровно одно число — количество общих элементов.

Формат входных данных

N_a N_b
a_1
a_2
...
a_{N_a}
b_1
b_2
...
b_{N_n}

Формат выходных данных

Одно целое число — количество общих элементов.

Примеры

стандартный ввод	стандартный вывод
5 5 1 1 2 2 3 0 1 3 3 4	2

Задача 3. Вычисление полинома

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	16 мегабайт

Вычисление полинома — необходимая операция для многих алгоритмов. Нужно вычислить значение полинома

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x^1 + a_0.$$

Так как число n может быть достаточно велико, требуется вычислить значение полинома по модулю MOD. Сделать это предлагается для нескольких значений аргумента.

Формат входных данных

Первая строка файла содержит три числа — степень полинома $2 \leqslant N \leqslant 100000$, количество вычисляемых значений аргумента $1 \leqslant M \leqslant 10000$ и модуль $10 \leqslant MOD \leqslant 10^9$.

Следующие $N + 1$ строк содержат значения коэффициентов полинома $0 \leqslant a_i \leqslant 10^9$ в порядке от старшего к младшему.

В очередных M строках содержатся значения аргументов $0 \leqslant x_i \leqslant 10^9$.

Формат выходных данных

Выходной файл должен состоять из ровно M строк — значений данного полинома при заданных значениях аргументов по модулю MOD.

Примеры

стандартный ввод	стандартный вывод
2 5 10 1 5 4 0 1 2 3 4	4 0 8 8 0
5 9 10 1 0 0 0 0 1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Задача 4. Считаем комментарии

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	256 мегабайт

Комментарием в языке `Object Pascal` является любой текст, находящийся между последовательностью символов, начинающей комментарий определённого вида, и последовательностью символов, заканчивающей комментарий этого вида.

Виды комментариев могут быть следующие:

1. Начинающиеся с набора символов `(*` и заканчивающиеся набором символов `*)`.
2. Начинающиеся с символа `{` и заканчивающиеся символом `}`
3. Начинающиеся с набора символов `//` и заканчивающиеся символом новой строки.

Еще в языке `Object Pascal` имеются литеральные строки, начинающиеся символом одиночной кавычки `'` и заканчивающиеся этим же символом. В корректной программе строки не могут содержать символа перехода на новую строку.

Будьте внимательны, в задаче и тестах к ней используются только символы с кодами до 128, то есть, кодировка `ASCII`. При тестировании своего решения будьте внимательны. Код одиночной кавычки — 39, двойной — 34.

Формат входных данных

На вход программы подаётся набор строк, содержащих фрагмент корректной программы на языке `Object Pascal`.

Формат выходных данных

Выходом программы должны быть четыре числа — количество комментариев первого, второго и третьего типов а также количество литеральных строк.

Пример

стандартный ввод	стандартный вывод

program test;
(*just for testing *)
var
(* variables
note that
// here is not comment
and (* here is
not a begin of
another comment
*)
x: integer; (* *)
begin
write('(*is not comment//');
write(' and (*here*) '
,x // y);
end. // It is comment
	3 0 2 2

Задача 5. Две кучи

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Имеется $2 \leqslant N \leqslant 23$ камня с целочисленными весами $W_1, W_2, \ldots, W_N$. Требуется разложить их на две кучи таким образом, чтобы разница в весе куч была минимальной. Каждый камень должен принадлежать ровно одной куче.

Формат входных данных

```
N
W1 W2 W3 ... WN
```

Формат выходных данных

Минимальная неотрицательная разница в весе куч.

Примеры

стандартный ввод	стандартный вывод
5 8 9 6 9 8	4
6 14 2 12 9 9 8	2

Лекция 2

2.1 Экстремальные задачи

Очень большое число задач, решаемых с помощью компьютеров, связано с оптимальным нахождением чего-либо. Это может быть наилучший путь путешественника, желающего посетить определённое число городов и потратить наименьшую сумму денег. Это может быть определение оптимального режима работы светофора, чтобы пропускать наибольшее количество транспортных средств за одно и то же время. Задача о рюкзаке относится к той же области.

Задачи нахождения оптимального значения, максимального или минимального, носят название *экстремальных*. Общий процесс решения таких задач называется *оптимизацией*. Методы оптимизации — отдельная, до сих пор развивающаяся область математики, и мы не претендуем на то, что мы сможем решать все поставленные задачи. Решать *некоторые* задачи мы, пожалуй, сможем, а для других *некоторых* задач мы сможем найти лишь не лучшее, но приемлемое решение.

2.2 Жадные алгоритмы

Жадные алгоритмы состоят из итераций и принимают решение на каждом шаге, стараясь найти *локально оптимальное* решение.

Как пример типичного жадного алгоритма, можно привести минимизацию значения функции методом, похожим на метод *покомпонентного спуска*. Предположим, что имеется непрерывная функция n переменных $f(x_1, x_2, \ldots, x_n)$, принимающая действительные значения на области определения. Она определяет поверхность в n-мерном пространстве. Один из

простейших алгоритмов минимизации такой функции заключается в следующем:

1. выбираем начальную точку $(x_1, x_2, \ldots, x_n)$, она становится текущей точкой алгоритма;
2. обследуя точки вокруг текущей, находим такую, в которой функция $f(x'_1, x'_2, \ldots, x'_n)$ имеет минимальное значение;
3. если найденная точка отлична от текущей, то делаем её текущей и переходим к второму шагу алгоритма;
4. конец.

Результаты этого алгоритма хорошо видны при попытке оптимизировать с его помощью одномерную функцию. Если функция *юнимодальна*, то есть имеет единственный оптимум, то алгоритм приводит к решению. Но для функций, имеющих несколько оптимумов, он может привести к неверным результатам.

Попытаемся найти минимум функции от двух переменных $f_1(x, y) = (x-3)^2 + (y+2)^2$.

За начальную точку примем $(x_0 = 0, y_0 = 0)$. Шаг поиска 0.1.

Простейший вариант нахождения минимума «осматривает» окрестности текущей точки в направлении осей координат. Есть, конечно, и другие варианты трактовки слова «окрестность текущей точки», но нам сейчас достаточно простейшей трактовки.

Вычисляем значения функции в текущей точке и в окрестных:

$$
\begin{aligned}
&f(0,0) = 3^2 + 2^2 = 13\\
&f(0+0.1,0) = 2.9^2 + 2^2 = 8.41 + 4 = 12.41\\
&f(0-0.1,0) = 3.1^2 + 2^2 = 9.61 + 4 = 13.61\\
&f(0,0+0.1) = 9 + 4.41 = 13.41\\
&f(0,0-0.1) = 9 + 3.61 = 12.61
\end{aligned}
$$

Так как значение функции в точке $(0.1, 0)$ меньше, чем в остальных кандидатах и меньше, чем в текущей точке, точка $(0.1, 0)$ становится текущей.

Код на языке `Си`, реализующий этот простой алгоритм, тоже прост:

```
#include <stdio.h>
double f(double x, double y) {
  return (x-3)*(x-3) + (y+2)*(y+2);
}

int main() {
  double x0 = 0., y0 = 0., d = 0.1;
```

```
  double dx[] = {d, -d, 0, 0}, dy[] = {0, 0,  d, -d};
  double newx, newy;
  bool bestfound = false;
  double maxf = f(x0, y0);
  while (!bestfound) {
    bestfound = true;
    for (int i = 0; i < 4; i++) {
      double newf = f(x0+dx[i],y0+dy[i]);
      if (newf < maxf) {
          maxf = newf;
          bestfound = false;
          newx = x0+dx[i];
          newy = y0+dy[i];
      }
    }
    if (!bestfound) {
      x0 = newx;
      y0 = newy;
    }
  }
  printf("Best f(%.1f,%.1f)=%.2f\n", x0, y0, maxf);
}
```

Обратите внимание на технический приём, который здесь используется: для нахождения точек-кандидатов мы использовали массивы `dx` и `dy`, содержащие приращения координат для всех четырёх возможных направлений попыток. Это позволило вычислять координаты кандидатов внутри цикла.

Следующий шаг. Текущая точка пока `(0.1, 0)`, и мы выбираем из точек `(0.2, 0)`, `(0,0)`, `(0.1, 0.1)`, `(0.1, -0.1)`. Сравнение значений функций в этих точках приводит нас к очередной точке `(0.2,0)`.

Далее маршрут проходит через точки от `(0.3,0)` до `(1,0)`, затем — до `(3,2)`.

Решение правильное.

Везение ли это? При любой ли начальной точке мы обязательно попали бы в нужную? Как выясняется, наша функция была слишком хороша — и данный алгоритм неизбежно привёл бы нас к правильному решению.

А что произойдёт с решением задачи для функции $f_2(x, y) = (x - 3)^2 + 10 \sin x + (y + 2)^2$?

Прогон нашего алгоритма с начальной точкой (0,0) выдаст, что лучшим решением будет точка (-0.7,2) с значением функции в ней ≈ 7.24, хотя минимум функции f_2 достигается в точке $(\approx 4.4, -2)$ с значением ≈ -7.6.

Проблема в том, что функция f_1 имела один глобальный минимум и не имела локальных, а функция f_2 наряду с глобальным минимумом имеет несколько локальных.

Если бы мы взяли начальную точку вблизи точки глобального минимума, то алгоритм бы сошёлся с предусмотренной нами точностью.

Говорят, что данный алгоритм *склонен к нахождению локальных экстремумов*.

Жадные алгоритмы, если они существуют, обычно работают быстро. Наша ближайшая задача — понять, в каких случаях жадные алгоритмы корректны и дают верное решение, а в каких случаях нам на них надеяться не стоит.

2.3 Задача об интервалах

Для начала попробуем решить следующую задачу:

Задача (задача об интервалах). На прямой дано множество отрезков. Необходимо найти максимальный размер подмножества непересекающихся отрезков из этого множества.

Эту задачу можно переформулировать в терминах времени: имеется аудитория, на которую имеется несколько претендентов. Каждый из претендентов заявляет время, в которое он может эту аудиторию занять, и время, в которое он эту аудиторию может освободить. Администрация здания, в котором находится аудитория, должна решить задачу: каким претендентам предоставить аудиторию, а каким отказать, обеспечив себе максимальный доход (считается, что каждый из претендентов платит одну и ту же сумму за каждое использование аудитории, независимо от времени аренды).

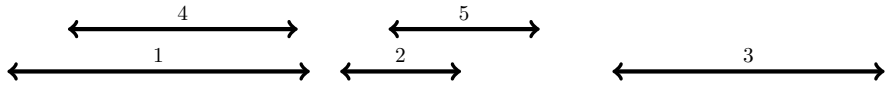


Рис. 2.7. Задача об интервалах: первый пример запросов

Эту задачу предлагается решать жадным алгоритмом. Так как жадные алгоритмы вперёд не заглядывают, мы должны выбрать какую-либо *стратегию* для принятия решения на каждом шаге.

Все варианты жадной стратегии основываются на следующем:

- упорядочить отрезки по какому-либо признаку;
- рассматривать отрезки по одному. Если выбранный отрезок не перекрывается с каким-либо из уже внесённых в выходное множество, то добавить его в это множество.

Жадность алгоритма здесь заключается в том, что каждый раз, когда мы видим подходящий вариант (рассматривая очередной отрезок), то сразу его хватаем.

Принципов упорядочивания можно выбрать несколько, но, как оказывается, не все из них одинаково полезны.

Здравый смысл может подсказать администрации выбирать претендентов с самым коротким временем аренды — раз все платят одинаково, почему бы не заработать побольше денег за одно и то же время?

1. Упорядочиваем по длительности и сначала выберем самые короткие отрезки (рис. 2.8).

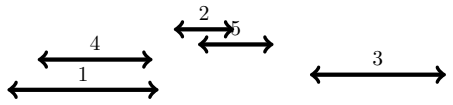


Рис. 2.8. Задача об интервалах: упорядочиваем запросы по длине

Итоговая расстановка даёт нам три отрезка (рис. 2.9):

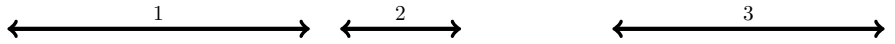


Рис. 2.9. Задача об интервалах: упорядочивание по длине после удаления лишних

2. Вторая интуитивно понятная стратегия — зачем аудитории простаивать, пусть её арендует претендент, предложивший самое раннее время. Если сформулировать эту стратегию в терминах отрезков, то мы выберем упорядочивание отрезков по левой границе и выбор самого левого из них (рис. 2.10).

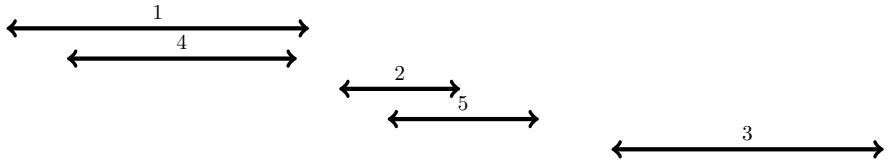


Рис. 2.10. Задача об интервалах: упорядочивание по левой границе

Итоговая расстановка тоже даёт нам три отрезка (рис. 2.11).

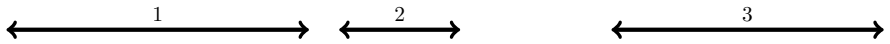


Рис. 2.11. Задача об интервалах: упорядочивание по левой границе после удаления лишних

3. Ещё одна интуитивно понятная стратегия — выбираем того претендента, который раньше всего освободит аудиторию, для того, чтобы у других осталось больше времени. В терминах отрезков — упорядочивание по правой границе (рис. 2.12).

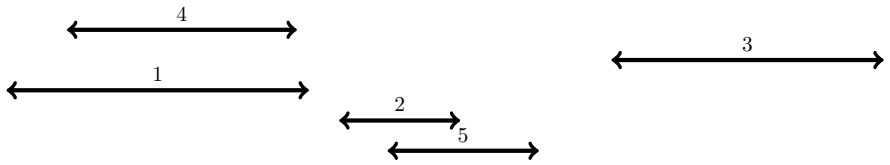


Рис. 2.12. Задача об интервалах: упорядочивание по правой границе

Итоговая расстановка опять даёт нам три отрезка, и это решение — верное.

Что же, видимо, все стратегии равнозначны? Все способы упорядочивания годятся?

А что насчёт такой расстановки (рис. 2.13)?

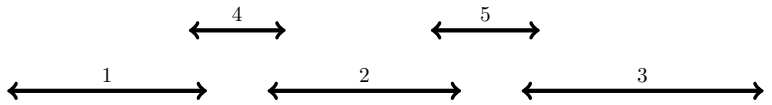


Рис. 2.13. Задача об интервалах: второй пример

Первый вариант (сначала самые короткие) даст нам решение, приведённое на рис. 2.14. Очевидно, оно неверное.

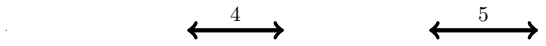


Рис. 2.14. Задача об интервалах: упорядочивание по длине — неверное решение

Второй и третий способы дают правильное решение (рис. 2.15):

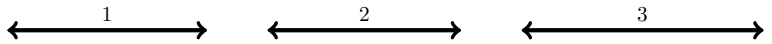


Рис. 2.15. Задача об интервалах: верное решение для второй и третьей стратегий

Можно сказать, что, выбирая стратегию для жадного алгоритма, мы устраиваем соревнование между алгоритмами-претендентами. На каждой из проверок какой-то алгоритм может показать худшие результаты, тогда в следующем раунде соревнования он принимать участие не будет. А пока мы делаем вывод, что первый вариант нам не даёт точного решения во всех случаях и выбывает именно он.

Чтобы такое состязание стратегий когда-то закончилось, мы должны найти такие исходные данные, которые могут опровергнуть какой-либо из вариантов. Можно сказать, что, придумывая стратегии для жадных алгоритмов, мы придумываем гипотезы, обосновывающие какую-то теорию. Как и в науке, ценным экспериментом является не тот, который подтверждает теорию, а тот, который опровергает ложные гипотезы.

При поиске ситуаций, когда мы хотим найти опровергающий какую-либо гипотезу вариант, мы можем наткнуться на следующее расположение отрезков:

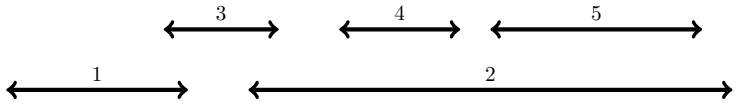


Рис. 2.16. Задача об интервалах: третий пример

Упорядочивание по началу отрезка даёт нам следующее (рис. 2.17):

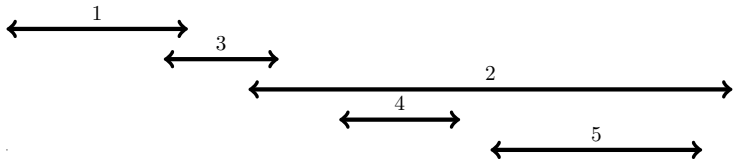


Рис. 2.17. Задача об интервалах: третий пример, упорядочивание по началу

И, соответственно, решение (рис. 2.18):

Рис. 2.18. Задача об интервалах: третий пример, решение

Упорядочивание по концу отрезка даёт нам (рис. 2.19):

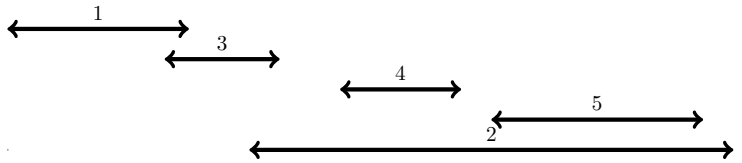


Рис. 2.19. Задача об интервалах: третий пример, упорядочивание по концу

И это приводит к верному решению (рис. 2.20):

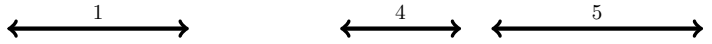


Рис. 2.20. Задача об интервалах: третий пример, верное решение

Как доказать, что данный алгоритм верно решает задачу?

Предложенное решение — устроить соревнование между стратегиями — к сожалению, доказательной силой не обладает. Мы можем только показать, что какой-то алгоритм на каком-то наборе тестов ведёт себя лучше, чем другие. Корректность стратегии такой метод доказать не может. Доказательство можно проводить по индукции. Сначала убедимся в том, что оптимальных решений для данной задачи может быть несколько. Это легко проверить.

Первый шаг — доказательство того, что существует оптимальное подмножество отрезков, которое содержит первый отрезок, получившийся при применении нашего алгоритма. Будем рассуждать так: пусть в некотором оптимальном подмножестве мы поменяем отрезок с минимальным значением конца на первый, выбранный нами на данном шаге алгоритма. Количество отрезков в выходном подмножестве не изменится — и подмножество останется решением. Для каждой левой границы x имеется оптимальное решение. Введём функцию $f(x)$, значением которой в точке x будет значение

минимального множества непересекающихся отрезков. Разумно предположить, что функция $f(x)$ — невозрастающая (доказать это можно от противного). Таким образом, существует оптимальное подмножество, содержащее первый отрезок.

Второй шаг — удаляем из множества отрезков первый отрезок и все отрезки, пересекающиеся с первым.

Третий шаг — повторяем алгоритм для усечённого множества, в котором снова находим первый отрезок.

Применив метод математической индукции, мы показали, что предложенный жадный алгоритм приводит к одному из оптимальных решений задачи.

> Жадные алгоритмы не заглядывают вперёд. Они повторяют локально оптимальные по какому-либо критерию шаги и надеются, что решение будет глобально оптимальным. Возможно, что найдётся такой локально оптимальный критерий — и общее решение окажется верным. Это бывает отнюдь не всегда, но тщательный выбор критерия может найти приемлемое решение.

2.4 Задача о резервных копиях

Задача (задача о резервных копиях). Имеется распределённая система, состоящая из N хранилищ различной ёмкости, причём в i-м хранилище можно разместить A_i блоков информации.

Хранение одного блока считается надёжным, если имеется две его копии в различных хранилищах. Требуется определить наибольшее количество надёжных блоков, которое можно разместить во всех хранилищах.

Для примера возьмём 4 хранилища размерами `(8, 7, 4, 3)`. В таблице одинаковыми числами отмечены одинаковые блоки. Число (номер блока) не может находиться дважды в одном столбце (хранилище).

8	7	4	3
1	1	2	2
3	3	4	4
5	5	6	6
7	7	-	-
8	8	-	-
9	9	-	-
10	10	-	-
11	-	11 -	-

Нетрудно убедиться, что приведённое решение — оптимально, так как удалось распределить все блоки, в каждом из хранилищ размещена его максимальная ёмкость.

Решение задачи. Попробуем решить жадным алгоритмом. Задачу можно решать многими стратегиями. Какая из стратегий — оптимальная?

Для удобства сведём задачу к эквивалентной: имеется `N` кучек камней, каждым ходом можно выбрать по одинаковому количеству камней из любой пары кучек. Найти такой порядок игры, при котором останется минимальное количество камней.

Первая стратегия: выбрать две наименьших кучи и взять из них одинаковое наибольшее количество камней. Для приведённой расстановки мы должны выбрать две кучи, 4 и 3, и взять из них по 3 камня. Затем из двух наименьших куч, 7 и 1, взять по одному камню. В результате с двумя оставшимися кучами, по 8 и 6 камней, уже ничего сделать нельзя — и получившееся решение оптимальным не является.

8	7	4	3
8	7	1	0
8	6	0	0
2	0	0	0

Вторая стратегия: выбрать наибольшую и наименьшую кучи и взять из них одинаковое наибольшее количество камней.

8	7	4	3
5	7	4	0
5	3	0	0
2	0	0	0

Опять не повезло — и оптимальное решение не достигнуто.

Третья стратегия: выбрать две наибольших кучи и взять из них одинаковое наибольшее количество камней.

8	7	4	3
1	0	4	3
1	0	1	0
0	0	0	0

На этот раз решение оказалось оптимальным. Как обычно в задачах на жадность, требуется или доказательство корректности стратегии, или контрпример.

Новая попытка:

8	7	7	6	5
1	0	7	6	5
1	0	1	0	5
0	0	1	0	4
0	0	0	0	3

И снова неудача. Есть ли вообще нужная стратегия?

Массовые неудачи могут подсказать нам, что, возможно, стоило бы так уменьшать количество камней в кучах, чтобы эти числа были одного порядка.

Четвёртая стратегия: выбрать наибольшую и наименьшую кучи и взять из них по одному камню.

Первая расстановка приводит к корректному результату:

8	7	4	3
7	7	4	2
7	6	4	1
6	6	4	0
5	6	3	0
5	5	2	0
4	5	1	0
4	4	0	0
0	0	0	0

И вторая расстановка — тоже:

8	7	7	6	5
7	7	7	6	4
7	7	6	6	3
7	6	6	6	2
6	6	6	6	1
6	6	6	5	0
6	6	5	4	0
6	5	5	3	0
5	5	5	2	0
5	5	4	1	0
5	4	4	0	0
4	4	3	0	0
4	3	2	0	0
3	3	1	0	0
3	2	0	0	0
1	0	0	0	0

Создаётся впечатление, что найден если не оптимальный алгоритм, то близкий к нему. Давайте докажем, что найденный алгоритм оптимален:

- Очевидно, что на каждой «большой» итерации наименьшая из куч опустошается, так как из неё по условиям алгоритма всегда производится взятие.
- Когда останется три кучи $A \geqslant B \geqslant C$, возможны следующие ситуации:
  a) $A > B + C$. Тогда ответ: $A - B - C$. Так как на каждом ходе A оставалось наибольшим, на каждом из ходов, приведших к позиции, происходило вычитание из A, это значит, что A больше суммы всех оставшихся, что, очевидно, даёт верное решение.
  b) $A = B + C$. Тогда результат равен нулю.

c) $A < B + C$. Тогда, после некоторой, возможно, нулевой последовательности ходов достигается ситуация, когда $A' = B' > C'$, которая сведётся к позиции $A' - \left\lfloor \frac{C'}{2} \right\rfloor, A' - \left\lceil \frac{C'}{2} \right\rceil$. В зависимости от чётности суммы результат будет равен либо нулю, либо единице.

Таким образом мы показали, что предложенная стратегия всегда приводит к оптимальному решению.

2.5 Ещё о рюкзаке

Вернёмся к задаче о рюкзаке из первой лекции. Мы уже знаем, что одно из точных решений имеет сложность $O(2^N)$. Можно попробовать найти приближённое решение задачи, используя жадный алгоритм. Формализуем условия задачи.

Задача (задача о рюкзаке). Пусть имеется N предметов, стоимость i-го предмета v_i, а масса w_i. Найти набор предметов с наибольшей стоимостью и массой, не превосходящей заданного W.

Попробуем применить следующий локально оптимальный алгоритм:

1. Расположим предметы в порядке убывания отношения $\frac{v_i}{w_i}$. Пусть они образуют упорядоченное множество B.
2. Установим оставшийся вес $L = W$.
3. Установим множество $S = \emptyset$.
4. Выбираем первый предмет I из упорядоченного множества, вес w_I которого не превосходит L.
5. Если такого предмета нет, то алгоритм закончен.
6. Кладём предмет в рюкзак, удаляя его из B: $B \leftarrow B - I; L \leftarrow L - w_I; S \leftarrow SI$. Переходим к 4-му шагу.

Данный алгоритм конечен и обязательно приведёт к какому-либо решению. Но решение, очевидно, может не быть оптимальным.

Например, при
$N = 3$
$W = 40$
$w_1 = 10; v_1 = 60$
$w_2 = 20; v_2 = 100$
$w_3 = 20; v_3 = 100$

алгоритм выберет последовательно первый и второй предметы. Их суммарная стоимость окажется 160.

Верное решение — выбрать второй и третий предметы. Их суммарная стоимость будет 200.

2.6 Алгоритм Хаффмана

Попробуем использовать жадный алгоритм для решения следующей задачи.

Задача. Имеется исходный текст, состоящий из символов. Поставить в соответствие каждому символу такую последовательность битов (*закодировать его таким образом*), чтобы:

- каждый из встречающихся символов получил свой двоичный код;
- множество кодов было *префиксным*;
- после кодирования исходного текста суммарная длина получившейся последовательности была бы минимальной из возможных.

Например, пусть имеется текст, состоящий из множества из четырёх символов:

```
AAAABAABABABABCBCAAAD
```

Его длина — 21 символ.

Один из возможных кодов — равномерный код, такой, как $(A \to 00)$, $(B \to 01)$, $(C \to 10)$, $(D \to 11)$, когда каждый символ кодируется одинаковым количеством бит.

В приведённом коде на кодирование каждого символа понадобится в точности два бита, и общая длина кода составит 42 бита.

Определение 10. ***Префиксным кодом*** *для набора символов называется код, в котором никакой код символа не начинается с другого кода.*

В частности, код $(A \to 00)$, $(B \to 10)$, $(C \to 01)$, $(D \to 101)$ префиксным не является, так как код символа D начинается с кода символа B.

У префиксных кодов есть важное свойство: они допускают однозначное декодирование. Ещё одно их название — код, удовлетворяющий условиям Фано.

Наша задача — найти *минимальный префиксный код* для множества символов.

2.6.1 Кодирование с помощью дерева.

Ещё раз рассмотрим равномерный код из четырёх символов ($A \to 00$), ($B \to 01$), ($C \to 10$), ($D \to 11$).

Попробуем представить его в виде двоичного дерева[7].

Для определённости пусть левая ветвь дерева будет представлять нулевой бит, правая — единичный. Тогда древесное представление кода будет выглядеть так (рис. 2.21):

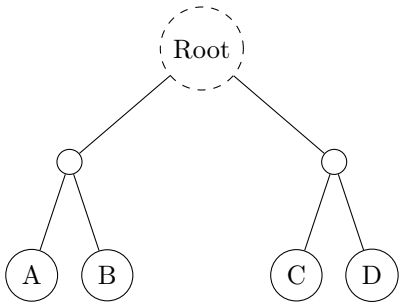


Рис. 2.21. Дерево представления кодов символов

Всегда имеется *корневой* узел. Часть узлов будет *вершинами* или *терминальными узлами*, а часть узлов понадобится только для того, чтобы связать корень дерева и его вершины. Представление кода в виде дерева показывает нам, что для соблюдения условия префиксности не должно находиться символов в узлах, ведущих к другим символам.

2.6.2 Алгоритм Хаффмана

Один из алгоритмов получения оптимальных префиксных кодов был предложен в 1952 году аспирантом `MIT` Дэвидом Хаффманом в его курсовой работе.

Для иллюстрации алгоритма возьмём следующий исходный текст: `AAAABAABABABABCBCAAAD`.

Первый этап: определим встречаемость символов:

[7]Тему деревьев, в частности, двоичных деревьев, мы разовьём подробно в четвёртой и пятой лекциях, а пока она будет только затронута.

- $F_A = 12$
- $F_B = 6$
- $F_C = 2$
- $F_D = 1$

Так как код должен быть префиксным, не имеет смысла располагать символы в узлах, отличных от терминальных. Пусть у всех таких терминальных узлов, содержащих символы, *вес узла* определяется как произведение встречаемости символа на длину пути до корневого узла (*глубину узла*). Тогда под *весом дерева* мы будем понимать сумму всех весов символов. Требуется построить дерево, вес которого минимален из всех возможных, т. е. для которого $\sum_{i=1}^{n} F_i \cdot L_i \to \min$, где L_i — глубина i-го символа.

Попробуем понять, какими свойствами должно обладать оптимальное дерево. Во-первых, если из какого-то узла исходит один путь, то, очевидно, можно поднять вверх это поддерево, сократив на единицу все длины путей до терминальных вершин,

Во-вторых, пустых терминальных вершин быть не должно.

В-третьих, из первых двух пунктов можно сделать вывод, что самое длинное кодовое слово должно быть парным, то есть на самой длинной ветке дерева должно висеть два яблока (символа).

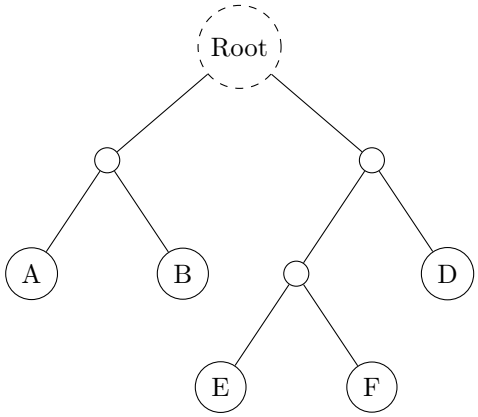


Рис. 2.22. Дерево представления кодов: самые длинные коды — всегда парные

Эти свойства помогут нам построить жадный алгоритм нахождения такого дерева.

1. Перед работой алгоритма создадим (пока пустые) узлы дерева, в которые поместим символы и их встречаемость. Пометим их все как необработанные. Эти узлы пока в дерево не собраны и располагаются отдельно.

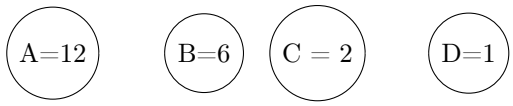


Рис. 2.23. Сборка дерева: начало

2. Так как два самых редко встречающихся символа должны иметь наибольшую длину и находиться на одной ветке, найдём их среди необработанных.
3. Для только что найденных узлов создаём новый, объединяющий их узел, детьми которого они будут. В него прописываем сумму встречаемостей узлов-детей.
4. После объединения узлов сами по себе они перестают нас интересовать, нам интересен узел-родитель. Помечаем узлы или вершины, как уже обработанные (отправляем вниз), а вновь созданный узел — как необработанный.
5. На каждом шаге алгоритма количество необработанных узлов уменьшается на единицу. Если необработанных не осталось, то алгоритм завершён.
6. Переходим к шагу 2.

Попробуем «прогнать» алгоритм для рассмотренного ранее множества символов.

Символы C и D реже всего встречаются, поэтому мы создаём «суперузел» CD, в который помещаем число 3 как сумму чисел 2 и 1, чисел из узлов C и D соответственно. Помечаем узлы C и D как обработанные (рис. 2.24).

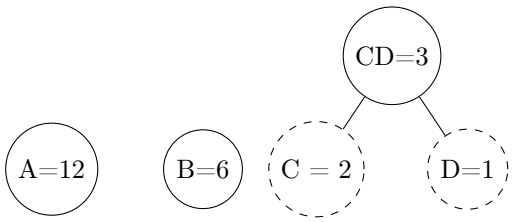


Рис. 2.24. Сборка дерева: создали суперузел `CD`

Для удобства опять выстроим все необработанные узлы в одну линию, под ней окажутся обработанные узлы.

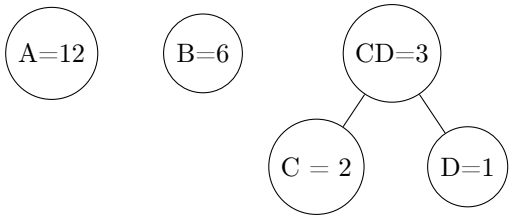


Рис. 2.25. Сборка дерева: осталось обработать `A`, `B` и `CD`

В следующей итерации мы подобным образом объединяем узлы `B` и `CD`, создав узел `BCD`.

Заключительное состояние показано на рис. 2.26.

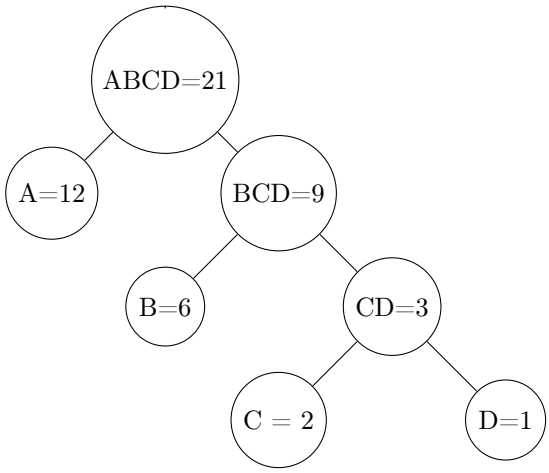


Рис. 2.26. Сборка дерева: обработка закончена

Коды, присвоенные терминальным узлам, содержащим символы, можно получить, обойдя дерево, начиная с вершины, и присвоив, например, пути налево бит 0, а пути направо — бит 1.

- $A \to 0$
- $B \to 10$
- $C \to 110$
- $D \to 111$

Общая длина всех кодовых слов $12 \cdot 1 + 6 \cdot 2 + 2 \cdot 3 + 1 \cdot 3 = 33 < 42$.

Мы изложили данный алгоритм, пока не имея базы для его реализации. Для его работы потребовались понятия: «найти два наименьших элемента в множестве», «построить дерево», «обойти дерево». Несмотря на то, что алгоритм по нашей классификации относится к классу жадных, для его реализации требуются новые структуры данных (упорядоченный массив, дерево) и новые алгоритмы (поиск наименьшего элемента, построение и обход деревьев). Мы вернёмся к алгоритму Хаффмана, как только изучим вспомогательные для него алгоритмы и структуры данных.

2.7 Префиксное дерево

В следующей задаче мы посмотрим, каким образом выбор подходящей структуры данных может повлиять на сложность алгоритма.

2.7.1 Задача о покрытии строки

Задача. Имеется набор «образцов» — $s_i,\ i = 1 \dots N$ — «слов», ни одно из которых не начинается с другого, то есть образующих префиксный код.

Имеется строка p — «предложение».

Требуется определить, можно ли составить предложение p из слов s_i.

Например, если имеется набор слов $s_i = \{ab, ca, ra, dab\}$, то предложение `abracadabra` из них составить можно `ab + ra + ca + dab + ra`, а вот предложения `barca`, `abracadabraa` — нет.

Получив эту задачу, мы задумываемся о том, что же является главными параметрами алгоритма. Размышления наводят нас на то, что N, длина предложения p, точно должна быть главным параметром, а вот что ещё? Пока алгоритм не составлен, больше информации о нём мы не имеем, поэтому пусть, например, вторым главным параметром будет M — сумма длин слов s_i.

Небольшие раздумья приводят нас к следующему жадному алгоритму:

1. Устанавливаем указатель позиции на начало предложения.
2. Из списка слов выбираем то, которое полностью совпадает с подстрокой, начинающейся с указателя в предложении.
3. Если такого слова не найдено, выводим «нет», алгоритм завершён, ответ получен.
4. Если такое слово есть, перемещаем указатель в предложении на длину слова.
5. Если все слова закончились, а совпадения нет, то алгоритм завершён с ответом «нет».
6. Если предложение закончилось, то выводим «да» и завершаем алгоритм.
7. Возвращаемся к пункту 2.

Пусть предложение и слова, которое мы хотим проверить, будут такими:

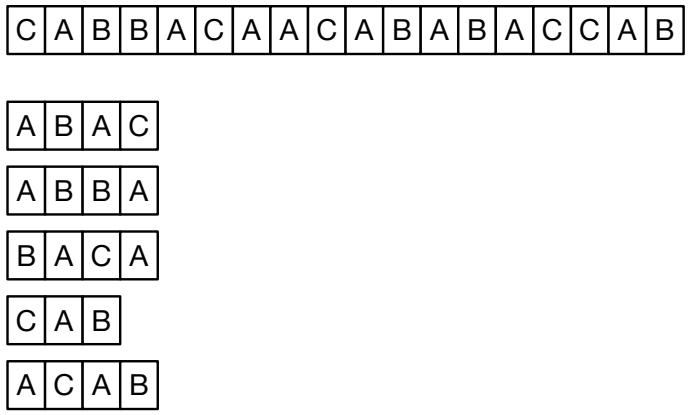


Рис. 2.27. Предложение и слова

Указатель предложения установлен на первый его символ — букву C. Первые три слова в списке успеха не принесли, а вот четвёртое — совпало.

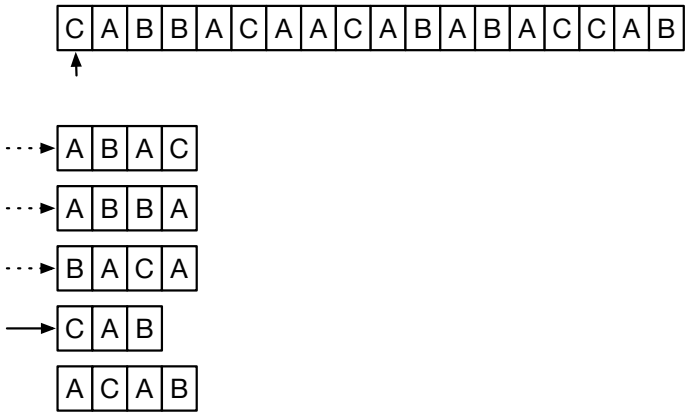


Рис. 2.28. Предложение и слова: поиск первого слова

Перемещаем указатель предложения на длину совпавшего слова и за-

пускаем следующую итерацию.

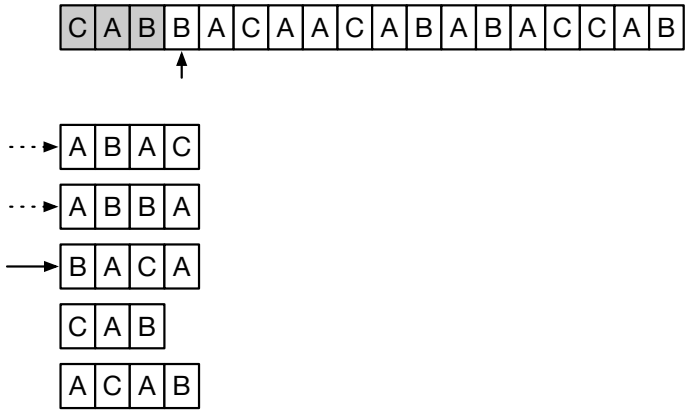


Рис. 2.29. Предложение и слова: поиск второго слова

После нахождения ещё одного слова итерации продолжаются.

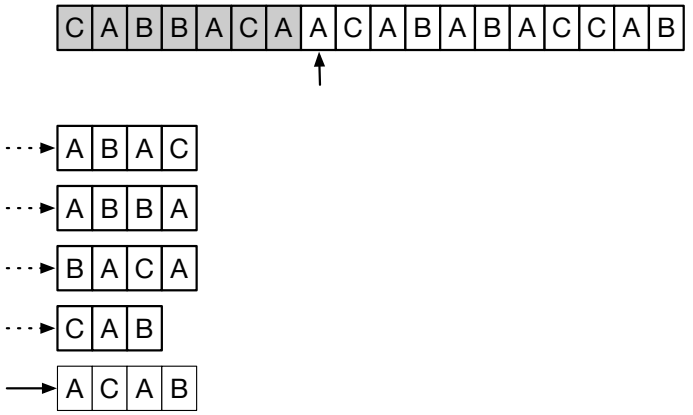


Рис. 2.30. Предложение и слова: поиск третьего слова

Как обычно, после создания алгоритма требуется оценить его сложность.

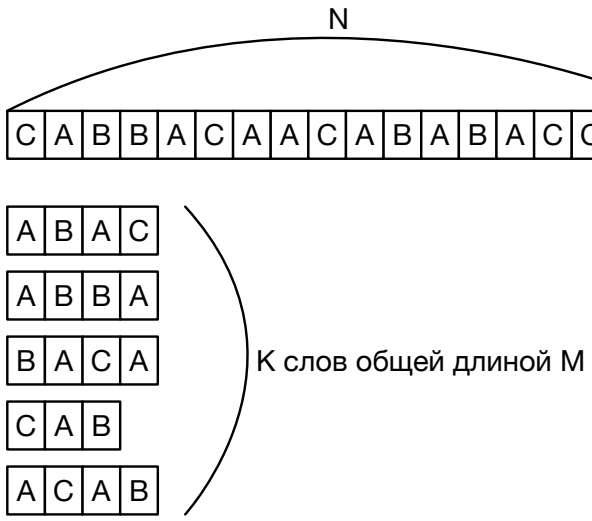


Рис. 2.31. Предложение и слова: оценка сложности алгоритма

Попытка оценить сложность алгоритма вызывает ряд вопросов, на которые нужно как-то ответить. Например, определить, что очередное слово из списка подошло, мы можем, только просмотрев всё слово. Вот определить, что слово не подошло, можно даже с первого символа слова. Сколько попыток поиска в слове будет происходить в среднем? Можно ли считать, что в среднем на каждое слово длины L придётся $\frac{L}{2}$ попыток? Точный ответ на этот вопрос зависит от функции распределения слов и букв в словах и является достаточно сложной самой по себе комбинаторной задачей. Нам требуется общая *оценка* сложности алгоритма, поэтому грубо оценим количество попыток на слово длины L как $\frac{L}{2}$.

Подобный вопрос: что такое «средняя длина слова»? Оценим её в $L = \frac{M}{K}$, где K — число слов. Нам понадобился ещё один главный параметр.

На одном этапе поиска мы в среднем (опять это не очень определённое слово) перебираем $\frac{K}{2}$ слов, каждое из которых имеет среднюю длину L,

при этом один этап продвигает нас в среднем на L позиций в предложении. Исходя из этого, можно *оценить* среднее количество этапов $T = \frac{N}{L}$.

Итого, общую сложность алгоритма в придуманном нами «среднем» сценарии можно оценить как $F = \frac{N}{L} \times \frac{K}{2} \times \frac{L}{2} = \frac{NK}{4} = O(NK)$.

Не кажется ли результат парадоксальным и неверным? Почему сложность алгоритма не зависит от общей длины всех слов?

Можно успокоить себя, рассмотрев крайние сценарии. Например, для $K = 1$, то есть при поиске покрытия ровно одним словом длины M (пусть N делится нацело на M), каждый успешный поиск продвигает нас по предложению на M позиций. Наибольшее количество поисков составляет $\frac{N}{M}$, в каждом из которых сравнивается M символов, что даёт сложность в $F = O(N)$. Результат оценки сложности теперь кажется более правдоподобным.

Исследование задачи для поиска другого алгоритма

Исследование сложности предложенного алгоритма дало нам понять, что для большого K эффективность алгоритма падает. Каждый раз после создания нового алгоритма мы ставим себе вопрос: имеется ли более эффективное решение? Неэффективность только что разработанного алгоритма состоит в том, что, обнаружив несовпадение с одной подстрокой, ничего не получаем для следующих.

Попробуем изменить структуру данных, усложнив представление тех строк, которые мы ищем. Это даст нам возможность проводить поиск параллельно по всем словам. Немного расширим дерево, которое мы использовали для решения задачи о кодировании Хаффмана, обобщив его до *префиксного дерева*. Алгоритм останется жадным, но использование другой структуры данных поменяет его сложность.

Префиксное дерево

Дерево, которое мы строили в предыдущей задаче (кодирование по Хаффману), уже было префиксным, но оно позволяло кодировать только последовательностями из нулей и единиц, так как каждый узел имел не более двух потомков. Идея о том, что каждая ветка однозначно определяет код, позволяет иногда уменьшить количество данных, требуемых для представления. Не умаляя общности, положим, что из каждого узла всегда выходит ровно три ветви, `A`, `B` и `C`. Каждая из веток приходит либо в узел, либо в вершину, либо в никуда.

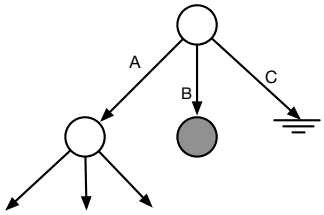


Рис. 2.32. Узел префиксного дерева

Рисовать ветви, уходящие в никуда, мы больше не будем:

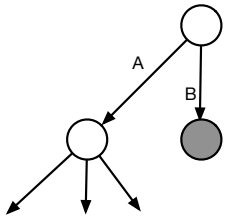


Рис. 2.33. Узел префиксного дерева упрощённо

Построение префиксного дерева

Пусть нам предоставлен следующий набор слов: `ABAC`, `ABBA`, `BACA`, `CAB`, `ACAB`.

Строим дерево для каждого слова посимвольно. Для первого слова, `ABAC`, дерево будет таким:

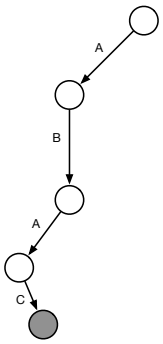


Рис. 2.34. Префиксное дерево после вставки первого слова

После добавления слов `ABBA` и `BACA` оно станет таким:

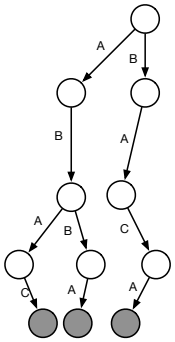


Рис. 2.35. Префиксное дерево после вставки трёх слов

Заключительное дерево после добавления всех слов из словаря:

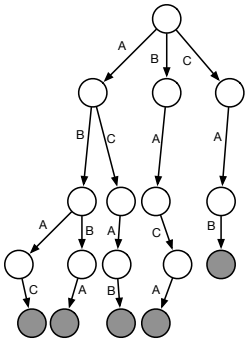


Рис. 2.36. Префиксное дерево после вставки всех слов

Алгоритм построения префиксного дерева оказывается достаточно простым.

1. Создаём вершину дерева — узел с пустыми ветвями.
2. Для каждого слова исполняем:
   (a) Устанавливаем указатель в вершину дерева.
   (b) Считываем очередную букву.
   (c) Если текущий узел не содержит нужной ветви, создаём эту ветвь и пустой узел на ней.
   (d) Переходим в нужную ветвь.
   (e) Если узел уже серый или не пустой, завершаем алгоритм с неудачей.
   (f) Если слово закончилось, то помечаем узел серым цветом.
   (g) Переходим к 2.2b
3. Завершаем алгоритм с успехом.

Оценка сложности алгоритма построения префиксного дерева

Один шаг алгоритма продвигает нас ровно на один символ в образце. А так как повторно образцы не обрабатываются, то количество операций есть $O(M)$, где M — сумма длин образцов.

Поиск с помощью префиксного дерева

Поиск по получившемуся дереву стал совсем простым:

C	A	B	B	A	C	A	A	C	A	B	A	B	A	C	C	A	B

Рис. 2.37. Предложение, в котором мы ищем слова

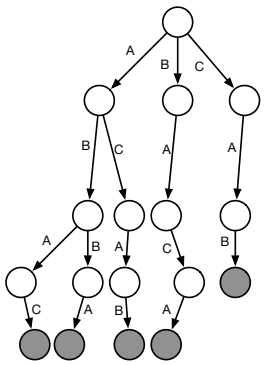


Рис. 2.38. Префиксное дерево, с помощью которого мы ищем слова

1. Устанавливаем указатели на начало строки и на вершину дерева.
2. Если указатель стоит за последним символом в строке, то:
   (a) Если указатель в дереве находится в корне — алгоритм завершён с успехом.
   (b) Иначе алгоритм завершён с неудачей.
3. Считываем очередной символ строки и передвигаем указатель.
4. Если такой ветви дерева нет — завершаем алгоритм с неудачей.
5. Переходим по дереву по соответствующей ветке.
6. Если мы попали в серый узел — возвращаемся в корень дерева.
7. Переходим к пункту 2.

Оценка сложности алгоритма поиска с использованием префиксного дерева

Каждое перемещение символа в предложении приводит к перемещению указателя в дереве. Сложность каждого перемещения постоянна ($O(1)$). Количество перемещений в случае успеха равно N, где N — длина строки. Итого: сложность алгоритма поиска — $O(N)$.

Общий алгоритм состоит из двух этапов — построения дерева и поиска по дереву, следовательно, общая сложность алгоритма — $(N + M)$. Вспомнив, что сложность первой версии алгоритма была $O(N \times K)$, убеждаемся, что, использовав более сложную структуру данных, мы получили более эффективный алгоритм.

2.8 Строки

Здесь и далее под *строкой* мы будем понимать структуру данных, применяемую для хранения текстовой информации. В языке `Си` под строкой понимают любую последовательность байтов, заканчивающуюся нулевым. Это свойство строк `Си`, с одной стороны, позволяет весьма эффективно реализовывать некоторые алгоритмы, но это же свойство часто небезопасно. Самым большим недостатком такого представления является то, что для определения длины строки требуется её полный просмотр, а эта операция весьма востребована. Например, для того, чтобы в языке `Си` получить строку `s3`, состоящую из конкатенации (слияния) строк `s1` и `s2`, требуется:

1. Определить длину строки `s1`: `size_t l1 = strlen(s1);`.
2. Определить длину строки `s2`: `size_t l2 = strlen(s2);`.
3. Выделить участок памяти, достаточный для того, чтобы туда поместились обе строки: `char *s3 = malloc(l1+l2+1);`. Мы не должны забывать, что нужно зарезервировать в том числе и место под заключительный ограничительный нулевой байт.
4. Скопировать первую строку в результирующую: `strcpy(s3,s1);`.
5. Дописать вторую строку к концу первой: `strcpy(s3+l1,s2);`. Можно, конечно, воспользоваться функцией `strcat: strcat(s3,s2);`, но `strcat` опять должен был бы определить место, куда разрешено дописывать, пробегая по строке `s3` с самого начала.

Нельзя сказать, что это всё очень эффективно — мы неоднократно пробегали по строкам `s1` и `s2`. Эффективность можно было бы повысить, добавив счётчик длины для строки и организовав бы всё вместе в одну струк-

туру. К счастью, в стандарте C++ это уже сделано в виде класса `string`. Наша операция конкатенации строк теперь смотрится так:

```
string s3 = s1 + s2;.
```

Длины строк являются теперь частью *экземпляра типа*[8], и сложность операции определения длины теперь $O(1)$ вместо $O(N)$ для строк языка Си. Если для каких-то целей нам потребуется строка Си, её можно получить операцией (методом) `c_str()`: `const char *cs1 = s1.c_str();`.

2.8.1 Z-функция

Мы применяем строки в том числе и для того, чтобы определять, содержится ли одна строка в другой, то есть для операций поиска. Как мы убедимся позднее, существует большое количество различных манипуляций, которые мы можем производить над строками, чтобы ускорить дальнейшие операции. Z-функция, которую мы сейчас рассмотрим, поможет нам в ряде алгоритмов обработки строк.

Дадим несколько определений. Пусть имеется строка `s[0..n)`.

Определение 11. ***Подстрокой*** *`sub(s,p,l)` строки `s` называется строка, состоящая из символов `s[p..p+l)`.*

Определение 12. ***Префиксом*** *длины `l` строки `s` называется подстрока `s[0..l)`.*

Определение 13. ***Суффиксом*** *длины `l` строки `s[0...n)` называется подстрока `s[n-l..n)`.*

Определение 14. ***Собственным префиксом*** *строки `s[0..n)` называется префикс длины `l < n`.*

Определение 15. ***Собственным суффиксом*** *строки `s[0..n)` называется суффикс длины `l < n`.*

Определение 16. ***Z-функция*** *от строки `s` и позиции `p` есть длина наибольшей подстроки строки `s`, начинающейся в позиции `p`, совпадающей с собственным префиксом строки `s`.*

[8] здесь мы погружаемся в объектное программирование и должны осознавать, что слово `string` теперь является именем типа объекта, а вот отдельные строки, такие как `s1`, `s2` и `s3` — экземплярами типа `string`.

Например, вот значения Z-функции для строки `abrashvabracadabra`:

a	b	r	a	s	h	v	a	b	r	a	c	a	d	a	b	r	a
0	0	0	1	0	0	0	4	0	0	1	0	1	0	4	0	0	1

Наша цель — разработать эффективный алгоритм решения задачи построения Z-функции от строки.

Обычная практика при разработке алгоритмов — вначале реализовать простейший, но не обязательно самый эффективный вариант. Доказать корректность такого решения может оказаться существенно проще, чем более сложного. При написании программы, реализующей алгоритм, простейшее решение может выступать как эталонное. Алгоритм не считается реализованным корректно до тех пор, пока хотя бы в одном случае, хотя бы на одном наборе входных данных результат его работы отличается от эталонного.

Z-функция: итерация 1

Корректность следующего алгоритма не вызывает сомнений, так как он просто реализует определение Z-функции:

1. Обнулить выходной массив `ret`.
2. Для всех значений `j` от 1 до размера строки делать:
   (a) Установить `ret[j]` равным длине максимальных совпадающих подстрок, начинающихся с `0` и с `j`.

Попробуем исполнить этот алгоритм (*прогнать* его) для различных входных данных. Стрелка над строкой на рисунке показывает совпавший префикс, стрелка под строкой — совпавшую часть подстроки. Начала и концы стрелок показывают позиции сопоставления.

Сопоставление для `p=1` даёт в результате 0.

Рис. 2.39. Z-функция: попытка сопоставления для `p=1`

Сопоставление для p=3 даёт совпадение одного символа.

Рис. 2.40. Z-функция: попытка сопоставления для p=3

Для p=7 сопоставилось четыре символа.

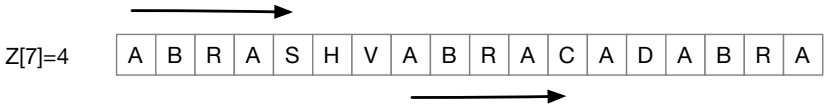


Рис. 2.41. Z-функция: попытка сопоставления для p=7

Эталонная программа, реализующая алгоритм, проста, функция z1 возвращает вектор Z-функции для строки-входного аргумента.

```
vector<int> z1(string const &s) {
  vector<int> ret(s.size());
  for (size_t j = 1; j < s.size(); j++) {
    size_t p = j;
    while (p < s.size() && s[p] == s[p-j])
      p++;
    ret[j] = p-j;
  }
  return ret;
}
```

Опять возникает три ключевых вопроса:

1. Корректен ли алгоритм?
2. Какова его сложность?
3. Можно ли его улучшить?

Ответ на первый вопрос — да, данная функция просто использует определение Z-функции.

Для ответа на второй вопрос можно оценить сложность, выявив наихудший случай, именно он даст нам оценку сверху по количеству требуемых операций. Ряд попыток прогона алгоритма показывает нам, что наихудший случай — строка, состоящая из одинаковых символов.

Сопоставление для `p=1` требует $N-1$ сравнение символов:

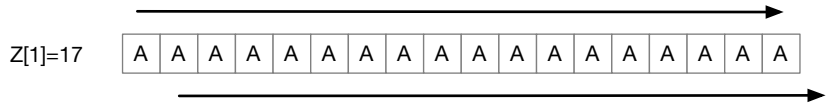


Рис. 2.42. Z-функция: наихудший случай. `p=1`

Сопоставление для `p=2` требует $N-2$ сравнения символов:

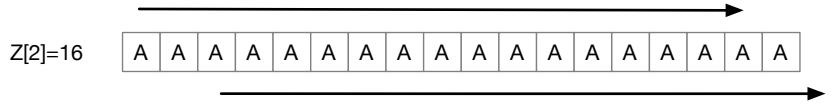


Рис. 2.43. Z-функция: наихудший случай. `p=2`

Общая сложность алгоритма составляет

$$T(N) = (N-1) + (N-2) + \cdots + 1 = O(N^2).$$

Пока мы не знаем, много это или мало. Возможно, для данного алгоритма это и есть предельная сложность? Чтобы понять, можно ли алгоритм улучшить, требуется понаблюдать за его поведением.

Рассмотрим строку ABACABACDABACADCAA и случай поиска значения функции для p=4:

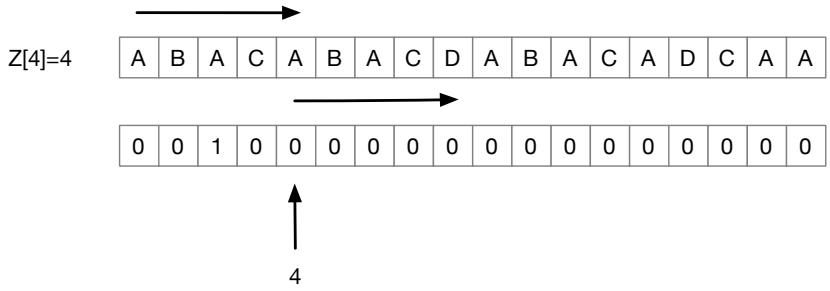


Рис. 2.44. Z-функция: поиск инварианта

Введём понятие *последняя сопоставленная подстрока*. Значение Z-функции для p=4 оказалось равно 4. Почему же сопоставление для p=5 даёт 0?

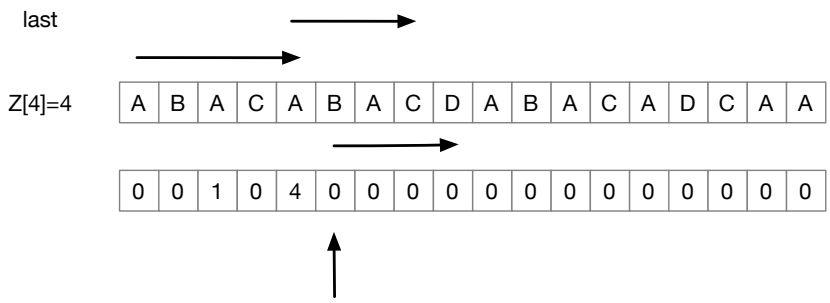


Рис. 2.45. Z-функция: последняя сопоставленная подстрока

Эврика! Если индекс p попадает внутрь *подстроки сопоставления*, то сдвигаем его сразу за правую границу. Нам эта подстрока больше не нужна. Если p *не* попадает внутрь подстроки сопоставления, то ищем как обычно.

В эталонный алгоритм вносим два изменения:

1. начало поиска теперь может сдвинуться сразу на несколько позиций;
2. при выходе `p` за границу подстроки сопоставления меняем эту подстроку на новую.

Z-функция: итерация 2

```
vector<int> z2(string const &s) {
  vector<int> ret(s.size());
  for (int j = 1, l = 0, r = 0; j < s.size(); j++) {
    int p = j > r ? j : j+min(r-j+1, ret[j-l]);
    while (p < s.size() && s[p] == s[p-j])
      p++;
    ret[j] = p-j;
    if (p > r) {
      l = j;
      r = p-1;
    }
  }
  return ret;
}
```

Те же самые три вопроса:

1. Корректен ли алгоритм?
2. Какова его сложность?
3. Можно ли его улучшить?

Сложность алгоритма теперь посчитать труднее.

Инвариант: положение конца подстроки сопоставления изменяется на её длину L. С другой стороны, поиск внутри строки сопоставления тоже составляет $O(L)$. Это означает, что каждый из символов строки обрабатывается за время $O(1)$, что и даёт нам сложность всего алгоритма: $T(N) = O(N)$.

Подумаешь, от $O(N^2)$ мы уменьшили сложность до $O(N)$. Время исполнения программы для разных `N` показывает, что чем больше входная строка, тем больше выигрыш во времени исполнения. В таблице приведён пример прогона алгоритмов на одном из типичных персональных компьютеров на архитектуре `X64`. Время исполнение приведено в секундах.

N	z1	z2
0.5K	0.00017	<0.00001
1K	0.00063	0.00001
2K	0.0021	0.00002
4K	0.0078	0.00004
8K	0.030	0.00009
16K	0.11	0.00014
32K	0.42	0.0026
64K	1.71	0.0051
128K	6.83	0.010
256K	25.8	0.019

Можно ли улучшить этот алгоритм? Вероятно, нет. Так как длина строки равна N — и для построения Z-функции потребуется заполнить не менее N элементов выходного массива, сложность не может быть меньше, чем $O(N)$. А где применяется Z-функция? Это мы увидим в задачах, которые будем разбирать на семинарах.

2.9 Домашние задания

Задача 6. Сумма элементов подмассива

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	256 мегабайт

Имеется массив V целых чисел, состоящий из $1 \leqslant N \leqslant 10^8$ элементов, $-2 \times 10^9 \leqslant V_i \leqslant 2 \times 10^9$.

Подмассивом называют непрерывное подмножество элементов массива, возможно, включающее в себя и полный массив.

Требуется найти наибольшую из возможных сумм всех подмассивов.

Формат входных данных

```
N
V1
V2
...
VN
```

Формат выходных данных

`MaximalSubarraysSum`

Пример

стандартный ввод	стандартный вывод
10 -4 4 3 3 -4 1 2 1 -4 0	10

Задача 7. Длинное сложение/вычитание

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

На вход подаётся три строки. Первая содержит представление длинного десятичного числа (первый операнд), вторая — представление операции, строки `+` или `-`, третья — представление второго операнда.

Длины первой и третьей строки ограничены 1000 символов. Вторая строка содержит ровно один символ.

Требуется исполнить операцию и вывести результат в десятичном представлении.

Формат входных данных

```
123
+
999
```

Формат выходных данных

```
1122
```

Примеры

стандартный ввод	стандартный вывод
232 + -100	132
-100 - 199	-299

Замечание

Постарайтесь реализовать программу таким образом, чтобы ей можно было воспользоваться в дальнейшем, в домашних и контрольных заданиях имеются задачи, в которых потребуется длинная арифметика.

Задача 8. Танец точек

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	256 мегабайт

На прямой располагается $1 \leqslant N \leqslant 10000$ точек с целочисленными координатами $-10^9 \leqslant V_i \leqslant 10^9$. Каждой из точек разрешается сделать ровно одно движение (танцевальное па) в любом направлении на расстояние не больше $0 \leqslant L \leqslant 10^8$ и остановиться на другой позиции. Какое минимальное количество точек может остаться на прямой после окончания танца? После танца все точки, оказывающиеся на одной позиции, сливаются в одну.

Формат входных данных

```
L N
V1
V2
...
VN
```

Формат выходных данных

```
MinimalNumberOfPoints
```

Пример

стандартный ввод	стандартный вывод
10 5 30 3 14 19 21	2

Задача 9. Ровно M простых

Имя входного файла:	стандартный ввод
Имя выходного файла:	стандартный вывод
Ограничение по времени:	2 секунды
Ограничение по памяти:	24 мегабайта

Требуется найти такое наименьшее натуральное число $2 \leqslant K \leqslant 2 \cdot 10^7$, что, начиная с этого числа, среди N натуральных чисел имеется ровно M простых.

Если такого числа не существует или оно больше $2 \cdot 10^7$, вывести -1.

Формат входных данных

M N

Формат выходных данных

K или -1

Примеры

стандартный ввод	стандартный вывод
4 10	3
3 15	14

Задача 10. Периодическая дробь

Имя входного файла:	стандартный ввод
Имя выходного файла:	стандартный вывод
Ограничение по времени:	1 секунда
Ограничение по памяти:	256 мегабайт

Выведите десятичное представление рациональной правильной дроби. Если в представлении присутствует период, то нужно вывести первое его вхождение в круглых скобках.

Формат входных данных

$1 \leqslant N < M \leqslant 150000000$

Формат выходных данных

Десятичное представление числа N/M

Примеры

стандартный ввод	стандартный вывод
6 70	0.0(857142)
17 250	0.068

Лекция 3

Мы уже неоднократно использовали такие выражения, как: «упорядочить отрезки по их правой части», «Упорядочить числа по возрастанию» и т. д. В решаемых подзадачах мы полагали, что у нас имеется какой-либо инструмент для их решений. Настало время познакомиться с ним немного поближе.

3.1 Задача сортировки

Имеется последовательность из n *ключей*.

$$k_1, k_2, \ldots, k_n.$$

Требуется: упорядочить ключи по *не убыванию* или *не возрастанию*.

Это означает: найти перестановку ключей

$$p_1, p_2, \ldots, p_n$$

такую, что

$$k_{p_1} \leqslant k_{p_2} \leqslant \cdots \leqslant k_{p_n}$$

или

$$k_{p_1} \geqslant k_{p_2} \geqslant \cdots \geqslant k_{p_n}.$$

Элементами сортируемой последовательности могут иметь любые типы данных. Обязательное условие — наличие *ключа*.

Возьмём, к примеру, последовательность:

(Москва, 10000000), (Нью-Йорк, 12000000), (Париж, 9000000), (Токио, 20000000), (Лондон, 10000000), (Дели, 9000000)

Пусть ключом будет число жителей. Мы можем упорядочить эту последовательность несколькими способами, так как у нас есть одинаковые ключи. Но, как оказывается, не все способы одинаково полезны и иногда важно обеспечить *устойчивость сортировки*.

3.1.1 Устойчивость сортировки

Алгоритм сортировки *устойчивый*, если он сохраняет относительный порядок элементов с одинаковыми ключами. Например, если в начальной последовательности

(Москва, 10000000), (Нью-Йорк, 12000000), (Париж, 9000000), (Токио, 20000000), (Лондон, 10000000), (Дели, 9000000)

Москва (имеющая одинаковое количество жителей с Лондоном) находилась перед Лондоном, то в устойчивой сортировке она тоже должна остаться перед ним:

(Токио, 20000000), (Нью-Йорк, 12000000), (Москва, 10000000), (Лондон, 10000000), (Париж, 9000000), (Дели, 9000000)

В неустойчивой сортировке они могут поменяться местами:
(Токио, 20000000), (Нью-Йорк, 12000000), (Лондон, 10000000), (Москва, 10000000), (Париж, 9000000), (Дели, 9000000)

Можно отметить ещё одно важное для теории алгоритмов свойство: при наличии одинаковых ключей возможны несколько вариантов отсортированных последовательностей, удовлетворяющих свойству отсортированности, но только устойчиво отсортированная последовательность ровно одна.

3.2 Сортировки сравнением

А как мы определяем, должен ли один элемент находиться ранее другого в отсортированной последовательности? Один из популярных способов — просто сравнить эти два элемента. Сортировки, сравнивающие пары элементов для их упорядочивания, так и называются: *сортировки сравнением*.

Очевидно, что для исполнения алгоритма такой сортировки нам необходимо определить операцию сравнения ключей.

$$a < b$$

Полагается, что

$$not(a < b) \wedge not(b < a) \rightarrow a = b$$

Это необходимое условие для соблюдения *закона трихотомии*: для любых a, b либо $a < b$, либо $a = b$, либо $a > b$.

3.2.1 Понятие *инверсии*

Определение 17. *Инверсия — пара ключей с нарушенным порядком следования.*

В последовательности `{4, 15, 6, 99, 3, 15, 1, 8}` имеются следующие инверсии: `(4,3), (4,1), (15,6), (15,3), (15,1), (15,8), (6,3), (6,1), (99,3), (99,15), (99,1), (99,8), (3,1), (15,1), (15,8)`

Перестановка соседних элементов, расположенных в ненадлежащем порядке, уменьшает инверсию ровно на 1.

Количество инверсий в любом множестве конечно, в отсортированном же равно нулю. Следовательно, количество обменов для сортировки конечно и не превосходит числа инверсий.

Наблюдение за уменьшением инверсий даёт нам алгоритм сортировки пузырьком.

3.2.2 Сортировка пузырьком

Сортировка пузырьком: идея

Похоже, что этот алгоритм знаком всем, кто учился в школе и у кого был предмет «информатика». Он действительно интуитивно понятен, и его реализация — одна из простейших.

Основная идея: до тех пор, пока найдётся пара соседних элементов, расположенных не в надлежащем порядке, меняем их местами.

$$\{\underbrace{10, 4}, 14, 25, 77, 2\}$$

$$\{4, \underbrace{10, 14}, 25, 77, 2\}$$

$$\{4, 10, \underbrace{14, 25}, 77, 2\}$$

$$\{4, 10, 14, \underbrace{25, 77}, 2\}$$

$$\{4, 10, 14, 25, \underbrace{77, 2}\}$$

$$\{4, 10, 14, 25, 2, 77\}$$

Сортировка пузырьком: корректность

Корректность алгоритма сортировки пузырьком прямо следует из определения инверсии:

- Если инверсий в последовательности нет, то алгоритм завершён.
- Если хотя бы одна инверсия в последовательности есть, то на очередном проходе по массиву количество инверсий уменьшается хотя бы на 1.
- Так как количество инверсий в конечной последовательности конечно, то алгоритм обязательно завершается.

Сортировка пузырьком: сложность

Сложность алгоритма посчитать также нетрудно. Так как один проход по массиву имеет сложность $O(N)$, установить, что массив уже отсортирован, быстрее невозможно, поэтому в лучшем случае сложность составляет именно $O(N)$, например, для последовательности {1, 2, 3, 4, 5, 6}.

Худший случай возникает тогда, когда на каждом проходе хотя бы одна пара элементов переставляется. Тогда количество проходов есть $O(N)$, количество сравнений будет $(N-1) + (N-2) + \cdots + 1 = O(N^2)$. Самый худший случай — последовательность в противоположном порядке, такая, как {6, 5, 4, 3, 2, 1}. К $O(N^2)$ сравнениям добавляются достаточно дорогие перестановки. Количество инверсий у этой последовательности максимально, $\frac{N \cdot (N-1)}{2}$, количество перестановок соответствует этому числу. Сложность алгоритма не изменилась — $O(N^2)$, но заметно возрос коэффициент амортизации.

А чему равно количество инверсий случайного массива в среднем? Для его нахождения потребуется привлекать относительно сложный математический аппарат. Те, кого это действительно интересует, могут прочитать это в книгах [4, 2].

В коде, приведённом ниже, алгоритм завершается именно тогда, когда при очередном проходе по массиву не удалось найти инверсий.

```
void bubblesort(int *a, int n) {
  bool sorted = false;
  while (!sorted) {
    sorted = true;
    for (int i = 0; i < n-1; i++) {
      if (a[i] > a[i+1]) {
        int tmp = a[i];
        a[i] = a[i+1];
        a[i+1] = tmp;
        sorted = false;
      }
    }
  }
}
```

Сортировка пузырьком: инвариант

Для этого и других алгоритмов сортировки мы будем находить *инвариант*, соблюдающийся при исполнении алгоритма. Это позволит нам более подробно исследовать алгоритм и определить его применимость. Инвариант: [9] после i-го прохода на верных местах находится не менее i элементов «справа»:

[9] Знакомые с олимпиадной математикой могут возразить: такое условие называют *полуинвариант*, и с математической точки зрения они будут правы. В теории алгоритмов инвариантом называют любое логическое выражение, остающееся истинным во время исполнения алгоритма или его части.

$$\{\boxed{5,3},15,7,6,2,11,13\}$$
$$\{3,\boxed{5,15},7,6,2,11,13\}$$
$$\{3,5,\boxed{15,7},6,2,11,13\}$$
$$\{3,5,7,\boxed{15,6},2,11,13\}$$
$$\{3,5,7,6,\boxed{15,2},11,13\}$$
$$\{3,5,7,6,2,\boxed{15,11},13\}$$
$$\{3,5,7,6,2,11,\boxed{15,13}\}$$
$$\{3,5,7,6,2,11,13,\underline{15}\}$$
$$\{3,5,6,2,7,11,\underline{13,15}\}$$
$$\{3,5,2,6,7,\underline{11,13,15}\}$$
$$\{3,2,5,6,\underline{7,11,13,15}\}$$
$$\{2,3,5,\underline{6,7,11,13,15}\}$$

Этот инвариант и дал название алгоритму: на одном проходе самый тяжёлый из ещё не обработанных элементов, подобно пузырьку, «всплывает» на свою позицию (хотя, конечно, немного странно говорить про тяжёлые элементы, что они всплывают, они, скорее, тонут).

Сортировка пузырьком: особенности

Давайте зафиксируем особенности этого алгоритма (в конце главы мы соберём особенности всех алгоритмов сортировки воедино):

- Крайне проста в реализации и понимании.
- Устойчива. Действительно, так как сортировка оперирует только соседними элементами, она при соседних элементах с одинаковыми ключами не способна перебросить один элемент через другой с таким же ключом.
- Сложность в наилучшем случае $O(N)$.
- Сложность в наихудшем случае $O(N^2)$.
- Сортирует на месте *(in-place)*. Это означает, что для сортировки не требуется создавать другой массив размера, сравнимого с исходным. Для реализации требуется всего 3-4 переменных.

3.2.3 Сортировка вставками

Сортировка вставками: идея

Это — ещё один из интуитивно понятных алгоритмов. Его идея заключается в том, что мы должны поддерживать следующий инвариант: после i-го прохода по массиву в первых i позициях будут находиться отсортированные i элементов первоначального массива. Алгоритм неформально можно описать так:

- Первый проход — нужно поместить самый лёгкий элемент на первую позицию.
- В i-м проходе ищется, куда поместить очередной a_i внутри левых i элементов.
- Элемент a_i помещается на место, сдвигая вправо остальные внутри области $0 \dots i$.

Сортировка вставками: инвариант и корректность

Инвариант сортировки вставками:

$$\underbrace{a_1, a_2, \dots, a_{i-1}}, a_i, \dots, a_n$$
$$a_1 \leqslant a_2 \leqslant \dots \leqslant a_{i-1}$$

На шаге i имеется упорядоченный подмассив $a_1, a_2, \dots, a_{i-1}$ и элемент a_i, который надо вставить в подмассив без потери упорядоченности.

После первого шага инвариант соблюдается. Соблюдение инварианта после остальных шагов показывает нам корректность алгоритма.

```
void insertion(int *a, int n) {
  for (int i = n-1; i > 0; i--) {
    if (a[i-1] > a[i]) {
      int tmp = a[i-1]; a[i-1] = a[i]; a[i] = tmp;
    }
  }
  for (int i = 2; i < n; i++) {
    int j = i;
    int tmp = a[i];
    while (tmp < a[j-1]) {
      a[j] = a[j-1]; j--;
    }
```

```
    a[j] = tmp;
  }
}
```

Сортировка вставками: сложность

- Худший случай — упорядоченный по убыванию массив. Тогда цикл вставки на i-й итерации каждый раз будет доходить до 1-го элемента, что даст нам сложность $O(i)$ на итерацию.
- Для вставки элемента a_i потребуется $i - 1$ итерация.
- Позиции ищутся для $N - 1$ элемента. Общее время

$$T(N) = \sum_{i=2}^{N} c(i-1) = \frac{cN(N-1)}{2} = O(N^2).$$

- Лучший случай — упорядоченный по возрастанию массив. $T(N) = O(N)$.
  Так как каждый раз приходит элемент a_{i+1}, больший элемента a_i, то операция перемещения не производится ни разу, и сложность вставки каждого из элементов, кроме первого, оказывается $O(1)$.

Сортировка вставками: особенности

- Сортировка упорядоченного массива требует $O(N)$. Если массив частично упорядочен, то большое количество перемещения обработанной позиции будет проводиться за время $O(1)$, что уменьшит общую сложность алгоритма.
- Сложность в худшем случае $O(N^2)$.
- Алгоритм устойчив. При перестановке элементов не существует ситуаций, когда элементы, имеющие одинаковые ключи, могут перескакивать друг через друга.
- Число дополнительных переменных не зависит от размера (*in-place*).
- Позволяет упорядочивать массив при динамическом добавлении новых элементов — *online*-алгоритм. Если массив уже отсортирован, то добавление нового элемента будет происходить со сложностью $O(N)$. Это свойство иногда оказывается очень важным. Отнюдь не все эффективные алгоритмы могут похвастаться таким поведением. Именно по этой причине сортировку вставками используют в качестве базового алгоритма в весьма популярной ныне сортировке TimSort (о которой мы поговорим на семинарах).

3.2.4 Сортировка Шелла

Давайте вернёмся к сортировке пузырьком и понаблюдаем за поведением сортируемых данных. Мы уже знаем, что один обмен неупорядоченных *соседних* элементов уменьшает инверсию на 1.

Наихудшим случаем для этого алгоритма будет упорядоченная в противоположном порядке последовательность:

$$I(\{8,7,6,5,4,3,2,1\}) = \frac{8 \cdot 7}{2} = 28.$$

Но почему обязательно нужно обменивать местами только соседние элементы? Может быть, попробовать обменивать элементы с расстоянием $d > 1$? Этот вопрос задал себе американский математик Дональд Шелл в 1959 году.

Попробуем пробежаться по массиву обычным проходом метода пузырька, но пусть будут проверяться элементы, находящиеся на расстоянии $d = 4$.

Обмен первого элемента с пятым даёт последовательность, инверсия которой

$$I(\{4,7,6,5,8,3,2,1\} = 21.$$

За один шаг инверсия уменьшилась с 28 до 21, то есть на 7.

Проделывая эту же операцию с вторым и шестым элементами, с третьим и седьмым и так далее, получаем новую последовательность:

$$\{8,7,6,5,4,3,2,1\}$$
$$\{4,7,6,5,8,3,2,1\}$$
$$\{4,3,6,5,8,7,2,1\}$$
$$\{4,3,2,5,8,7,6,1\}$$
$$\{4,3,2,1,8,7,6,5\}$$

Проделаем то же самое, уменьшив шаг d до двух:

$$\{4,3,2,1,8,7,6,5\}$$
$$\{2,3,4,1,8,7,6,5\}$$
$$\{2,1,4,3,8,7,6,5\}$$
$$\{2,1,4,3,6,7,8,5\}$$
$$\{2,1,4,3,6,5,8,7\}$$

Обратите внимание на то, что при классической сортировке пузырьком самый маленький из элементов, 1, за два прохода переместился бы на две единицы влево, а если он стоял на крайней правой позиции, то количеством проходов, меньшим, чем $N-1$, обойтись было бы невозможно. У нас же единица после второго прохода уже оказалась на второй позиции.

Третий проход при $d=1$ даёт следующие шаги:

$$\{2,1,4,3,6,5,8,7\}$$
$$\{1,2,4,3,6,5,8,7\}$$
$$\{1,2,3,4,6,5,8,7\}$$
$$\{1,2,3,4,5,6,8,7\}$$
$$\{1,2,3,4,6,5,7,8\}$$

После всего трёх проходов удалось отсортировать самый неудобный для сортировки пузырьком массив. Сам Шелл предложил выбирать шаги сортировки в виде убывающей геометрической прогрессии с шагами $\frac{N}{2}, \frac{N}{4}, \ldots, 1$. Как вследствие оказалось, предложенный Шеллом вариант оказался не лучшим из возможных. Да, в среднем сложность алгоритма уменьшилась, но в худшем случае (для специально сформированной последовательности) она осталась $O(N^2)$.

Было предложено (и до сих пор предлагается!) много последовательностей шагов. Удивительно, что от выбора последовательности зависит сложность алгоритма!

Например, для последовательности $a=\{1,4,13,\ldots,3a_{n-1}+1,\ldots\}$ сложность в среднем составляет $O(N^{\frac{4}{3}})$, а в худшем — $O(N^{\frac{3}{2}})$.

Между тем, для последовательности $d=\{1,8,23,77,\ldots,4^{i+1}+3\cdot 2^i+1,\ldots\}$ сложность в наихудшем случае составляет $O(N^{\frac{4}{3}})$. Д. Кнут много страниц своей книги [3] посвятил исследованию сложности этой сортировки. Поиски идеальной последовательности продолжаются.

Сортировка Шелла (вариант d=1,4,13,...)

```
void shellsort(int  *a, int n) {
  int h;
  for (h = 1; h <= n / 9; h = 3*h + 1)
    ;
  for ( ; h > 0; h /= 3) {
    for (int i = h; i < n; i++) {
```

```
      int j = i;
      int tmp = a[i];
      while (j >= h && tmp < a[j-h]) {
        a[j] = a[j-h];
        j -= h;
      }
      a[j] = tmp;
    }
  }
}
```

В первых строках выбирается начальный шаг, который затем уменьшается по приведённому закону. Обратите внимание, что данная реализация весьма похожа на сортировки вставками — ищется место очередного элемента, и он путешествует в нужном направлении.

Сортировка Шелла: особенности

- Сортировка упорядоченного массива требует $O(N)$.
- Алгоритм неустойчив. Действительно, при начальных, больших шагах сравнения возможен обмен далеко отстоящих друг от друга элементов. Алгоритм не видит элементов в промежутке между обмениваемыми, поэтому может поменять порядок элементов с одинаковыми ключами.
- Число дополнительных переменных не зависит от размера (*in-place*).
- Низкий коэффициент амортизации и простота реализации делают этот алгоритм конкурентом популярным алгоритмам при не очень больших N.

3.2.5 Сортировка выбором

А почему бы не уменьшить количество обменов элементов между собой? Если мы поставим себе такую цель, то получим сортировку выбором. Идея этой сортировки также интуитивно понятна: на первом шаге мы находим наименьший элемент массива и меняем его с первым. У нас появился упорядоченный набор из одного элемента. Каждый очередной шаг будет добавлять к этому набору по наименьшему из тех элементов, которые пока не входят в набор. На втором шаге — находим наименьший элемент из подмассива, начинающегося с позиции 2, и меняем его с элементом, находящимся на позиции 2, и так далее.

Инвариант: после i итераций упорядочены первые i элементов.

$$\{5, 3, 15, 7, 6, \boxed{2}, 11, 13\}$$
$$\{\underline{2}, \boxed{3}, 15, 7, 6, 5, 11, 13\}$$
$$\{\underline{2, 3}, 15, 7, 6, \boxed{5}, 11, 13\}$$
$$\{\underline{2, 3, 5}, 7, \boxed{6}, 15, 11, 13\}$$
$$\{\underline{2, 3, 5, 6}, \boxed{7}, 15, 11, 13\}$$
$$\{\underline{2, 3, 5, 6, 7}, 15, \boxed{11}, 13\}$$
$$\{\underline{2, 3, 5, 6, 7, 11}, 15, \boxed{13}\}$$
$$\{\underline{2, 3, 5, 6, 7, 11, 13}, \boxed{15}\}$$
$$\{\underline{2, 3, 5, 6, 7, 11, 13, 15}\}$$

Сортировка выбором: особенности

- Во всех случаях сложность $O(N^2)$. Да, к сожалению, найдя минимальный элемент в подмассиве, мы ничего не можем сказать о расположении остальных элементов. На каждой итерации мы должны производить поиск снова. Количество операций в поиске на первой итерации — N, на второй — $N-1$ и т. д. Сумма этих значений и даёт $O(N^2)$.
- Алгоритм можно реализовать и в устойчивом, и в неустойчивом режиме. Если при поиске минимума мы остановимся на самом первом из минимальных элементов, то, очевидно, обменов, нарушающих устойчивость, производиться не будет. В противном случае алгоритм окажется неустойчивым.
- Число дополнительных переменных не зависит от размера (*in-place*).
- Рекордно маленькое по сравнению с другими алгоритмами сортировки количество операций обмена $O(N)$. Это может пригодиться, в частности, для сортировок массивов с очень большими элементами. Если данные, которые мы сортируем, можно сравнивать очень быстро, но для их перемещения требуется большое время, то этот алгоритм оказывается одним из кандидатов.

3.2.6 Нахождение k-й порядковой статистики

Определение 18. *k-й порядковой статистикой массива называется k-й по упорядочиванию величины элемент массива.*

Операция упорядочивания в этом определении явным образом не определяется, поэтому упорядочивать мы можем по любой удобной нам операции.

Минимальный элемент массива — 1-я порядковая статистика при использовании традиционной операции `<`. Максимальный элемент при использовании этой же операции будет иметь порядковую статистику N.

Медиана — «средний» по величине элемент. Примерно половина элементов не больше медианы, примерно половина не меньше, точное значение и точное определение зависит от чётности множества. Обратите внимание, что медиана — это не среднее значение! Например, для множества `{1, 1, 1, 1, 1, 10}` медиана равна 1, а среднее значение — 2.5.

Легко ли найти k-ю порядковую статистику? Рассмотрим частные случаи.

`k=1` Нахождение минимума — очевидно, что сложность $O(N)$.

`k=2` Нахождение второго по величине элемента. Простой способ: хранить значения двух элементов, минимального и второго по величине. Тогда при обработке следующего элемента возможны три варианта: он не больше минимального; он больше минимального, но не больше второго элемента; он больше второго элемента. В первом случае он становится минимальным, а старое значение минимума отправляется во второй элемент. Во втором случае заменяется только значение второго элемента. Ну а в третьем случае ничего делать не надо. Каждая итерация требует до *двух* сравнений.

`k=3` Неужели придётся повторять алгоритм случая `k=2`, расширив его на *три* сравнения? Тогда потребуются три переменные.

`Произвольное k` Требуется ли использовать $O(k)$ памяти и тратить $O(k)$ операций на одну итерацию?

Оказывается, наивный алгоритм не столь уж хорош — и алгоритм нахождения k-й порядковой статистики методом *разделяй и властвуй*, как мы сейчас выясним, значительно эффективнее.

1. Выбираем случайным образом элемент v массива S.
2. Разобьём массив на три подмассива S_l, элементы которого меньше, чем v; S_v, элементы которого равны v,и S_r, элементы которого больше, чем v.
3. Введём функцию $Selection(S, k)$, где S — массив, а k — номер поряд-

ковой статистики.

$$selection(S,k) = \begin{cases} selection(S_l,k), & \text{если } k \leqslant |S_l| \\ v, & \text{если } |S_l| < k \leqslant |S_l| + |S_v| \\ selection(S_r, k - |S_l| - |S_v|), & \text{если } k > |S_l| + |S_v| \end{cases}$$

Как обычно, под операцией $|S|$ имеется в виду операция получения числа элементов множества S, здесь это массив.

Проиллюстрируем алгоритм на конкретных значениях.

Пусть нужно найти $k = 6$ порядковую статистику в массиве $S = \{10, 6, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8\}$.

Выбираем произвольный элемент. Пусть рулетка выпала на элемент со значением «элемент 8». Этот элемент мы будем называть *ведущим* (*pivot*). Ведущий элемент разбил наш массив на три подмассива: $S_l = \{6, 7, 3, 2, 4, 5, 6\}$ с размером $|S_l| = 7$, $S_v = \{8\}$ с размером $|S_v| = 1$ и $S_r = \{10, 14, 18, 13\}$ с размером $|S_r| = 4$.

Так как мы ищем порядковую статистику 6, а размер первого массива (множества, элементы которого меньше v) равен 7, то очевидно, что искомое находится в первом множестве; $k < |S_l| \to$ первый случай.

Мы уменьшили размер массива, в котором производим поиск. Проделаем вторую итерацию. $S = \{6, 7, 3, 2, 4, 5, 6\}$. Пусть случайным образом выбран элемент 5.

$$\begin{aligned} S_l &= \{3, 2, 4\} \quad |S_l| = 3 \\ S_v &= \{5\} \quad |S_v| = 1 \\ S_r &= \{6, 7, 6\} \quad |S_r| = 3 \end{aligned}$$

$k > |S_l| + |S_v| \to$ третий случай.

Третий проход: $S = \{6, 7, 6\}$

Выбран произвольный элемент 6.

$$\begin{aligned} S_l &= \{\} \quad |S_l| = 0 \\ S_v &= \{6, 6\} \quad |S_v| = 2 \\ S_r &= \{7\} \quad |S_r| = 1 \end{aligned}$$

$|S_l| < k \leqslant |S_l| + |S_v| \to$ второй случай.

Ответ: 6.

Нахождение k-статистики. Сложность

Разобранный нами алгоритм относится к классу алгоритмов *разделяй и властвуй*, следовательно, к нему можно применить основную теорему о рекурсии.

Вначале зададимся вопросом: а что же будет, если каждый раз выбор элемента даст уменьшение подзадачи в 2 раза?

Тогда

$$T(N) = T\left(\frac{N}{2}\right) + O(N)$$

Ясно, что после разбиения на подзадачи только одна из оставшихся будет нас интересовать, поэтому количество подзадач $a = 1$. В идеальном варианте каждая новая подзадача меньше задачи в 2 раза, то есть $b = 2$. Операция «перетасовки» массива, отправляющая все элементы меньше ведущего влево от него, а все элементы больше ведущего вправо от него, имеет сложность $O(N)$. Коэффициент $d = 1$.

Так как $\log_b a = \log_2 1 = 0 < 1$, то в действие вступает первая ветвь основной теоремы о рекурсии, следовательно, $T(N) = O(N)$. Возможно, результат покажется довольно неожиданным, но вспомним, что сумма геометрической прогрессии с множителем $\frac{1}{2}$ стремится к удвоенному первому элементу.

Худший случай наступает, если при выборе элемента каждый раз оказывается, что либо $|S_l| = 0$, либо $|S_r| = 0$, то есть если ведущим элементом станет наименьший или наибольший элемент массива. Тогда

$$T(N) = N + (N-1) + \cdots = \Theta(N^2).$$

К счастью, такое событие маловероятно. Нетрудно посчитать, что его вероятность равна $p = \prod\limits_{i=N..2} \frac{2}{i}$. Для $N = 10$, например, эта вероятность равна $p \approx 1.4 \times 10^{-4}$, а для $N = 20$ — $p \approx 1.1 \times 10^{-13}$.

Назовём *хорошим элементом* такой, что его порядковый номер L в отсортированном массиве удовлетворяет выражению

$$\frac{1}{4}|S| \leqslant L \leqslant \frac{3}{4}|S|.$$

Вероятность k элемента оказаться *хорошим* $p = \frac{1}{2}$, а математическое ожидание количества испытаний для выпадения *хорошего* элемента $E = 2$.

Следовательно

$$T(N) \leqslant T\left(\frac{3}{4}N\right) + O(N) \to O(N).$$

Таким образом, у нас появился алгоритм, который *в среднем* способен находить k-ю порядковую статистику за время $O(N)$, или, как говорят, за *линейное время*, но в маловероятных худших случаях его сложность может составлять $O(N^2)$. Как мы вскоре убедимся, иметь такие худшие случаи — судьба большого количества эффективных алгоритмов.

3.2.7 Быстрая сортировка

Наиболее популярный сейчас в качестве алгоритма сортировки общего назначения алгоритм быстрой сортировки почти повторяет алгоритм поиска k-й порядковой статистики. В его основе лежит то же самое разбиение массива ведущим элементом на два подмассива — и рекурсивная сортировка левой и правой частей. Хотелось бы ведущий элемент сделать как можно ближе к медианному, но такой элемент так просто не найти.

Рассмотрим алгоритм на примере. В качестве «подопытного кролика» будет выступать массив $S = \{10, 5, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8\}$.

- Разделение 1. Пусть ведущий элемент = 8.

$$S_{1l} = \{5, 7, 3, 2, 4, 5, 6, 8\}, S_{1r} = \{10, 14, 18, 13\}$$
$$S_1 = \{\underbrace{5, 7, 3, 2, 4, 5, 6, 8}, \underbrace{10, 14, 18, 13}\}$$

- Рекурсивное разделение 2. Пусть ведущий элемент равен 5, $S_1 = \{5, 7, 3, 2, 4, 5, 6, 8\}$

$$S_{2l} = \{5, 3, 2, 4, 5\}, S_{2r} = \{7, 6, 8\}$$
$$S_2 = \{\underbrace{5, 3, 2, 4, 5}, \underbrace{7, 6, 8}\}$$

- И левая, и правая части нового массива, в конце концов, оказались достаточно малы для того, чтобы избежать рекурсии. Пусть в нашем примере такая граница проходит для массивов длиной 5 или меньших. На этом этапе в качестве «обычной» сортировки популярен алгоритм сортировки обменом.

$$S_2 = \{\underbrace{2, 3, 4, 5, 6}, \underbrace{6, 7, 8}\}$$

S_{1r} после обычной сортировки станет равным $S_{1r} = \{10, 14, 14, 18\}$
Так как и левая, и правая части массива на каждом из этапов сортируются на месте (а ведущий элемент на месте уже находился), то массив оказывается отсортирован.

Быстрая сортировка: сложность

Это алгоритм напоминает алгоритм поиска k-й порядковой статистики, за тем исключением, что теперь нас интересуют обе подзадачи, и, следовательно, коэффициент порождения задач b в лучшем из случаев будет достигать двух. Следовательно, при выборе медианного элемента при следующих значениях параметров основной теоремы о рекурсии (количество подзадач $a = 2$, размер подзадачи $b = 2$, коэффициент консолидации $d = 1$) сложность будет следующей:

$$T(N) = T\left(\left\lceil \frac{N}{2} \right\rceil\right) + T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + \Theta(N).$$

$\log_b a = \log_2 2 = 1 \to T(N) = O(N \log N)$.

Если каждый раз при выборе ведущего элемента будет выбираться минимальный или максимальный элемент, то сложность алгоритма попадёт в зону, занятую примитивными алгоритмами с квадратичной сложностью. Вероятность такого события при условии случайного выбора равна

$$p = \frac{2}{N} \cdot \frac{2}{N-1} \cdot \dots \cdot \frac{2}{3} = \frac{2^{N-1}}{N!}$$

Например, при $N = 10$ эта вероятность будет составлять $p = 1.4 \times 10^{-4}$, а при $N = 20 \to p = 1.1 \times 10^{-13}$. Возможно, такой маленькой вероятностью можно пренебречь. На практике один из простых способов — выбрать медианный элемент из трёх случайных элементов массива.

Быстрая сортировка: особенности

- Все операции, как мы видим, могут производиться без требования дополнительных массивов, поэтому сортировка может проводиться на месте.
- Сложность в наихудшем случае $O(N^2)$, но вероятность такого события весьма мала. Если требуется обязательно использовать именно этот алгоритм и абсолютно неприемлема даже минимальная вероятность

квадратичного времени, стоит больше уделить внимание вопросу выбора ведущего элемента.
- В прямолинейной реализации глубина рекурсивных вызовов может достигать N, что потребует $O(N)$ для стека вызовов. Это тоже часто бывает неприемлемо.
- Сложность в среднем $O(N \log N)$.

3.2.8 Сортировка слиянием

Алгоритмы нахождения k-статистики и быстрой сортировки основаны на *операции выборки* — нахождения k-го минимального элемента в массиве. Ещё один алгоритм сортировки основан на операции, противоположной операции выборки — объединении уже отсортированных массивов. Нетрудно убедиться, что при наличии двух отсортированных массивов примерно одинакового размера их объединение в отсортированный массив можно произвести за $O(N)$, где N — длина образовавшегося массива. Объединение отсортированных массивов называют *двухпутевым слиянием*. *Декомпозиция* — разделение массива на подмассивы — весьма простая операция. По сути, она может заключаться в выборе границы между подмассивами. Имея эти операции, можно начинать сортировку массива $S = \{10, 5, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8\}$.

Первая декомпозиция (проводимая за $O(1)$) даёт нам подмассивы $S_l = \{10, 5, 14, 7, 3, 2\}$ и $S_r = \{18, 4, 5, 13, 6, 8\}$. Декомпозиция для левой половины даёт подмассивы $S_{ll} = \{10, 5, 14\}$ и $S_{lr} = \{7, 3, 2\}$. Установив порог перехода на обычную сортировку в три элемента, мы после данной декомпозиции его достигаем, и после сортировки подмассивы становятся равными $S_{ll} = \{5, 10, 14\}$ и $S_{lr} = \{2, 3, 7\}$. Их слияние даст нам полностью отсортированный массив $S_l = \{2, 3, 5, 7, 10, 14\}$. Проведя подобные операции с правой частью массива, мы получим отсортированную правую половину $S_r = \{4, 5, 6, 8, 13, 18\}$, после чего очередная операция слияния даст нам полностью отсортированный массив $S = \{2, 3, 4, 5, 6, 7, 8, 10, 13, 14, 18\}$.

Псевдокод для алгоритма достаточно прост и прямолинеен:

```
void mergeSort(int a[], int low, int high) {
  if (high - low < THRESHOLD) {
    plainSort(a, low, high);
  } else {
    int mid = (low + high) / 2;
    mergeSort(a, low, mid);
    mergeSort(a, mid+1, high);
```

```
    merge(a, low, mid, high);
  }
}
```

Здесь `THRESHOLD` — порог перехода на сортировку обычным способом. Теоретически можно было бы его не ставить, но на практике слишком мелкие операции слияния скорости не добавляют.

Сортировка слиянием: сложность

В данном алгоритме мы наблюдаем классический пример работы принципа *разделяй и властвуй* — обработка различных частей массива совершенно независима, следовательно, работает основная теорема о рекурсии.

$$T(N) = T\left(\left\lceil \frac{N}{2} \right\rceil\right) + T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + \Theta(N)$$

- Количество подзадач всегда равно двум $a = 2$.
- Размер подзадачи всегда близок к половине основной $b = 2$.
- Операция слияния имеет сложность $O(N)$, коэффициент степени при операции консолидации $d = 1$.
  Мы попадаем на ветку $\log_b a = \log_2 2 = 1$, следовательно, $T(N) = O(N \log N)$.

Сортировка слиянием: особенности

- Операция декомпозиции может быть простой, но не имеется простых алгоритмов слияния двух отсортированных массивов в один без использования дополнительной памяти. Обычно требует добавочно $O(N)$ памяти.
- Интересный и полезный факт: сложность не зависит от входных данных и всегда равна $O(N \log N)$[10].
- Как мы вскоре убедимся, этот алгоритм при небольшой модификации прекрасно подходит для *внешней* сортировки, то есть сортировки файлов.

[10] Конечно же, в этом случае сложность алгоритма более точно описывается символом Θ: $\Theta(N \log N)$, но, как мы говорили раньше, мы, скорее, практики, и будем ещё использовать $O-$нотацию вместо $\Theta-$нотации.

3.3 Нижняя оценка сложности алгоритмов

Обратили ли вы внимание на то, что самые быстрые алгоритмы, `QuickSort` и слиянием, имеют одинаковую сложность $O(N \log N)$? Возможно ли, не меняя условие задачи, — а оно заключается в том, что мы упорядочиваем два элемента на основе операции сравнения, — придумать алгоритм сортировки с меньшей сложностью? Оказывается, невозможно. Для наброска доказательства воспользуемся понятием *дерева решений*.

3.3.1 Деревья решений

В дереве решений имеются вершины, которые могут быть *терминальными*, то есть заключительными, не имеющими потомков, и нетерминальными, иначе называемыми *узлами*. Каждый узел имеет ровно двух потомков и помечен меткой вида $i : j$, где $1 \leqslant i, j \leqslant N$. В этом узле происходит сравнение двух элементов множества S, S_i и S_j. Каждая терминальная вершина содержит окончательное решение задачи в виде одной из перестановок множества $\{1, 2, \ldots, N\}$.

Тогда задачу сортировки можно определить как прохождение дерева от корневой вершины к терминальной. Сравнив два элемента в каком-нибудь из узлов, мы принимаем решение в зависимости от результата операции сравнения и идём либо по левой ветке дерева, либо по правой. Если мы точно установили, что для данной последовательности сравнений имеется перестановка, которая соответствует всем операциям, то мы попали в терминальную вершину, которая и соответствует найденной перестановке.

Необходимое условие: в терминальных вершинах должны оказаться все возможные перестановки, иначе мы можем предъявить алгоритму ту перестановку, которой не имеется во множестве терминальных вершин — и алгоритм не сможет выдать решение.

На рисунке — одно из возможных деревьев решений для решения задачи сортировки трёх элементов. Мы начинаем сравнение с пары `(1,2)` и, если элемент 1 меньше элемента 2 или равен ему, сравниваем элементы 2 и 3 иначе — элементы 1 и 3.

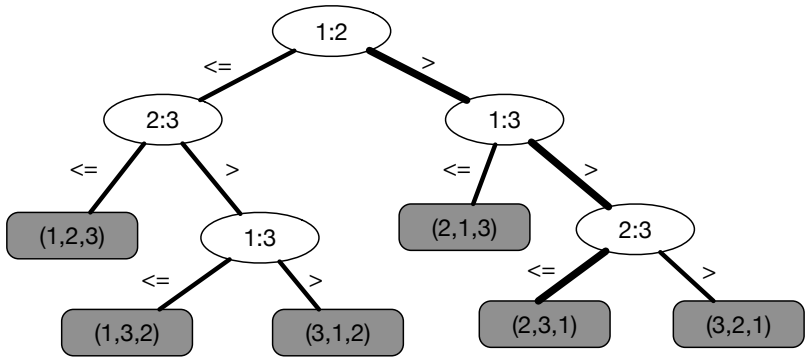


Рис. 3.46. Дерево принятия решений

Тёмным цветом выделены терминальные вершины. Очевидно, что количество терминальных вершин равно $N!$. Узел является нетерминальной вершиной и общее количество узлов равно $N!-1$. Действительно, дерево можно представить как вариант соревнований с выбыванием: если рассматривать перевёрнутый вариант дерева, то узел можно трактовать как победителя пары. В каждом состязании пар количество претендентов уменьшается на единицу, следовательно, общее количество состязаний в дереве (в нашем случае сравнений) будет равно $N! - 1$.

Сложность алгоритма сортировки определяется количеством узлов в самом длинном маршруте от корня до терминальной вершины. При одинаковой для всех терминальных вершин длине пути H дерево наибольшей вместимости есть полное двоичное дерево (об этом в следующей лекции), которое состоит из $2^H - 1$ вершин.

Минимальная глубина дерева, содержащего V узлов и вершин, не может быть меньше $H \geqslant \log_2 V$.

Логарифмируя формулу Стирлинга для вычисления факториалов:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n,$$

получаем, что $H \sim N \log N$, что и требовалось доказать. Однако мы не опускаем руки и пытаемся найти более быстрые способы сортировки, основанные на других принципах.

3.4 Сортировка со свойствами элементов

3.4.1 Сортировка подсчётом

Есть ли алгоритмы сортировки со сложностью, меньшей $O(N \log N)$?
Да, если использовать свойства ключей.
Пусть множество значений ключей дискретно и ограничено

$$D(K) = \{K_{min}, \ldots, K_{max}\}.$$

Тогда при наличии добавочной памяти количеством $|D(K)|$ ячеек сортировку можно произвести за $O(N)$.

Продемонстрируем это, для примера отсортировав массив `S={10, 5, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8}`

Пусть заранее известно, что значения массива — натуральные числа, которые не превосходят 20.

Этап 1. Заводим массив $F[1..20]$, содержащий вначале нулевые значения.

$$F = \boxed{0}$$

Каждый элемент массива F в будущем будет содержать количество использований элемента с таким значением в исходном массиве.

Этап 2. Проходим по массиву S. $S_1 = 10; F_{10} \leftarrow F_{10} + 1$.

$$F = \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$$

$S_2 = 5; F_5 \leftarrow F_5 + 1$.

$$F = \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$$

...
$S_{12} = 5; F_8 \leftarrow F_8 + 1$.

$$F = \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{2}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}$$

Этап 3. Проходим по ненулевым элементам F. Так как второй элемент массива F содержит число 1, то это число — количество элементов, равных 2, в отсортированном массиве. 5-й элемент массива F содержит число 2, следовательно, в отсортированный массив мы добавляем два таких элемента.

$$S = \{2, 3, 4, 5, 5, 6, 7, 8, 10, 13, 14, 18\}$$

Обратите внимание, что в этом алгоритме для размещения элементов отсортированного массива мы не применяли операций сравнения. Мы просто подсчитали количество необходимых элементов, используя их значение в качестве *ключа* в другом массиве. А если бы сортируемые элементы были бы вещественными числами? Строками? Структурами? Увы, в этом случае алгоритм нам бы не помог.

Сортировка подсчётом: особенности

- Ключи должны быть перечислимы и могут использоваться в качестве индекса для вспомогательного массива счётчиков.
- Пространство значений ключей должно быть ограниченным.
- Требуется дополнительная память $O(|D(K)|)$ для размещения счётчиков.
- Сложность всего алгоритма определяется, исходя из сложности его этапов: на первом этапе обнуление массива в $|D(K)|$ элементов требует $O(|D(K)|)$; на втором этапе, подсчёта, требуется $O(N)$ операций; третий этап — повторный проход по массиву счётчиков, что даёт нам $O(|D(K)|)$. Итого:

$$T(N) = O(|D(K)|) + O(N) + O(|D(K)|) = O(|D(K)|) + O(N).$$

3.4.2 Поразрядная сортировка

Похоже на то, что поразрядная сортировка, которую мы будем сейчас рассматривать — первая автоматизированная промышленная сортировка. Её история идёт от обработки результатов переписи населения конца XIX-го — начала XX-го века. Основным хранителем информации были перфокарты, на которых цифры пробивались в определённых позициях. Электромеханическое устройство могло распределять поступающие перфокарты на несколько выходных карманов в зависимости от значения пробивки в заданной позиции.

Давайте воспользуемся этим примером и попробуем отсортировать множество трёхзначных чисел без использования операций сравнения и без чрезмерных затрат памяти на массив счётчиков (этот массив для выбранных нами чисел должен был бы содержать 1000 элементов). Воспользуемся усложнённым вариантом сортировки подсчётом — *поразрядной сортировкой*.

Разобьём ключ на фрагменты — разряды — и представим его как массив фрагментов. Все ключи должны иметь одинаковое количество фрагментов.

Например: ключ 375 можно разбить на 3 фрагмента $\{3, 7, 5\}$, и ключ 5 — тоже на 3 $\{0, 0, 5\}$.

В качестве примера попробуем отсортировать массив `S={153, 266, 323, 614, 344, 993, 23}` при разбиении на 3 фрагмента.

Этап 1. `{ {1, 5, 3}, {2, 6, 6}, {3, 2, 3}, {6, 1, 4}, {3, 4, 4}, {9, 9, 3}, {0, 2, 3} }`

Рассматривая последний фрагмент как ключ, устойчиво отсортируем фрагменты методом подсчёта.

`{ {1, 5, 3}, {3, 2, 3}, {9, 9, 3}, {0, 2, 3}, {3, 4, 4}, {6, 1, 4}, {2, 6, 6} }`

Этап 2. Теперь устойчиво отсортируем по второму фрагменту. `{ {6, 1, 4}, {3, 2, 3}, {0, 2, 3}, {3, 4, 4}, {1, 5, 3}, {2, 6, 6}, {9, 9, 3} }`

Этап 3. И, наконец, по первому фрагменту. `{ {0, 2, 3}, {1, 5, 3}, {2, 6, 6}, {3, 2, 3}, {3, 4, 4}, {6, 1, 4}, {9, 9, 3} }`

При размере исходного массива N и размере вспомогательного массива 10 нам потребовалось провести всего три сортировки подсчётом, что даёт сложность в $3O(N) = O(N)$.

Возвращаясь к нашим предкам, посмотрим, что же они делали. Необходимым условием было размещение сортируемых ключей в одних и тех же позициях перфокарты (а перфокарта могла содержать от 45 до 80 позиций). Стопку сортируемых перфокарт помещали во входной карман, установив переключатель на последнюю цифру ключа. После нажатия на кнопку начиналась операция сортировки: перфокарты, содержащие в заданной позиции ноль, попадали в один приёмный карман, цифру один — в другой и так далее. Затем работник, не меняя порядка внутри перфокарт в кармане (устойчивая сортировка), собирал перфокарты в нужном порядке, получая тем самым частично отсортированную по ключу стопку. Затем переключатель устанавливался на предпоследнюю цифру ключа — и далее по тому же принципу. После последнего прохода собранная стопка содержала полностью отсортированную последовательность.

Поразрядная сортировка: особенности

- Требует ключи, которые можно трактовать как множество перечислимых фрагментов.
- Требует дополнительной памяти $O(|D(K_i)|)$ на сортировку фрагментов.
- Сложность постоянна и равна $O(N \cdot |D(K_i)|)$.

Для практической реализации можно сделать замечание: операция разделения числа на фрагменты в виде десятичных цифр — на современных компьютерах чрезвычайно медленная, так как требует операций целочисленного деления и нахождения остатков. Гораздо быстрее использовать разделение на фрагменты по набору битов или байтов. В этом случае для выделения фрагментов будет достаточно очень быстрых побитовых операций.

3.5 Внешняя сортировка

3.5.1 Сортировка больших данных

Сортировать и упорядочивать данные по какому-либо критерию — задача очень распространённая. Для данных, к которым идёт много запросов от различных клиентов, придумали удобный механизм — базы данных. У баз данных свои способы хранения, свои языки запросов, и для наших задач их функциональность пока избыточна. Здесь мы ограничимся существенно более простой задачей — сортировкой большого количества данных. Для таких данных можно выделить две основных проблемы:

- Для сортируемых данных недостаточно быстрой оперативной памяти, и приходится пользоваться существенно более медленной внешней — `HDD` и `SSD`.
- Время сортировки превосходит приемлемые границы. Отсортировать миллиарды записей даже при очень хорошо реализованных алгоритмах сортировки — тяжёлая задача, и мы её здесь решать не будем. Сосредоточимся на первой — сортировке при недостатке оперативной памяти. Такая сортировка носит название *внешней*.

3.5.2 Внешняя сортировка слиянием

В ситуации, когда помещение всех данных для обработки в быструю оперативную память невозможно, приходится использовать внешнюю память. Для сортировки нам достаточно использовать абстракцию «лента», предоставляющую следующие методы:

- `create` — создать новую пустую ленту и открыть её с возможностью записи.
- `open` — открыть существующую ленту исключительно для чтения.
- `close` — перестать использовать ленту.
- `getdata` — прочитать информацию с ленты.

• `putdata` — записать информацию на ленту.

Мы уже знаем, что в алгоритме сортировки слиянием в оперативной памяти на фазе слияния каждый из входных массивов последовательно просматривается от начала до конца, и выходной массив пишется строго последовательно. Это наталкивает на мысль, что подобный алгоритм мог бы быть полезным и для операций работы с лентами. Рассмотрим операцию *слияние* двух лент, *двухпутевое слияние*. Входными данными двухпутевого слияния являются две отсортированные ленты, выходными — другая отсортированная лента. Введём ещё один термин — *чанк* (*chunk*) — фрагмент данных, помещающихся в оперативной памяти.

Попробуем решить задачу прямолинейно, используя всю доступную оперативную память для обработки за раз максимально возможных фрагментов.

Пусть исходная лента содержит 8 чанков (рис. 3.47).

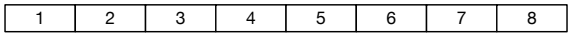


Рис. 3.47. Внешняя сортировка: исходные данные

На первом этапе считывается первый чанк, сортируется внутренней сортировкой и отправляется на первую временную ленту (рис. 3.48).

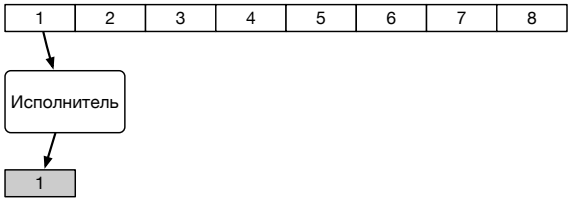


Рис. 3.48. Внешняя сортировка: обработан первый чанк

На втором этапе второй чанк сортируется и отправляется на вторую временную ленту (рис. 3.49).

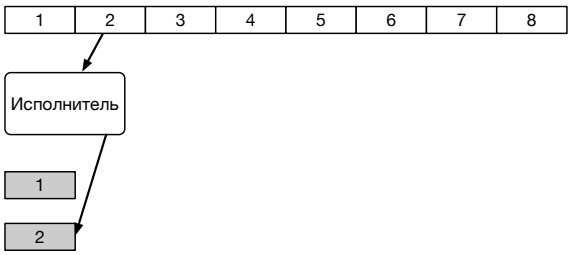


Рис. 3.49. Внешняя сортировка: обработано два чанка

Третий этап — слияние. Сливаются первая и вторая временные ленты (рис. 3.50).

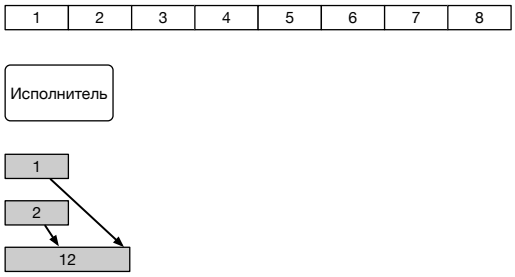


Рис. 3.50. Внешняя сортировка: обработано и слито два чанка

Аналогично считываются, сортируются и выводятся на временные ленты чанки 3 и 4 (рис. 3.51).

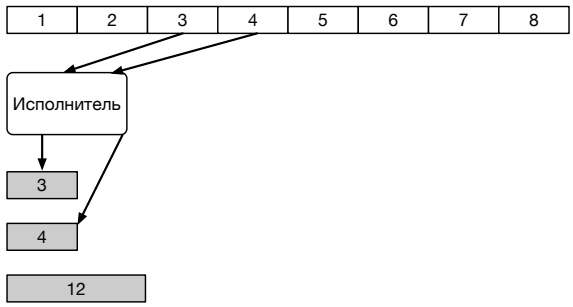


Рис. 3.51. Внешняя сортировка: отсортировано и выведено ещё два чанка

Аналогично временные ленты сливаются в ещё одну, четвёртую. Увы, здесь мы не можем использовать меньшее количество лент (рис. 3.52).

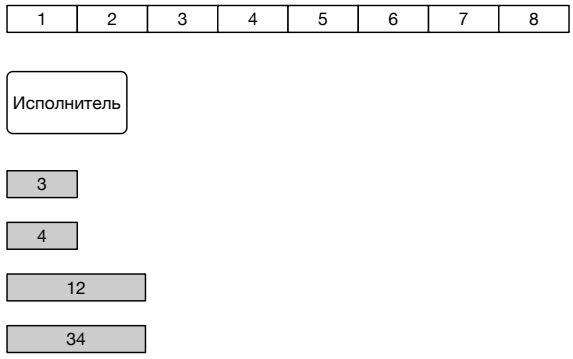
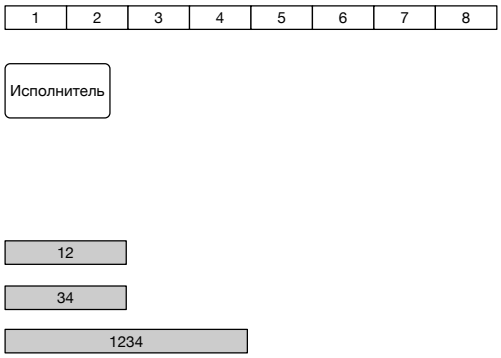


Рис. 3.52. Внешняя сортировка: используется уже четыре ленты

Сливаем ленты, содержащие чанки 12 и 34, получаем ленту 1234.



Для получения ленты 5678 требуется 4 временные ленты. Плюс лента 1234. Итого — 5 лент.

Нельзя сказать, что придуманный нами алгоритм оказался очень простым. При его реализации приходится тщательно отслеживать состояние каждой ленты и фиксировать, что на ней находится. Тем не менее, оценим его сложность.

- Внутренних сортировок в этом алгоритме столько же, сколько было чанков в исходных данных, K.
- Общая сложность внутренней сортировки

$$O\left(\frac{N}{K} \times \log \frac{N}{K}\right) = O(N \log N).$$

- Сложность каждой из операций слияния — $O(N)$.
- Количество операций слияния $O(\log K)$.
- Общая сложность $O(N \log N)$.
- Сложность по количеству временных лент $O(\log K)$.

Ничего нового. Сложность алгоритма с использованием внешней памяти не увеличилась — и это обнадёживает. А можно ли упростить алгоритм до такой степени, чтобы он не использовал большое количество лент?

Попытаемся воспользоваться наблюдением, что при операции слияния дополнительной памяти не требуется, достаточно памяти для двух элементов. Попробуем произвести внешнюю сортировку без использования большого буфера и отказаться от использования *чанков*. Вместо этого мы будем использовать *серии* — неубывающие последовательности на ленте. Как оказывается, использование сортировки сериями позволит нам обойтись всего двумя дополнительными лентами, не ухудшив сложность алгоритма.

3.5.3 Сортировка сериями

Пусть имеется лента

$$\underbrace{14, 4, 2, 7, 5, 9, 6, 11, 3, 1, 8, 10, 12, 13}.$$

Заводим две вспомогательные ленты, в каждую из которых помещаем очередную серию длины 1 из входной ленты.

$$\begin{cases} \underbrace{14}, \underbrace{2}, \underbrace{5}, \underbrace{6}, \underbrace{3}, \underbrace{8}, \underbrace{12} \\ \underbrace{4}, \underbrace{7}, \underbrace{9}, \underbrace{11}, \underbrace{1}, \underbrace{10}, \underbrace{13} \end{cases}$$

Формируем пары из первого элемента первой серии вместе с первым элементом из второй серии, второго элемента первой серии и второго элемента второй серии и так далее. Каждую из пар упорядочиваем — и результат попарно сливаем в исходный файл. Эту операцию можно совершить, имея в оперативной памяти ровно два элемента — по одному элементу из каждой последовательности.

$$\begin{cases} \underbrace{14}, \underbrace{2}, \underbrace{5}, \underbrace{6}, \underbrace{3}, \underbrace{8}, \underbrace{12} \\ \underbrace{4}, \underbrace{7}, \underbrace{9}, \underbrace{11}, \underbrace{1}, \underbrace{10}, \underbrace{13} \end{cases}$$

Инвариант операции слияния серий: совокупная последовательность является серией удвоенной длины.

$$\underbrace{4, 14}, \underbrace{2, 7}, \underbrace{5, 9}, \underbrace{6, 11}, \underbrace{1, 3}, \underbrace{8, 10}, \underbrace{12, 13}.$$

Серии длины 2 попеременно помещаем на выходные ленты.

$$\underbrace{4, 14}, \underbrace{2, 7}, \underbrace{5, 9}, \underbrace{6, 11}, \underbrace{1, 3}, \underbrace{8, 10}, \underbrace{12, 13}.$$

$$\begin{cases} \underbrace{4, 14}, \underbrace{5, 9}, \underbrace{1, 3}, \underbrace{12, 13} \\ \underbrace{2, 7}, \underbrace{6, 11}, \underbrace{8, 10}. \end{cases}$$

Снова каждую из серий попарно сливаем в исходную ленту.

$$\begin{cases} \underbrace{4, 14}, \underbrace{5, 9}, \underbrace{1, 3}, \underbrace{12, 13} \\ \underbrace{2, 7}, \underbrace{6, 11}, \underbrace{8, 10}. \end{cases}$$

Длина полных серий в выходной ленте — не меньше 4. Только последняя серия может иметь меньшую длину.

$$\underbrace{2,4,7,14},\underbrace{5,6,9,11},\underbrace{1,3,8,10},\underbrace{12,13}$$

.

Третий этап: формируем временные ленты сериями по 4. И здесь, и в других подобных случаях только последняя серия на ленте может иметь длину меньше 4.

$$\underbrace{2,4,7,14},\underbrace{5,6,9,11},\underbrace{1,3,8,10},\underbrace{12,13}$$

$$\begin{cases}\underbrace{2,4,7,14}, & \underbrace{1,3,8,10}\\ \underbrace{5,6,9,11}, & \underbrace{12,13}\end{cases}$$

Сливаем серии длины 4.

$$\begin{cases}\underbrace{2,4,7,14}, & \underbrace{1,3,8,10}\\ \underbrace{5,6,9,11}, & \underbrace{12,13}\end{cases}$$

Слияние серий длины 4 обеспечивает длину серий 8.

$$\underbrace{2,4,5,6,7,9,11,14},\underbrace{1,3,8,10,12,13}$$

Последний этап: разбивка на серии длины 8 с последующим слиянием.

$$\underbrace{2,4,5,6,7,9,11,14},\underbrace{1,3,8,10,12,13}$$

.

Разбивка:

$$\begin{cases}\underbrace{2,4,5,6,7,9,11,14}\\ \underbrace{1,3,8,10,12,13}\end{cases}$$

Слияние:

$$\underbrace{1,2,3,4,5,6,7,8,9,10,11,12,13,14}$$

Алгоритм, как мы видим, оказался достаточно изящным и не требовательным к количеству лент. Более того, для его работы вполне достаточно памяти всего для двух элементов!

Сортировка сериями: оценка сложности алгоритма

- За один проход примем операцию разбивки с последующим слиянием.
- На каждом проходе участвуют все элементы лент по два раза.
- Инвариант: длина серии после прохода k равна 2^k.
- Алгоритм завершается, когда длина серии оказывается не меньше N.
- Итого требуется $\lceil \log_2 N \rceil$ проходов по файлам.
- Сложность алгоритма: $O(N \log N)$.
- Сложность по памяти: $O(1)$.
- Сложность по ресурсам: две временные ленты.

3.5.4 Сортировка сериями: возможные улучшения

Несмотря на изящество алгоритма, его анализ показывает, что длина серии начинается от 1 и для полной сортировки требуется ровно $\lceil \log_2 N \rceil$ итераций, но первые итерации при этом производятся с небольшой длиной серии.

А ведь мы можем сократить число итераций, использовав больше, чем 1 элемент памяти. Вспомним, что, хотя нас и ограничили по памяти, какой-то её объём нам доступен.

Сортировка сериями: улучшенный вариант

- Подбираем такое число k_0, при котором серия длиной 2^{k_0} помещается в доступную память.
- Разбиваем исходную ленту на серии: считывается первый чанк длиной 2^{k_0}, сортируется внутренней сортировкой, пишется на первую ленту. Второй чанк после сортировки пишется на вторую ленту — и это повторяется до истощения входных данных.
- После подготовки возвращаемся к алгоритму сортировки сериями, начиная с серии длиной 2^{k_0}.

Мы заменили k_0 проходов алгоритма одним. Следовательно, количество итераций сокращается на $k_0 - 1$, при этом сложность алгоритма не меняется, оставшись $O(N \log N)$. Меняется лишь коэффициент амортизации. Тем не менее, для практического применения улучшенный вариант может дать значительную выгоду.

Пусть $N = 10^8$. Тогда $\log_2 N \approx 27$. Для $k = 20$ в память помещается 2^{20} элементов, что вполне реально. Тогда общее количество итераций составит $27 - 20 + 1 = 8$ вместо 28. Действительно, выгодно.

3.6 Сортировка и параллельные вычисления

Современные компьютеры содержат по несколько вычислительных ядер — исполнителей машинного кода, и их хотелось бы использовать для ускорения процесса сортировки. Как это сделать? Требуется учесть нюансы, которые возникают при параллельном исполнении.

3.6.1 Особенности параллельного исполнения

- Каждый из исполнителей (*вычислительный поток*, *thread*) может исполнять свой поток инструкций.
- В программном коде это выглядит как одновременное исполнение нескольких функций (процедур).
- Все исполнители могут иметь совместный доступ к общим данным.
- Совместный доступ к общим переменным — и благо, и зло одновременно. Благо — так как это удобный способ взаимодействия, обмен данными. Зло — так как может привести к конфликтам.

Рассмотрим одновременное исполнение[11] двух функций, использующих совместные переменные.

```
int a = 2, b = 10;

void thread1() {
  a += b; // 1a
  b = 5;  // 1b
}

void thread2() {
  b = 13; // 2b
  a *= b; // 2a
}
```

Можно ли с определённостью ответить на вопрос, чему равны переменные a и b после окончания обоих исполнителей? Увы, порядок исполнения недетерминирован — и при различных прогонах алгоритма могут получаться разные результаты. Дело в том, что при многопоточном исполнении и

[11] Мы вступаем здесь на территорию, на которой принята немного другая терминология. В частности, понятия *одновременность* в многопоточном программировании нет, одновременное исполнение принято здесь называть *соисполнением*, а совместные переменные называются *общей памятью*.

одновременном использовании одних и тех же переменных задача становится комбинаторной: результат зависит от взаимного порядка исполнения, а таких порядков исполнения может быть несколько.

Если обозначить за $t(x)$ абсолютное время окончания исполнения соответствующих инструкций, то заведомо известно лишь то, что $t(1a) < t(1b)$ и $t(2b) < t(2a)$.

Вот некоторые из возможных путей исполнения:

$1a \to 1b \to 2a \to 2b$

$1a \to 2a \to 1b \to 2b$

$1a \to 2a \to 2b \to 1b$

...

Более того, операции $1a$ и $2a$ неделимы (*атомарны*) для исполнителя «Язык Си», но они отнюдь не атомарны для исполнителя «современный процессор»!

Инструкция `a += b` превратится в несколько операций:

1. Загрузка значения переменной b в регистр процессора.
2. Загрузка значения переменной a в регистр процессора.
3. Добавление значения b регистру, содержащему значение a.
4. Сохранение получившегося значения из регистра в переменную a.

Пока на одном вычислительном ядре процессора исполняется любая из этих операций, операционная система может передать управление другому исполнителю.

Для борьбы с такими проблемами автор алгоритма должен предусматривать использование *примитивов синхронизации* — средств, препятствующих одновременному изменению одних и тех же переменных. К сожалению, на исполнение *примитивов синхронизации* требуется значительное время, за которое можно исполнить сотни и тысячи обычных операций.

Перед нами не стоит задачи разработки оптимальных алгоритмов, максимально использующих все вычислительные ядра процессора, нам нужно понять, какие свойства алгоритмов позволят нам избежать проблем при их проектировании. Простейший способ — ограничить использование общих данных *критическими секциями*. Чем меньше взаимодействуют вычислительные потоки между собой — тем лучше. Это означает, что основные операции лучше всего производить над *локальными* для каждого исполнителя данными и только в отдельные моменты использовать *точки синхронизации* для обмена.

Разработка параллельных версий классических алгоритмов — отдельная, очень сложная задача. Алгоритмы, которые наиболее эффективны

в варианте для одного исполнителя, часто непригодны для варианта с несколькими исполнителями.

Вернёмся к параллельной сортировке. Она имеет много общих свойств с внешней:

1. Данные разбиваются на непересекающиеся подмножества.
2. Каждое подмножество обрабатывается независимо.
3. После независимой обработки используется слияние.

Для параллельного исполнения первого и третьего этапов требуются более изощрённые алгоритмы, чем мы использовали. А вот второй этап действительно можно проводить независимо. Так как этот этап составляет наибольшее время во всём исполнении алгоритма, то уменьшение времени его работы даст наибольший вклад в сложность всего алгоритма. Впрочем, как это конкретно реализовать — уже тема для другой дисциплины. Мы просто хотели обратить внимание на то, что в современном мире практически не осталось одноядерных процессоров, способных исполнять не более одного потока одновременно, и что при разработке и анализе алгоритма стоит обращать внимание на возможность его распараллеливания на несколько вычислительных ядер.

3.7 Сравнительный анализ методов сортировки

В конце лекции, как и было обещано, дадим сводную таблицу по рассмотренным методам сортировки. Сортировку `Heap` мы рассмотрим уже в следующей лекции, а популярную ныне сортировку `TimSort` — на семинарах.

Алгоритм	Лучший случай	Средний случай	Худший случай	Память	Устойчива?
Пузырьком	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Шелла	$O(N^{\frac{7}{6}})$	$O(N^{\frac{7}{6}})$	$O(N^{\frac{4}{3}})$	$O(1)$	Нет
Вставками	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Выбором	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Быстрая	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(1)$	Да/Нет
Слиянием	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Да
TimSort	$O(N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Да
Heap	$O(N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$	Нет
Подсчётом	$O(N)$	$O(N)$	$O(N)$	$O(N)$	Да
Поразрядная	$O(N)$	$O(N)$	$O(N)$	$O(N)$	Да/Нет

3.8 Домашние задания

Задача 11. Максимальная тройка

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1.5 секунд
Ограничение по памяти:	8 мегабайт

Имеется не более 1000000 целых чисел, каждое из которых лежит в диапазоне от -1000000 до 1000000. Найти максимально возможное значение произведений любых трёх, различных по номерам, элементов массива.

Формат входных данных

```
N
A1
A2
...
AN
```

Формат выходных данных

`MaxPossibleProduct`

Пример

стандартный ввод	стандартный вывод
10 -1 2 3 -4 -2 5 -1 5 -3 -2	75

Задача 12. Сортировка по многим полям

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

В базе данных хранится N записей вида $(Name, a_1, a_2, \ldots, a_k)$ – во всех записях одинаковое число параметров. На вход задачи подаётся приоритет полей – перестановка на числах $1, \ldots, k$. Записи нужно вывести по невозрастанию в соответствии с этим приоритетом. В случае, если приоритет полей 3 4 2 1, это следует воспринимать так: приоритет значений из 3 колонки — самый высокий, приоритет значений из колонки 4 — ниже, приоритет значений из колонки 2 — ещё ниже, а приоритет значений из колонки 1 самый низкий.

Формат входных данных

$N \leqslant 1000$
$k : 1 \leqslant k \leqslant 10$
$p_1\ p_2\ \ldots\ p_k$ – перестановка на k числах, разделитель — пробел
N строк вида
Name $a_1\ a_2\ \ldots\ a_k$, разделитель — пробел

Формат выходных данных

N строк с именами в порядке, согласно приоритету

Пример

стандартный ввод	стандартный вывод
3 3 2 1 3 A 1 2 3 B 3 2 1 C 3 1 2	B A C

Замечание

Так как колонка под номером 2 — самая приоритетная, переставить записи можно только двумя способами: (A, B, C) и (B, A, C). Следующий по приоритетности столбец — первый, и он позволяет выбрать из возможных перестановок только (B, A, C). Так как осталась ровно одна перестановка, третий приоритет не имеет значения.

Задача 13. Оболочка

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Имеется массив из N целочисленных точек на плоскости.

Требуется найти периметр наименьшего охватывающего многоугольника, содержащего все точки.

Формат входных данных

```
N
x1 y1
x2 y2
...
xn yn
```

$5 \leqslant N \leqslant 500000$

$-10000 \leqslant x_i, y_i \leqslant 10000$

Формат выходных данных

Одно вещественное число — периметр требуемого многоугольника с двумя знаками после запятой.

Примеры

стандартный ввод	стандартный вывод
5 2 1 2 2 2 3 3 2 1 2	5.66

Задача 14. Очень быстрая сортировка

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1.5 секунд
Ограничение по памяти:	512 мегабайт

Имеется рекуррентная последовательность $A_1, A_2, \ldots, A_N$, строящаяся

по следующему правилу:

$A_1 = K$

$A_{i+1} = (A_i * M)\%(2^{32} - 1)\%L$.

Требуется найти сумму всех нечётных по порядку элементов в отсортированной по неубыванию последовательности по модулю L.

Для входных данных

5 7 13 100

последовательность будет такой:

$\{7; 7 * 13\%100 = 91; 91 * 13\%100 = 83; 83 * 13\%100 = 79; 79 * 13\%100 = 27\}$, то есть, $\{10; 91; 83; 79; 27\}$.

Отсортированная последовательность $\{7; 27; 79; 83; 91\}$.

Сумма элементов на нечётных местах $= (7 + 79 + 91)\%100 = 77$.

Формат входных данных

$N\ K\ M\ L$

$5000000 \leqslant N \leqslant 60000000,\ 0 \leqslant K, L, M \leqslant 2^{32} - 1$

Формат выходных данных

$RESULT$

Примеры

стандартный ввод	стандартный вывод
5 7 13 100	77

Замечание

Для представления элементов последовательности необходимо использовать тип данных `unsigned int`.

Для получения массива используйте цикл

```
a[0] = K;
for (int i = 0; i < N-1; i++)
    a[i+1] = (unsigned int)((a[i]*(unsigned long long)M)&0xFFFFFFFFU)%L;
```

Внимание! Изменение типа данных и/или метода генерации элементов массива может привести (и на различных компиляторах приводит) к другой последовательности!

Задача 15. Внешняя сортировка.

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 секунды
Ограничение по памяти:	2 мегабайта

В файле `"input.txt"` содержатся строки символов, длина каждой строки не превышает 10000 байт. Файл нужно отсортировать в лексикографическом порядке и вывести результат в файл `"output.txt"`. Вот беда, файл занимает много мегабайт, а в вашем распоряжении оказывается вычислительная система с очень маленькой оперативной памятью. Но файл должен быть отсортирован!

Пример

input.txt	output.txt
aksjdfhaskjdh alsdajsldjaslkdjaslkdj asldkjaslkdjlkjq qweqweqweqweqweqwe pqiwepoiqwpoeiqwpeoiqwpeo qpeoiqwpoeiqwpe	aksjdfhaskjdh alsdajsldjaslkdjaslkdj asldkjaslkdjlkjq pqiwepoiqwpoeiqwpeoiqwpeo qpeoiqwpoeiqwpe qweqweqweqweqweqwe

Лекция 4

4.1 Задача поиска. Абстракция поиска

Информация нужна, чтобы ей пользоваться. Мы проводим дорогостоящую операцию сортировки именно для того, чтобы пользоваться информацией стало удобнее. В конце концов, сортировка сама по себе обычно никого не интересует — интересны её результаты. Представим себе словарь с переводом иностранных слов, в котором нет порядка. Поиск в таком словаре занимал бы чудовищное время. Упорядоченный по алфавиту словарь — другое дело.

Сформулируем расширенную задачу поиска.

1. Этап первый. Сбор информации. Её накопление. В применении к словарю — лингвисты формируют карточки, на которых для каждого слова имеется его значение.
2. Этап второй. Организация информации (переупорядочивание, сортировка). Уже готовые карточки сортируются в алфавитном порядке.
3. Этап третий. Извлечение информации (собственно поиск). В упорядоченном словаре поиск слова происходит намного быстрее, чем в неупорядоченном.

При построении словаря новые карточки могут сразу вноситься таким образом, чтобы не нарушать упорядоченность, то есть второй и третий этапы могут смешиваться.

Более обобщённая задача: абстракция *хранилище*.

- Задача: построение эффективного хранилища данных.
- Требования:
  - Поддержка больших объёмов информации.
  - Возможность быстро находить данные.
  - Возможность быстро модифицировать, в том числе и удалять

данные.

- Методы абстракции:
  - `Create` — добавление новой записи к хранилищу.
  - `Read` — поиск данных по ключу.
  - `Update` — обновление данных по ключу.
  - `Delete` — удаление данных.

По первым буквам методов данной абстракции её часто так и называют: `CRUD`.

Итак, первая подзадача, которую требуется решить для реализации хранилища — `Read`. Формализуем её частный случай, который назовём `find`.

Пусть имеется множество ключей $a_1, a_2, \ldots, a_n$. Требуется определить индекс ключа, совпадающего с заданным значением key.

```
bunch a;
index = a.find(key);
```

Здесь `bunch` — некое абстрактное хранилище элементов, содержащих ключи. Его организацию мы будем уточнять далее.

> Хорошая организация хранилища входит в расширенную задачу поиска.

4.2 Последовательный поиск

Самая простая ситуация: к поиску мы не готовились — и ключи не упорядочены, они находятся в простом массиве.

Индекс	0	1	2	3	4	5	6	7	8	9
Ключ	132	612	232	890	161	222	123	861	120	330
Данные	АВ	СА	ЯФ	АВ	АА	НД	ОР	ОС	ЗЛ	УГ

Операция `find(a, 222)` возвратит индекс, равный 5, а вот, например, элемента 999 в хранилище нет, поэтому операция поиска должна возвратить нечто, которое заведомо не может служить индексом при доступе к данному массиву. Например, таким индексом может быть число -1 или номер элемента за границей массива. Остановимся на последнем варианте.

`find(a, 999) = 10` (элемент за границей поиска).

Похоже, что программа на `Си/C++`, реализующая последовательный поиск — одна из простейших во всём курсе.

```
int dummysearch(int a[], int N, int key) {
  for (int i = 0; i < N; i++) {
    if (a[i] == key) {
      return i;
    }
  }
  return N;
}
```

При случайных значениях ключей в массиве и при поиске присутствующего в нём ключа вероятность найти ключ в i-м элементе $P_i = \frac{1}{N}$.

Математическое ожидание числа поисков в этом случае $E = \frac{N}{2}$. Число операций сравнения $2N$ в худшем случае $T(N) = O(N)$.

Обратили ли вы внимание на то, что при использовании исполнителя «язык C++» на каждой итерации цикла происходит *две* операции сравнения? Сравнивается ключ, который мы ищем, с очередным элементом массива — и, кроме того, номер элемента массива с количеством элементов. Язык C++ достаточно близок к машинному, и это значит, что эти операции сравнения будут действительно присутствовать в машинном коде.

Проделаем небольшую подготовку: положим тот элемент, который мы ищем, в элемент, находящийся *за* последним элементом массива (конечно, там должно быть предусмотрено для него место).

Индекс	0	1	2	3	4	5	6	7	8	9	***10***
Ключ	132	612	232	890	161	222	123	861	120	330	***999***
Данные	АВ	СА	ЯФ	АВ	АА	НД	ОР	ОС	ЗЛ	УГ	??

Результаты поисков не изменились. `find(a, 222)` по-прежнему возвращает 5, а `find(a, 999)` — 10.

```
int cleversearch(int a[], int N, int key) {
  a[n] = key;
  int i;
  for (i = 0; a[i] != key; i++)
    ;
  return i;
}
```

Убрана одна операция сравнения индекса с его границей, теперь, несмотря на то, что всё ещё $T(N) = O(N)$, скорость алгоритма практически удвоилась [12].

Хорошая подготовка данных может привести к лучшим результатам.

Сложность операций для неупорядоченного массива

- `Create` — $O(1)$. Мы просто добавляем ключ к концу массива, расширяя его.
- `Read` — $O(N)$.
- `Update` — $O(N)$. Элемент нужно сначала найти.
- `Delete` — $O(N)$. После поиска удаляемого элемента массив требуется сжать.

4.3 Поиск с сужением зоны

К сожалению, неупорядоченные данные обрабатывать достаточно трудно. Если в зоне поиска имеется упорядочивание — всё становится значительно лучше.

Сортировка даёт нам возможность упорядочить по какому-либо отношению. Тогда задачу поиска можно переформулировать следующим образом:

- Имеется множество ключей

$$a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$$

- Требуется определить индекс ключа, совпадающего с заданным значением key, или указать, что такого значения в множестве не существует.

Если мы мысленно разобьём множество на два примерно равных подмножества, то принцип «разделяй и властвуй» здесь будет работать идеально.

1. Искомый элемент равен центральному? Да — нашли.
2. Искомый элемент меньше центрального? Да — рекурсивный поиск в левой половине.
3. Искомый элемент больше центрального? Да — рекурсивный поиск в правой половине.

[12] Автору приходилось использовать этот приём на последнем этапе сложного поиска по графам, что давало прирост производительности на всём алгоритме на 20-30%.

- Вход алгоритма: упорядоченный по возрастанию массив, левая граница поиска, правая граница поиска.
- Выход алгоритма: номер найденного элемента или -1.

Формализуя алгоритм, можно получить следующее решение:

```
int binarySearch(int val, int a[], int left, int right) {
  if (left >= right) return a[left] == val? left : -1;
  int mid = (left+right)/2;
  if (a[mid] == val) return mid;
  if (a[mid] < val) {
    return binarySearch(val, a, left, mid-1);
  }  else {
    return binarySearch(val, a, mid+1, right);
  }
}
```

Глубину рекурсии оценить легко: каждый раз интервал поиска уменьшается примерно в два раза, поэтому количество рекурсивных вызовов здесь не превзойдёт $O(\log N)$, где N — размер множества.

Вот как происходит, например, поиск ключа 313:

$$\{\overbrace{\underbrace{1,4,23,45,67,}_{\text{левая часть}}}^{\text{левая часть}}\ \overbrace{68}^{\text{центр}}\ ,\overbrace{89,105,144,279,313,388}^{\text{правая часть}}\}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{12\text{ элементов}}$$

Так как $313 > 68$, то ключ — справа.

$$\{\overbrace{\underbrace{89,105}_{\text{левая часть}}}^{\text{левая часть}}\ ,\overbrace{144}^{\text{центр}}\ ,\overbrace{279,313,388}^{\text{правая часть}}\}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{6\text{ элементов}}$$

$313 > 144 \rightarrow$ ключ справа

$$\{\overbrace{\underbrace{279}_{\text{левая часть}}}^{\text{левая часть}}\ ,\overbrace{313}^{\text{центр}}\ ,\overbrace{\underbrace{388}_{\text{правая часть}}}^{\text{правая часть}}\}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{3\text{ элемента}}$$

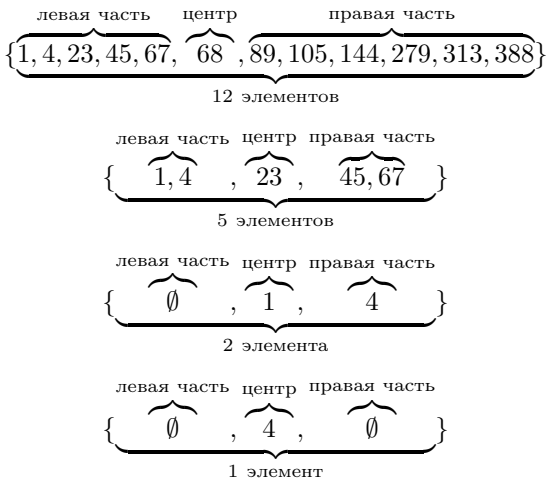
$313 = 313 \rightarrow$ ключ найден.

Сложность алгоритма здесь достаточно очевидна, но для порядка попрактикуемся в основной теореме о рекурсии.

- Количество подзадач $a = 1$.
- Каждая подзадача уменьшается в $b = 2$ раза.
- Сложность консолидации $O(1) = O(N^0) \rightarrow d = 0$.

Так как $d = \log_b a$, то $T(N) = \log N$.

Поиск отсутствующего в множестве элемента, например, 3, даёт максимальную глубину рекурсии:



Переход от рекурсии к итерации

Использование рекурсии, конечно, изящно, но итеративные алгоритмы обычно практичнее.

```
int binarySearch(int val, int a[], int left, int right) {
  while (left < right) {
    int mid = (left + right)/2;
    if (a[mid] == val) return mid;
    if (a[mid] < val) {
      right = mid - 1;
    }  else {
      left =  mid + 1;
    }
  }
  return a[left] == val? left : -1;
}
```

В худшем случае на каждой итерации в данном варианте алгоритма нужно провести три операции сравнения. Как мы уже говорили, операция

сравнения — не самая быстрая из всех операций на современных компьютерах, и неплохо бы уменьшить количество таких операций.

Мы можем попытаться уменьшить количество разбиений множества, разбивая его на большее количество интервалов. Почему бы не разбивать, например, на три интервала?

```
int ternarySearch(int val, int a[], int left, int right) {
  if (left >= right) return a[left] == val? left : -1;
  int mid1 = (left*2+right)/3;
  int mid2 = (left+right*2)/3;
  if (val < a[mid1]) {
    return ternarySearch(val, a, left, mid1-1);
  } else if (val == a[mid1]) {
    return mid1;
  } else if (a < a[mid2]) {
    return ternarySearch(val, a, mid1+1, mid2-1);
  } else if (a == a[mid2]) {
    return mid2;
  } else {
    return ternarySearch(val, a, mid2+1, right);
  }
}
```

Добились ли мы выигрыша?

По числу рекурсивных вызовов — выигрыш в $\dfrac{\log 3}{\log 2} = \log_2 3 \approx 1.58$ раз.

Количество сравнений увеличилось с 3 до 5, проигрыш в ≈ 1.67 раз.

Имеется много вариантов поиска с сужением интервала, каждый из которых может иметь свои плюсы для решения конкретной задачи. Однако математику не обмануть — и асимптотическая сложность алгоритма остаётся логарифмической.

Сложность операций для упорядоченного массива

- `Create` — $O(N)$. Для добавления элемента требуется расширить массив.
- `Read` — $O(\log N)$.
- `Update` — $O(N)$. Элемент нужно сначала найти за $O(\log N)$, затем передвинуть его налево или направо, в зависимости от величины изменения. Увы, но эта операция занимает $O(N)$.

- `Delete` — $O(N)$. После поиска удаляемого элемента массив требуется сжать.

4.4 Распределяющий поиск

Поиск с использованием свойств ключа.

Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?

Без вспомогательных данных — нет.

Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве? Вариант ответа: если $M > \log N$, то предварительной сортировкой можно добиться того, что сложность составит $O(N \log N) + M \cdot O(\log N) = O(N \log N)$.

А быстрее можно?

В некоторых случаях — да.

Если $|D(Key)|$ невелико, то имеется способ, похожий на сортировку подсчётом.

Создаётся *инвертированный массив*.

$a = \{2, 7, 5, 3, 8, 6, 3, 9, 12\}$. $|D(a)| = 12 - 2 + 1 = 11$.

$a_{inv}[2..12] = \{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1\}$

$a[0] = 2 \to a_{inv}[2] = 0$

$a[1] = 7 \to a_{inv}[7] = 1$

$a[2] = 5 \to a_{inv}[5] = 2$

$a_{inv}[2..12] = \{0, 6, -1, -1, 5, -1, 4, 7, -1, -1, 8\}$

Из следующих двух таблиц видно, откуда взялось название «инвертированный».

index	0	1	2	3	4	5	6	7	8
key	2	7	5	3	8	6	3	9	12

key	2	7	5	3	8	6	3	9	12
index	0	1	2	3	4	5	6	7	8

Алгоритм состоит из двух этапов — подготовки данных и собственно поиска. Подготовка данных состоит в создании инвертированного массива. Если диапазон значений слишком велик, то подготовка завершается неуспехом — и мы будем вынуждены искать другие пути.

```
int * prepare(int a[], int N, int *min, int *max) {
  *min = *max = a[0];
  for (int i = 1; i < N; i++) {
```

```
    if (a[i] > *max) *max = a[i];
    if (a[i] < *min) *min = a[i];
  }
  if (*max - *min > THRESHOLD) return nullptr;
  int *ret = new int[*max - *min + 1];
  for (int i = *min; i <= *max; i++) {
    ret[i] = -1;
  }
  for (int i = 0; i < N; i++) {
    ret[a[i] - *min] = i;
  }
  return ret;
}
```

Второй этап: поиск.

```
// Подготовка
int min, max;
int *ainv = prepare(a, N, &min, &max);
if (ainv != nullptr) {
    // Поиск ключа key
    result = -1;
    if (key >= min && key <= max) result = ainv[key - min];
    ...
    delete [] ainv;
} else {
    // Какой-то другой поиск.
}
```

Сложность алгоритма:

- $O(N)$ на подготовку.
- $O(M)$ на поиск M элементов.
- $T(N, M) = O(N) + O(M) = O(N)$.
- Сложность операций:
  - ***find*** — $O(1)$;
  - ***insert*** — $O(1)$;
  - ***remove*** — $O(1)$.
- Жёсткие ограничения на множество ключей.
- При наличии $f(key)$ сводится к хеш-поиску.

4.5 Структура данных «список»

Массив — достаточно удобная структура данных для проведения с ней многих операций. Для сохранения упорядоченности массива в случае появления нового элемента требуется передвигать все элементы за позицией вставки. Чтобы избежать этой сложной операции, применим хранение элементов в другой структуре данных — списке.

Список — структура данных, которая реализует абстракции:

- `insertAfter` — добавление элемента за текущим;
- `insertBefore` — добавление элемента перед текущим;
- `insertToFront` — добавление элемента в начало списка;
- `insertToBack` — добавление элемента в конец списка;
- `find` — поиск элемента;
- `size` — определение количества элементов.

Для реализации списков обычно требуется явное использование указателей.

```
struct linkedListNode { // Одна из реализаций
  someType data; // Таких полей - произвольное число
  linkedListNode *next; // Связь со следующим
};
```

Внутренние операции создания элементов — через `malloc`, `calloc`, `new`.

```
...
linkedListNode *item = new linkedListNode();
item->data = myData;
...
```

Различные варианты представлений:

В линейном виде (рис. 4.53):

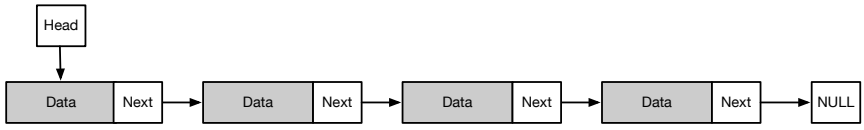


Рис. 4.53. Связный список: линейное представление

В виде кольца (рис. 4.54):

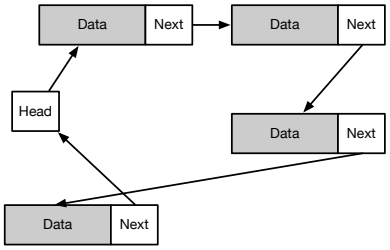


Рис. 4.54. Связный список: кольцевое представление

Стоимость операций:

Операция	Время	Память
`insertAfter`	`O(1)`	`O(1)`
`insertBefore`	`O(N)`	`O(1)`
`insertToFront`	`O(1)`	`O(1)`
`insertToEnd`	`O(N)`	`O(1)`
`find`	`O(N)`	`O(1)`
`size`	`O(N)`	`O(1)`

Вот примерный код для создания элемента.

```
typedef double myData;

linkedListNode *list_createNode(myData data) {
  linkedListNode *ret = new linkedListNode();
  ret->data = data;
  ret->next = nullptr;
  return ret;
}
```

Создание списка из одного элемента (рис. 4.55).

```
linkedListNode *head = list_createNode(555.666);
```

Рис. 4.55. Связный список из одного элемента

Добавление элемента в хвост списка (рис. 4.56). Сначала создание:

```
linkedListNode *oth = list_createNode(123.45);
```

Рис. 4.56. Связный список перед добавлением элемента в хвост

Затем стыковка.

```
head->next = oth;
```

Результат добавления — на рис. 4.57.

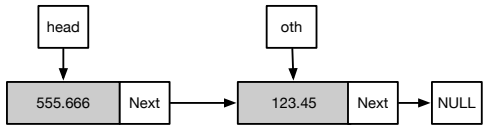


Рис. 4.57. Связный список после добавления элемента в хвост

Если список уже состоит из нескольких элементов, то добавление элемента в хвост списка всё так же несложно (рис. 4.58).

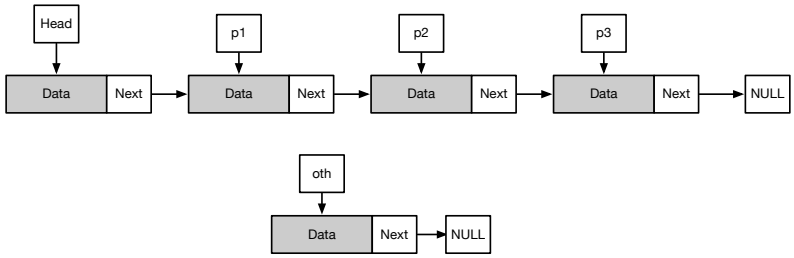


Рис. 4.58. Связный список: добавление в хвост списка из нескольких элементов

Проход по всем элементам до хвоста (`traversal`, `walk`). Если требуется, можно остановиться на нужном.

```
linkedListNode *ptr = head;
while (ptr->next != nullptr) {
  ptr = ptr->next;
}
ptr->next = oth;
```

Заключительное состояние после вставки (рис. 4.59):

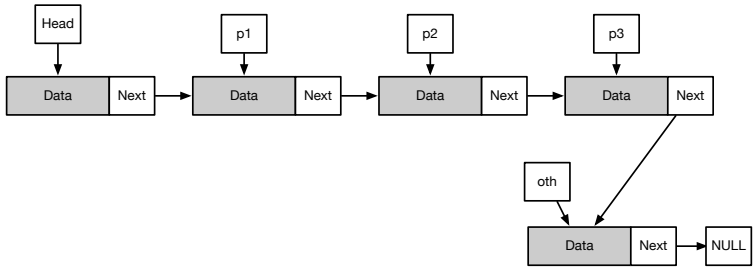


Рис. 4.59. Связный список: добавление в хвост списка из нескольких элементов, после операции

Сложность операции — $O(N)$

Вставка `insertAfter` за заданным элементом `p1` примитивна.

```
// вставка ЗА элементом p1
oth->next = p1->next;
p1->next = oth;
```

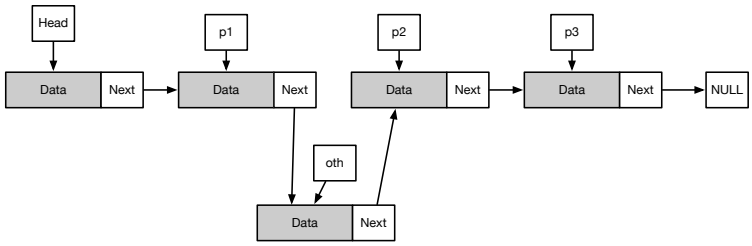


Рис. 4.60. Связный список: вставка за заданным элементом

Вставка ПЕРЕД заданным элементом сложнее, так как требуется найти предшественника этого элемента. К сожалению, нет других возможностей, кроме как перебор всех элементов с головы списка. Только после этого у нас появляется точка опоры.

```
// вставка ПЕРЕД элементом p2
linkedListNode *ptr = head;
while (ptr->next != p2) {
  ptr = ptr->next;
}
oth->next = p2;
ptr->next = oth;
```

Удаление элемента `p2` — тоже сложная операция, так как нужно найти удаляемый элемент и его предшественника. К счастью, это можно сделать за один проход.

```
// поиск элемента p2
linkedListNode *ptr = head;
while (ptr->next != p2) {
  ptr = ptr->next;
}
// ptr - предшественник p2
```

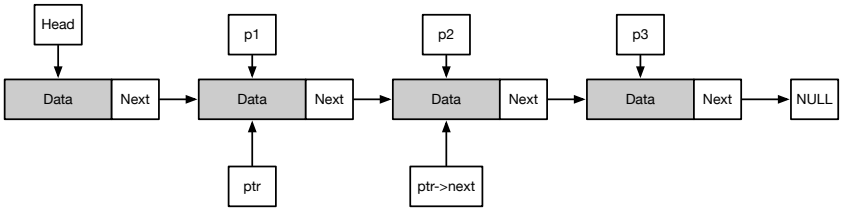


Рис. 4.61. Связный список: ищем удаляемый элемент, сохраняя предшественника

Удаление элемента из списка заключается в аккуратной перестановке указателей (рис. 4.62).

```
ptr->next = p2->next;
delete p2;
```

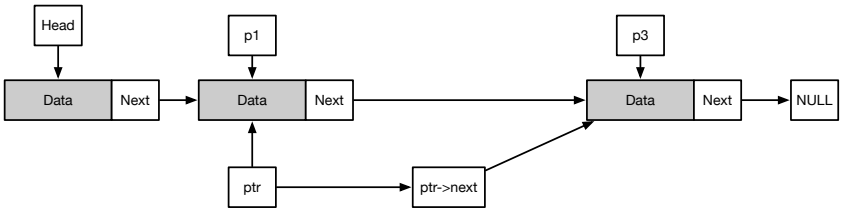


Рис. 4.62. Связный список: удаляем найденный элемент

Операцию `size` можно реализовать двояко: медленно через операцию `walk` до `nullptr(NULL)` или быстро, поддерживая инвариант размера списка в структуре данных (потребуется изменить все методы вставки/удаления).

Выбранное нами представление списка удобно не для всех применений. В реальной жизни в программе должно всегда различаться, производится ли работа с головой списка или с каким-то другим элементом. Если в программе где-то сохраняется копия головы списка в отдельной переменной, то при операциях изменения головы списка копия становится неактуальной,

что чревато проблемами в будущем (особенно если программа разрабатывается несколькими людьми). Чтобы избежать этого, удобно иметь список с неизменной головой.

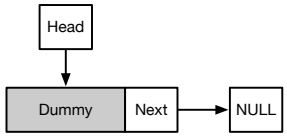


Рис. 4.63. Связные списки: хранение фиктивной головы списка в отдельном узле

Это — пустой список, содержащий ноль элементов.

Список, состоящий из одного элемента `p1` (рис. 4.64):

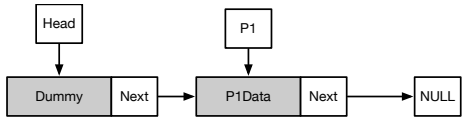


Рис. 4.64. Связный список: список с фиктивной головой и одним элементом

Такое представление упрощает реализацию за счёт одного дополнительного элемента.

Ещё раз оценим сложность основных операций:
- Вставка элемента в голову списка — $O(1)$.
- Вставка элемента в хвост списка — ***O(N)***.
- Поиск элемента — $O(N)$.
- Удаление известного элемента — ***O(N)***.
- Вставка элемента ЗА заданным — $O(1)$.
- Вставка элемента ПЕРЕД заданным — ***O(N)***.

Можно ли улучшить худшие случаи?

Худшие случаи можно улучшить, если заметить, что операция «слева-направо» реализуется более эффективно, чем операция «справа-налево» — и восстановить симметрию, добавив второй указатель, показывающий на предыдущий элемент (рис. 4.65).

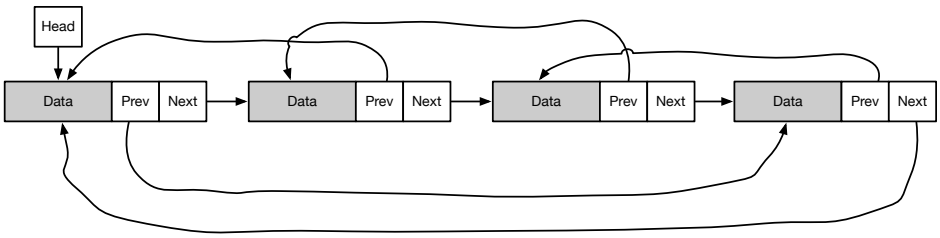


Рис. 4.65. Двусвязный список

Нетрудно обобщить уже реализованные операции над связным списком для двусвязного. Их сложность такова:

- Вставка элемента в голову списка — $O(1)$.
- Вставка элемента в хвост списка — ***O(1)***.
- Поиск элемента — $O(N)$.
- Удаление заданного элемента — ***O(1)***.
- Вставка элемента ЗА заданным — $O(1)$.
- Вставка элемента ПЕРЕД заданным — ***O(1)***.

Операции вставки и удаления несколько усложняются, так как за указателями приходится следить более тщательно.

Для вставки элемента `oth` после элемента `p1`:

1. подготавливаем вставляемый элемент;
2. сохраняем указатель `s = p1->next`;
3. `oth->prev = p1`;
4. `oth->next = s`;
5. `s->prev = oth`;
6. `p1->next = oth`;

Для удаления элемента `p1` из списка:

1. сохраняем указатель `s = p1->next`;

2. `s->prev = p1->prev;`
3. `p1->prev->next = s;`
4. Освобождаем память элемента `p1`;

Списки в практическом применении обычно используют для представления быстро изменяющегося множества объектов.

Пример из математического моделирования: множество машин при моделировании автодороги. Они:

- появляются на дороге (вставка в начало списка);
- покидают дорогу (удаление из конца списка);
- перестраиваются с полосы на полосу (удаление из одного списка и вставка в другой).

Примеры из системного программирования в ядре операционной системы

Представление множества исполняющихся процессов, претендующих на процессор. Удобно найти процесс, которому будет передано исполнение (голова списка), и перенести его в другой список в конец.

Представление множества запросов ввода/вывода. Для одного процесса множество запросов ввода-вывода часто упорядочено — и его удобно представлять в виде списка.

Важная особенность: лёгкий одновременный доступ от различных процессорных ядер. Оказывается, для операций над односвязными списками существуют атомарные операции вставки и удаления — и это делает такие списки отличными кандидатами для представления таких множеств.

Ещё пример из системного программирования: одна из простейших реализаций выделения/освобождения динамической памяти (`calloc/new/free/delete`).

- Вначале свободная память описывается пустым списком.
- Память в операционной системе выделяется *страницами*.
- При заказе памяти:
  - если есть достаточный свободный блок памяти, то он разбивается на два подблока, один из которых помечается занятым и возвращается в программу;
  - если нет достаточной свободной памяти, запрашивается несколько страниц у системы и создаётся новый элемент в конце списка (или изменяется старый).

4.5.1 Абстракция *очередь*

С помощью списков легко реализовать абстракцию *очередь*, реализующую (помимо неизбежных операций создания и удаления) следующий набор операций:

- **enqueue**: добавить элемент в конец очереди;
- **dequeue**: извлечь с удалением элемент из начала очереди;
- **empty**: определить, пуста ли очередь.

Ещё одно популярное название этой абстракции — `FIFO`, First In First Out.

Эта абстракция, наряду с похожей на неё абстракцией стека (`LIFO`), будет широко использоваться в алгоритмах на графах.

4.6 Структура данных «дерево»

Отличие деревьев от списков — наличие нескольких наследников. Несколько самых популярных деревьев: двоичные (бинарные), троичные (тернарные). Остальные обычно называются `N`-ричными.

4.6.1 Представление деревьев

Компьютерные представления деревьев обычно достаточно прямолинейны и подчиняются следующим соглашениям.

- Любое `N`-ричное дерево может представлять деревья меньшего порядка.
- Если потомка нет, соответствующий указатель равен `NULL` (или `nullptr` для современных версий `C++`).
- Деревья `1`-ричного порядка существуют (списки).

```
struct tree {
  tree *children[3];
  myType data;
  ...
};
```

Пример троичного дерева, или дерева 3-порядка.

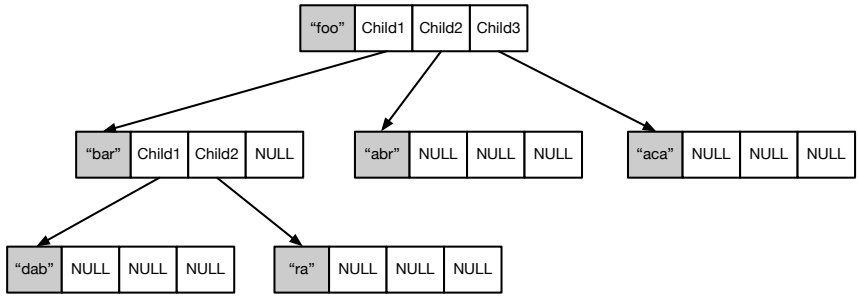


Рис. 4.66. (3)-дерево

Узлы в деревьях часто неравнозначны, и их делят на две группы:

- ***Вершины***, не содержащие связей с потомками. Другое название — терминальные вершины или терминальные узлы.
- ***Узлы***, содержащие связи с потомками.

Вот ещё термины:

- ***Родитель (parent)*** — узел, потомком которого является данный.
- ***Дети (children)*** — потомки данного узла.
- ***Братья (sibs)*** — узлы, имеющие одного родителя.
- ***Глубина (depth)*** — длина пути от корня до узла $D_{node} = D_{parent} + 1$.

Создание элемента (узла) дерева — конструктор объекта «узел».

```
struct tree {
  string data;
  tree *child[3];
  tree(string init) { //  Конструктор
    child[0] = child[1] = child[2] = nullptr;
    data = init;
  }
  ...
};
```

Чтобы построить дерево, показанное на рисунке, достаточно исполнить следующий код:

```
tree *root = new tree("foo");
root->child[0] = new tree("bar");
```

```
root->child[1] = new tree("abr");
root->child[2] = new tree("aca");
root->child[0]->child[0] = new tree("dab");
root->child[0]->child[1] = new tree("ra");
```

Деревья — одна из популярнейших структур данных, и их использование можно наблюдать в различных разделах Computer Science.

Указатели — это хорошо и красиво. Однако их использование подразумевает заказ динамической памяти и её освобождение, что не является очень быстрой операцией. В некоторых применениях можно обойтись и без указателей, храня узлы в заранее заказанном массиве. Для этого все возможные узлы нумеруются, например, начиная с 0. Тогда для `N`-дерева для узла с номером K номера детей будут $K \cdot N + 1 \ldots K \cdot N + N$. Например, для 2-дерева корневой узел будет иметь номер 0, узлы первого уровня ($D = 2$) — номера 1 и 2, второго ($D = 3$) — от 3 до 6.

Для двоичных деревьев узлы часто удобнее нумеровать с 1. Тогда нумерация и детей, и родителей упрощается. Например, для родителя под номером 10 дети будут иметь номера 20 и 21 ($2K, 2K + 1$), а для ребёнка номер 35 родителем будет узел под номером 17 ($K/2$). Это особенно удобно, если учесть, что операции умножения и деления на два можно производить с помощью побитовых операций сдвига.

Конечно, представление бинарного дерева в виде массива эффективно только для тех деревьев, которые всегда или почти всегда имеют двух потомков у каждого родителя. Мы будем называть такие деревья *плотными*.

Приведём пример неудачного выбора представления. Все современные компиляторы в какой-то момент времени представляют выражения языков программирования в виде деревьев. Оказывается, это представление удобно для проведения каких-либо преобразований программы, например, распространения констант (вычисления константных выражений на этапе компиляции).

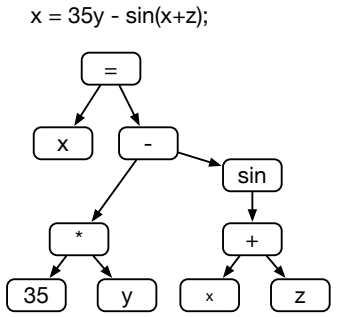


Рис. 4.67. Представление выражения языка программирования в виде дерева

Если дерево представлено в виде указателей, то расход памяти минимален.

Если выбрать представление в виде массива, то для данного выражения нумерация узлов будет такой.

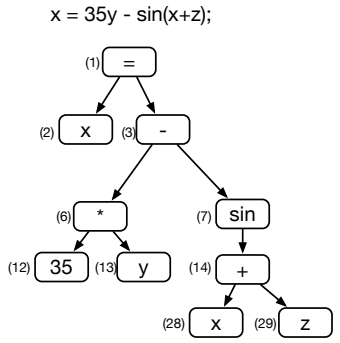


Рис. 4.68. Нумерация узлов в дереве, представляющем выражение

Представление в виде массива потребует значительного количества памяти:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	28	29
=	x	-			*	sin					35	y	+		x	z

Количество требуемой памяти в таких массивах есть $O(2^{D_{max}})$, что при разреженном дереве крайне невыгодно. Впрочем, плотные деревья встречаются не так уж и редко, и мы с ними вскоре столкнёмся.

4.6.2 Обход деревьев

Дерево — рекурсивная по своей сущности структура данных, и алгоритмы работы с деревьями часто рекурсивны. Одна из распространённых задач — посетить все узлы дерева (*обойти* его).

Для бинарного дерева порядок обхода определяется порядком обработки трёх сущностей — левого потомка, правого потомка и самого узла. Таким образом, существует `6=3!` способов обхода бинарного дерева. На практике чаще всего применяют четыре основных варианта рекурсивного обхода:

- прямой;
- симметричный;
- обратный;
- обратно симметричный.

При прямом способе обхода сначала обрабатывают сам узел, затем рекурсивно посещают левого и правого потомков именно в этом порядке.

```
void walk(tree *t) {
  work(t);
  if (t->left != nullptr) walk(t->left);
  if (t->right != nullptr) walk(t->right);
}
```

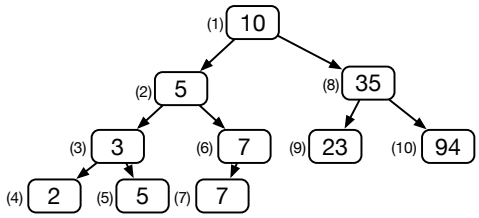


Рис. 4.69. Обход дерева: прямой

Симметричный способ обхода инициирует посещение сначала левого потомка, затем обработку самого узла, после чего посещается правый потомок.

```
void walk(tree *t) {
  if (t->left != nullptr) walk(t->left);
  work(t);
  if (t->right != nullptr) walk(t->right);
}
```

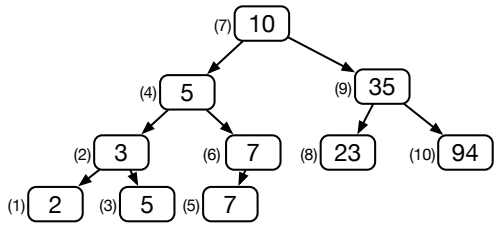


Рис. 4.70. Обход дерева: симметричный

При обратном способе обхода сначала посещаются потомки, левый и правый, и только потом очередь доходит до самого узла.

```
void walk(tree *t) {
  if (t->left != nullptr) walk(t->left);
  if (t->right != nullptr) walk(t->right);
  work(t);
}
```

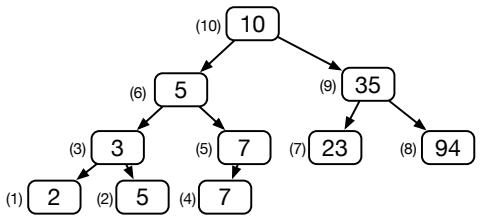


Рис. 4.71. Обход дерева: обратный

Очень удобно бывает параметризовать функцию обработки узла. В следующем примере объявляется тип данных `walkFunction` — как указатель на функцию, принимающую указатель на узел дерева и ничего не возвращающую.

```
using walkFunction = void (*)(tree *);
  void walk(tree *t, walkFunction wf) {
  if (t->left != nullptr) walk(t->left, wf);
  if (t->right != nullptr) walk(t->right, wf);
  wf(t);
}

void printData(tree *t) {
  printf("t[%p]='%s'\n", t, t->data.c_str());
}

int main() {
  tree *root = new tree("foo");
  root->left = new tree("bar");
  root->right = new tree("abr");
  root->left->left = new tree("aca");
```

```
  root->left->left->left = new tree("dab");
  root->left->right = new tree("ra");
  walk(root, printData);
}
```

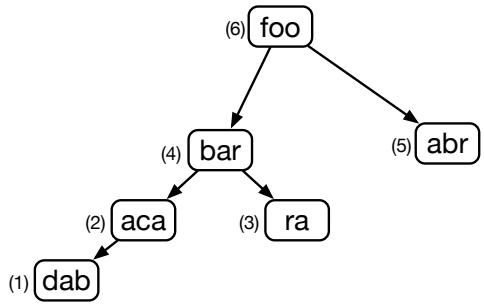


Рис. 4.72. Пример обхода двоичного дерева

```
t[0x7ff0e1c03290]='dab'
t[0x7ff0e1c03260]='aca'
t[0x7ff0e1c032d0]='ra'
t[0x7ff0e1c03200]='bar'
t[0x7ff0e1c03230]='abr'
t[0x7ff0e1c031d0]='foo'
```

Последний интересный обход — обратно симметричный. Его ещё называют выводом генеалогического дерева.

```
using walkFunction = void (*)(tree *, int lev);

void walk(tree *t, walkFunction wf, int lev) {
  if (t->right != nullptr) walk(t->right, wf, lev+1);
  wf(t, lev);
  if (t->left != nullptr) walk(t->left, wf, lev+1);
}

void printData(tree *t, int lev) {
  for (int i = 0; i < lev; i++) {
```

```
    printf("  ");
  }
  printf("%s\n", t->data.c_str());
}

int main() {
  ...
  walk(root, printData, 0);
}
```

Вывод программы:

```
  abr
foo
    ra
  bar
    aca
      dab
```

Кажется странным? Посмотрите на него справа и представьте, что слово `foo` есть корень растущего вниз дерева.

При использовании динамических структур данных для некоторых операций важно выбрать верный обход дерева. Вспомним префиксное дерево из второй лекции (рис. 2.32).

Мы оставили на потом важный вопрос — освобождение памяти от всех заказанных объектов. Заказ памяти под узлы (и поддеревья) происходил динамически, оператором `new`. Имелся узел `root`, от которого шло построение дерева. Выбранная нами структура данных не предусматривала хранение информации о предках, таким образом корневой узел являлся центром всего построения.

Мы должны отметить важный факт, что при освобождении заказанной памяти все поля объекта, который мы удаляем, даже сохранённые, скорее всего, исказятся.

Для верного понимания происходящего напомним порядок выделения и уничтожения памяти в конструкторе и деструкторе объектов на примере следующей структуры:

```
struct node {
  node *children[3];
  bool  is_leaf;
  node();
  ~node();
};
```

Конструктор:

```
node::node() {
  children[0] = children[1] = children[2] = nullptr;
  is_leaf = false;
}
```

1. Система выделяет память из *кучи*, достаточную для хранения всех полей структуры.
2. После этого исполняется инициализация полей (написанный нами код).

Деструктор:

```
node::~node() {
  printf("node destructor is called\n");
}
```

1. Исполняется написанный нами код.
2. Система освобождает занятую память, после чего обращение к освобождённой памяти приводит к ошибкам.

Именно по указанным причинам нужно правильно спланировать обход дерева для полного освобождения заказанных ресурсов. Удаление корневого узла приводит к тому, что остальные узлы останутся недоступны. Такие недоступные узлы называются *мусором*. Указатель, для которого выполнена операция `delete`, становится *висячей ссылкой* (*dangling pointer*), так как он теперь не указывает ни на что реально полезное. Попытка проведения операции `delete` над потомками, скорее всего, приведёт к аварийному завершению программы. Если же эту операцию не проводить, то возникнет более тяжёлая для диагностики ситуация — *утечка памяти* (*memory leak*).

Чтобы не было ни аварийного завершения, ни утечки памяти, удаление узлов нужно производить с самого нижнего.

Итак, вначале имеется следующее дерево, серым цветом помечены терминальные узлы — и только их мы имеем право удалять вначале. Это означает, что мы должны выбрать обратный обход.

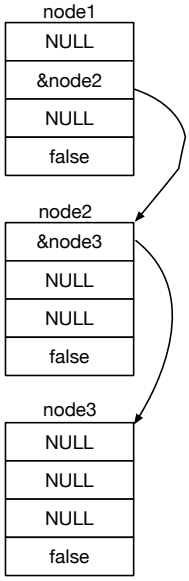


Рис. 4.73. Размещение дерева в памяти

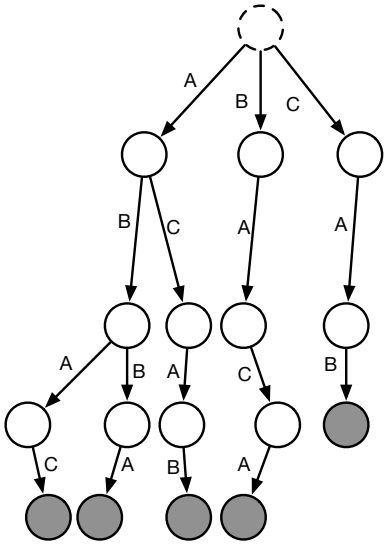


Рис. 4.74. Префиксное дерево: серым цветом помечены терминальные вершины, штриховыми линиями — удаляемые узлы

```
void destroy(node *n) {
  for (int i = 0; i < 3; i++) {
    if (n->children[i] != nullptr) {
      destroy(n->children[i]);
    }
  }
  delete n;
}
```

Добравшись до терминального узла, мы получаем право его удалить. Как только у узла удалены все потомки, узел тоже можно удалять (рис. 4.75).

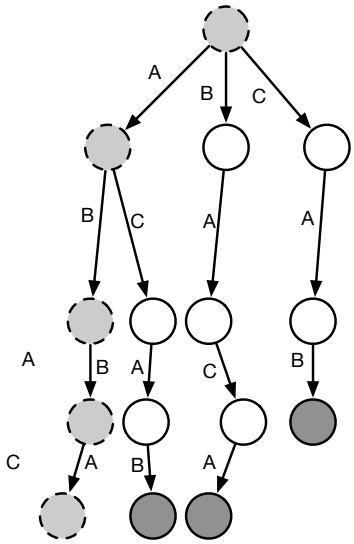


Рис. 4.75. Префиксное дерево: самую левую нижнюю вершину можно безопасно удалить

4.7 Бинарная куча и *приоритетная очередь*

Наступило время извлечь пользу из уже обсуждённого нами способа хранения бинарных деревьев в массиве.

Определение 19. ***Полное бинарное дерево*** T_H *высоты* H *есть бинарное дерево, у которого путь от корня до любой вершины содержит ровно* H *рёбер, при этом у всех узлов дерева, не являющихся листьями, есть и правый, и левый потомок.*

Полное бинарное дерево высоты 3 выглядит так (рис. 4.76):

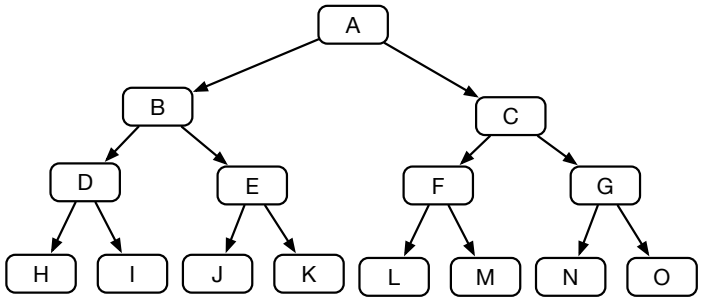


Рис. 4.76. Пример полного бинарного дерева

Для рекурсивной структуры данных можно дать рекурсивное определение:

Определение 20. ***Полное бинарное дерево*** T_H *высоты* H *есть бинарное дерево, у которого к корню прикреплены левое и правое полные бинарные поддеревья* T_{H-1} *высоты* $H-1$.

По этому определению, число узлов в дереве T_H есть $N = 2^{H+1} - 1$,

$$H = \log_2 (N+1),$$

Мы уже знаем про абстракцию обычной очереди, когда первый поступивший в очередь элемент будет первым же извлечён. В ряде случаев такой

стратегии недостаточно — и требуется как-то упорядочивать поступающих в очередь, например, по их приоритету. Классический пример — очередь к врачу. Посетителя с сильной зубной болью могут принять вне очереди. А если таких больных несколько, то из них тоже образуется очередь. Проще считать, что очередь — одна, но у каждого посетителя свой приоритет.

Определение 21. ***Приоритетная очередь*** *(`priority queue`) — очередь, элементы которой имеют приоритет, влияющий на порядок извлечения. Первым извлекается наиболее приоритетный элемент.*

На первый взгляд, простейшим способом поддержания приоритетной очереди мог бы быть полностью упорядоченный массив. Однако стоимость поддержания упорядоченности в массиве весьма велика. Например, обнаружить самый приоритетный элемент легко — достаточно отсортировать массив в нужном порядке, и он будет находиться в самом конце. Извлечение этого элемента займёт $O(1)$. А вот со вставкой будет намного хуже. Бинарным поиском мы обнаружим место вставки (это потребует сложности $O(\log N)$), а что затем? Затем придётся сдвигать все элементы правее точки вставки вправо, что даст сложность $O(N)$.

Перед тем, как реализовать приоритетную очередь, определимся с её абстракцией, то есть с теми методами, которые нам необходимы.

Интерфейс абстракции *приоритетная очередь*:

- `insert` — добавляет элемент в очередь;
- `fetchPriorityElement` — получает самый приоритетный элемент, но не извлекает его из очереди;
- `extractPriorityElement` — извлекает самый приоритетный элемент из очереди.

К счастью, требуется реализовать только заданный интерфейс, а это значит, что мы вольны выбирать подходящее нам внутреннее представление.

С точки зрения использования элементы будут последовательно извлекаться в порядке от самого приоритетного. Например, следующий список городов с их населением может быть представлен в виде приоритетной по убыванию населения очереди:

Значение (`value`)	Приоритет (`priority`)
Москва	12000000
Казань	1500000
Урюпинск	10000
Малиновка	200

Определение 22. ***Бинарная куча*** — *бинарное дерево, удовлетворяющее следующим условиям:*

- *Приоритет любой вершины не меньше приоритета потомков.*
- *Дерево является правильным подмножеством полного бинарного, допускающим плотное хранение узлов в массиве.*

Другое название этой структуры данных — *пирамида* (`heap`).

В невозрастающей пирамиде приоритет каждого родителя не меньше приоритета потомков.

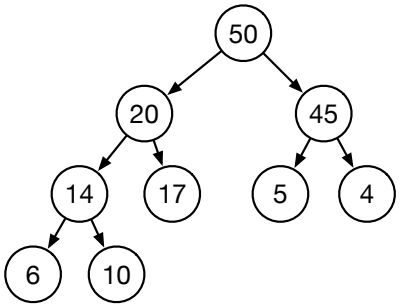


Рис. 4.77. Бинарная куча: невозрастающая пирамида

Так как дерево с узлами — плотное, удобно хранить его в виде массива массива с индексами от 1 до N.

50	20	45	14	17	5	4	6	10

Удобство такого хранения трудно переоценить:

- Индекс корня дерева всегда равен 1 — самый приоритетный элемент.
- Индекс родителя узла i всегда равен $\left\lfloor \frac{i}{2} \right\rfloor$.
- Индекс левого потомка узла i всегда равен $2i$.
- Индекс правого потомка узле i всегда равен $2i + 1$.

```
struct bhnode { // Узел
  string data;
  int priority;
};
```

```
struct binary_heap {
  bhnode *body;
  int     bodysize;
  int     numnodes;
  binary_heap(int maxsize);
  ...
};
```

Операция создания бинарной кучи определённого размера заключается в простом выделении памяти под массив, хранящий элементы. Нулевой элемент массива мы использовать не будем. Будем фиксировать количество помещённых в кучу элементов в переменной `numnodes`. Ещё нам понадобится операция обмена элементов кучи по их индексам.

```
binary_heap::binary_heap(int maxsize) {
  body = new bhnode[maxsize+1];
  bodysize = maxsize;
  numnodes = 0;
}

~binary_heap::binary_heap() {
  delete body;
}

void binary_heap::swap(int a, int b) {
  std::swap(body[a],body[b]);
}
```

Сложность операции создания бинарной кучи — $T_{create} = O(N)$.

Операция поиска самого приоритетного элемента тривиальна. Её сложность — $T_{fetchMin} = O(1)$.

```
bhnode *binary_heap::fetchPriorityElement() {
  return numnodes == 0? nullptr : body + 1;
}
```

Операция добавления элемента требует небольшой эквилибристики. Правильная бинарная куча должна поддерживать два инварианта — структурной целостности, то есть представимости в виде бинарного дерева, и упорядоченной целостности, то есть свойства «потомки узла не могут иметь приоритет, больший, чем у родителя». Структурную целостность поддерживать сложнее, поэтому мы с неё и начнём.

Этап 1. Вставка в конец кучи.

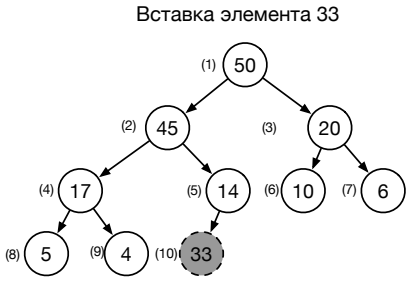


Рис. 4.78. Бинарная куча: вставка элемента в конец кучи

Отлично! *Структура* кучи не испортилась!

50	45	20	17	14	10	6	5	4	33

Однако пока не выдержана упорядоченность.

Этап 2. Корректировка значений.

Только что вставленный элемент может оказаться более приоритетным, чем его родитель. Тогда поменяем их местами.

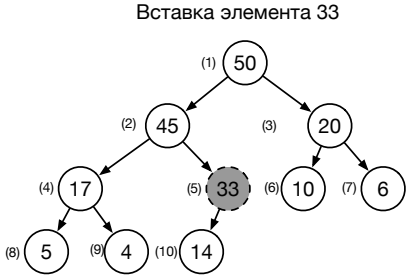


Рис. 4.79. Бинарная куча: подъём более приоритетного элемента вверх

Куча удовлетворяет всем условиям.

50	45	20	17	33	10	6	5	4	14

Попытаемся вставить элемент, который имеет приоритет больше, чем все элементы в куче.

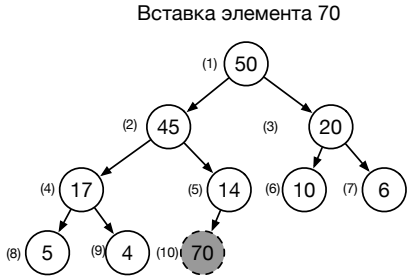


Рис. 4.80. Бинарная куча: вставка элемента с большим приоритетом

50	45	20	17	14	10	6	5	4	70

Он находится не на своём месте — и меняется местами с родителем (ползёт вверх по дереву).

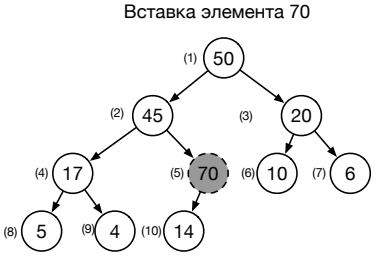


Рис. 4.81. Бинарная куча: продвижение элемента вверх, шаг 1

50	45	20	17	70	10	6	5	4	14

Опять он не на своём месте — и опять меняется местами с родителем.

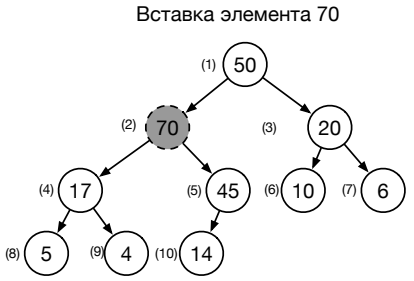


Рис. 4.82. Бинарная куча: продвижение элемента вверх, шаг 2

50	70	20	17	45	10	6	5	4	14

Максимальный элемент переместился в корень. Алгоритм завершён.

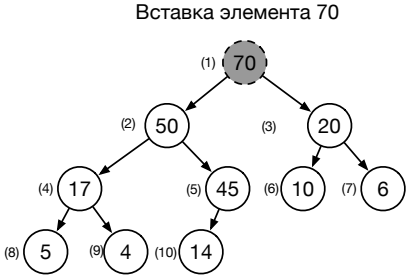


Рис. 4.83. Бинарная куча: продвижение элемента вверх, финал

70	50	20	17	45	10	6	5	4	14

Корректность алгоритма базируется на двух фактах:

- Инвариант структурной целостности не нарушен ни в один момент времени.
- После каждого шага перемещения вставленного элемента поддерево с корнем в текущем элементе поддерживает инвариант упорядоченной целостности.

Сложность алгоритма определяется высотой дерева и составляет $T_{Insert} = O(\log N)$.

```
int binary_heap::insert(bhnode node) {
  if (numnodes > bodysize) {
    return -1; // или расширяем.
  }
  body[++numnodes] = node;
  for (int i = numnodes; i > 1 && body[i].priority > body[i/2].priority;
    i /= 2) {
    swap(i, i/2);
  }
}
```

Операция удаления самого приоритетного элемента кажется более сложной — ведь после удаления корневого элемента нарушается структурная целостность. Восстанавливать структурную целостность, сохраняя упорядоченность — задача не из простых. Выход здесь заключается в повторном использовании принципа: делать сначала то, после чего существует простой путь к исправлению. Как мы выяснили при операции вставки, упорядоченную целостность восстанавливать легче, поэтому первый шаг при удалении корневого элемента должен сохранять структурную целостность. Так как после удаления корня количество элементов уменьшится на один, то отправляем самый последний элемент кучи в корень, уменьшая при этом `numnodes` (рис. 4.84).

Структурная целостность не изменилась, но могла измениться упорядоченная целостность. Требуется восстановление, функция `heapify`.

```
void binary_heap::heapify(int index) {
  for (;;) {
    int left = index + index;
    int right = left + 1;
    // Кто больше, [index], [left], [right]?
    int largest = index;
    if (left <= numnodes &&
     body[left].priority > body[index].priority)
```

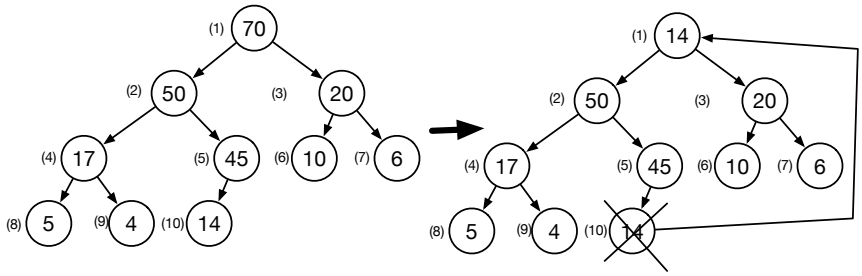


Рис. 4.84. Бинарная куча: удаление. Перемещение последнего элемента в корень

```
      largest = left;
    if (right <= numnodes &&
     body[right].priority > body[largest].priority)
      largest = right;
    if (largest == index) break;
    swap(index, largest);
    index = largest;
  }
}
```

Идея функции проста: начиная с корневого элемента, мы проводим соревнование между тремя кандидатами — теми, кто может занять это место. Если кандидаты (левый и правый потомки) менее приоритетны, чем текущий претендент, то алгоритм завершён. Иначе претендент меняется местами с самым приоритетным из потомков — и операция повторяется уже на уровне ниже. Так как каждая операция обмена передвигает претендента на один уровень вниз, процесс обязательно завершается не более, чем за $O(\log N)$ шагов, что и составляет сложность данного алгоритма.

Проиллюстрируем алгоритм на примере. После перемещения последнего узла (14) в корень он становится претендентом, а кандидатами оказываются узлы (50) и (20). Индекс восстановления (номер претендента) пока равен 1 (рис. 4.85).

Претендента обменяли на кандидата (50) с индексом 2. Теперь элемент

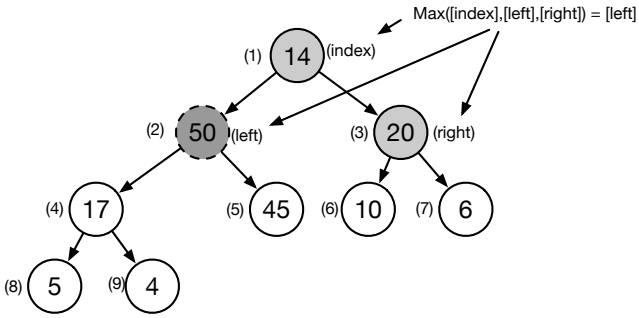


Рис. 4.85. Бинарная куча: удаление. Соревнование за место в корне

под этим индексом — новый претендент (рис. 4.86).

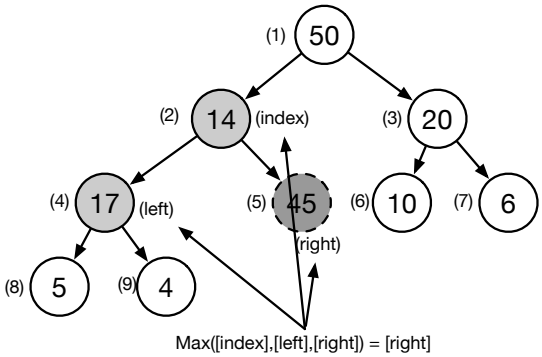


Рис. 4.86. Бинарная куча: удаление. Соревнование на следующем уровне

Теперь претендентом становится элемент с индексом 5. После завершения этой операции восстановление завершено (рис. 4.87).

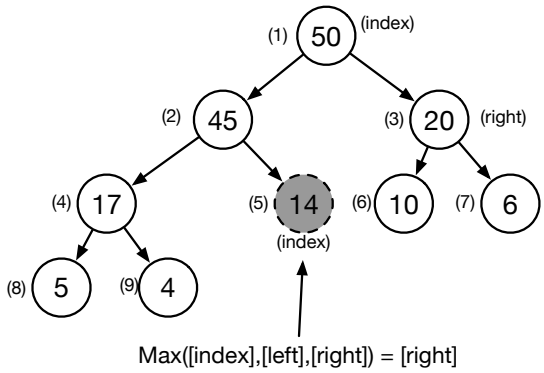


Рис. 4.87. Бинарная куча: соревнование завершено

4.8 HeapSort

Возможность получать из бинарной кучи самый приоритетный элемент за $O(\log N)$ и добавлять элементы в бинарную кучу за $O(\log N)$ вызывает желание реализовать ещё один алгоритм сортировки. Что интересно, этот алгоритм будет иметь сложность $O(N \log N)$ в *худшем* случае.

Для доказательства этого рассмотрим сначала простой метод реализации сортировки массива размером N.

1. Создать бинарную кучу размером N. Это потребует сложности $O(N)$.
2. Поочерёдно вставить в неё все N элементов массива. Сложность этого этапа есть

$$O(\log 1) + O(\log 2) + \cdots + O(\log N)) < O(\log N) + \cdots + O(\log N)) = N \log N.$$

3. Поочерёдно извлекать с удалением самый приоритетный элемент из бинарной кучи — с помещением в последовательные N позиций исходного массива.

Сложность этого этапа есть

$$O(\log N) + \cdots + O(\log 2) + O(\log 1)) < O(\log N) + \cdots + O(\log N) = N \log N.$$

Общая сложность всех этапов опять составляет $O(N \log N)$.

Такая прямолинейная организация не особенно хороша: потребуется добавочная память на бинарную кучу размером N элементов. Небольшая хитрость — и добавочной памяти можно избежать.

Модифицируем функцию `heapify` для того, чтобы она могла работать с произвольным массивом, адресуемым с нуля:

```
void heapify(int *a, int i, int n)
{
  int curr = a[i];
  int index = i;
  for (;;) {
    int left = index + index + 1;
    int right = left + 1;
    if ( left < n && a[left] > curr)
      index = left;
    if ( right < n && a[right] > a[index])
      index = right;
    if (index == i ) break;
    a[i] = a[index];
    a[index] = curr;
    i = index;
  }
}
```

Теперь сортировка заключается в том, что мы создаём бинарную кучу размером n на месте исходного массива, переставляя его элементы. Затем на шаге i мы обмениваем самый приоритетный элемент кучи (он всегда располагается на позиции 0) с элементом под номером $n - i - 1$. Размер кучи при этом уменьшается на единицу, а самый приоритетный элемент занимает теперь положенное ему по рангу место.

```
void sort_heap(int *a, int n) {
  for(int i = n/2-1; i >= 0; i--) {
    heapify(a, i, n);
  }
  while( n > 1 ) {
```

```
    n--;
    swap(a[0],a[n]);
    heapify(a, 0, n);
  }
}
```

Возникает вопрос: если эта сортировка гарантирует нам сложность $O(N \log N)$ даже в самом худшем случае, а быстрая сортировка, `QuickSort`, не гарантирует, то почему не использовать только эту сортировку? Кстати, эта сортировка была основной на компьютерах, выпускаемых в 1960-1970 годы.

Первая причина в том, что в быстрой сортировке используется меньшее количество операций обмена с памятью, а излишней работы с памятью на современных компьютерах стоит избегать. Вторая причина тоже связана с памятью, точнее — с тем фактом, что N обращений к последовательным ячейкам памяти исполняется до 10-15 раз быстрее, чем столько же обращений к случайным ячейкам памяти. Это связано с наличием ограниченного количества аппаратной *кэш-памяти* на современных процессорах. При наличии различных алгоритмов, исполняющих одну и ту же задачу, некоторые из них могут быть *дружелюбны к кэшу* (*cache-friendly*), а некоторые — нет. Поэтому наилучшие по времени исполнения алгоритмы могут быть разными в различное время и на различных вычислительных системах.

4.9 Домашние задания

Задача 16. Провода

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

На складе есть провода различной целочисленной длины. Их можно разрезать на части. Необходимо получить K кусочков одинаковой целочисленной и как можно большей длины. Найти максимальную длину M, при которой можно получить по меньшей мере K кусочков этой длины. Все оставшиеся на складе куски проводов длиной, меньшей M, в подсчёте не участвуют.

Формат входных данных

В первой строке — количество проводов на складе N и необходимое количество кусочков K. В следующих N строках — длины проводов.

$1 \leqslant N \leqslant 100000$

$1 \leqslant K \leqslant 100000$

Формат выходных данных

M — максимальная длина, на которую можно разрезать все провода так, чтобы получилось не менее K кусочков.

Примеры

стандартный ввод	стандартный вывод
5 7 15 12 5 13 6	6

Задача 17. Брокеры

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	64 мегабайта

В стране Бурляндии фирма «Котлетный рай» имеет много отделений, работающих сравнительно автономно. После неких экономических преобразований такая форма функционирования оказалась невыгодной — и фирма решила сливать капиталы отделений, образуя укрупнённые департаменты, отвечающие за несколько отделений сразу. Цель фирмы — слить все отделения в один громадный департамент, владеющий всеми капиталами.

Проблема заключается в том, что, по законам Бурляндии, операция слияния капиталов отделений должна проводиться государственной брокерской службой, которая не может производить более одной операции слияния в одной фирме одновременно. Вторая проблема состоит в том, что брокерская служба берёт за свои услуги один процент всех средств, получившихся в результате слияния двух подразделений. Важно спланировать порядок операций слияния таким образом, чтобы фирма потратила на слияние как можно меньшую сумму.

Формат входных данных

На вход программы подаётся число отделений $2 \leqslant N \leqslant 1000000$, за которым следует N капиталов отделений $1 \leqslant C_i \leqslant 1000000$.

Формат выходных данных

T — минимальная сумма из возможных, которую должна заплатить брокерам фирма «Котлетный рай», с двумя знаками после запятой.

Примеры

стандартный ввод	стандартный вывод
5 1 2 3 4 5	0.33
10 2 10 100 30 7 4 15 2 15 80	6.52

Задача 18. Пересечение множеств

Имя входного файла:	стандартный ввод
Имя выходного файла:	стандартный вывод
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Задано $2 \leqslant N \leqslant 1000$ множеств из $3 \leqslant M \leqslant 10000$ элементов, значения которых могут находиться в диапазоне от -2000000000 до 2000000000. Значения каждого множества задаются в произвольном порядке. Гарантируется, что для одного множества все задаваемые значения — различные.

Требуется найти наибольший размер множества, получаемого при пересечении какой-либо из пар заданных множеств.

Формат входных данных

```
N M
A_1[1] A_1[2] ... A_1[M]
A_2[1] A_2[2] ... A_2[M]
......
A_N[1] A_N[2] ... A_N[M]
```

Примеры

стандартный ввод	стандартный вывод
3 4 9 7 1 8 5 7 6 3 5 9 8 6	2
4 5 -2 6 8 4 -1 5 3 10 -5 -1 7 8 -5 -1 -2 -1 8 4 9 0	3

Задача 19. Составные строки

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	0.5 секунд
Ограничение по памяти:	16 мегабайт

В первой строке входного файла содержится число $4 \leqslant N \leqslant 1000$, последующие N строк состоят только из заглавных букв латинского алфавита и образуют множество, то есть равных элементов среди них нет. Длина строки не превышает 1000 букв. Некоторые из этих строк можно составить, приписав друг за другом каких-то два элемента множества, возможно, совпадающие. Например, если исходное множество содержит элементы `A, B, AB, AA, C, ABC`, то элемент `AB` можно составить из элементов `A` и `B`, а строку `AA` — из элементов `A` и `A`.

Ваша задача — вывести все элементы множества, которые можно составить из пары элементов множества, по одному на строку в лексикографически возрастающем порядке.

Примеры

стандартный ввод	стандартный вывод
5 A AB B AA ABC	AA AB
10 ABC DEFG AB ABCAB DEFGA FG ABFG ABCAFG FGFG ABABC	ABCAB ABFG FGFG

Задача 20. Длинное деление

Имя входного файла:	стандартный ввод
Имя выходного файла:	стандартный вывод
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

На вход подаётся три строки. Первая содержит представление длинного неотрицательного десятичного числа (первый операнд), вторая — всегда строка /, третья — десятичное представление второго положительного операнда.

Длины первой и третьей строки ограничены 100000 символами. Вторая строка содержит ровно один символ /.

Требуется исполнить операцию и вывести результат в десятичном представлении.

Формат входных данных

999
/
9

Формат выходных данных

111

Пример

стандартный ввод	стандартный вывод
11876187623876123293872987 / 1987987198711	5973975904662

Лекция 5

5.1 Абстракция *отображение*

Абстракция *массив* устанавливает соответствие между номерами элементов (или, как говорят, *индексами*) и их значениями. Важное свойство массивов — возможность «пробежать» по всем возможным индексам, получив все значения по ним. Второе важное свойство — сложность операций извлечения по индексу и установки значения по индексу очень мала и составляет $O(1)$. Можно сказать, что абстракция *массив* устанавливает соответствие между подмножеством множества неотрицательных целых чисел в заданном диапазоне (множества ключей) и множеством значений по этим ключам. Если расширить множество ключей до произвольных величин, то мы приходим к понятию абстракции *отображение*.

Рис. 5.88. Пример отображения: каждому городу соответствует численность его населения

Определение 23. ***Отображение*** — *абстракция, устанавливающая направленное соответствие между двумя множествами (множеством ключей и множеством данных) и реализующая над ними определённые операции.*

Абстракция *отображение* есть аналог дискретной функции. Одно из определений математической функции: ***Функция есть отображение множества*** D ***на множество*** E (рис. 5.89).

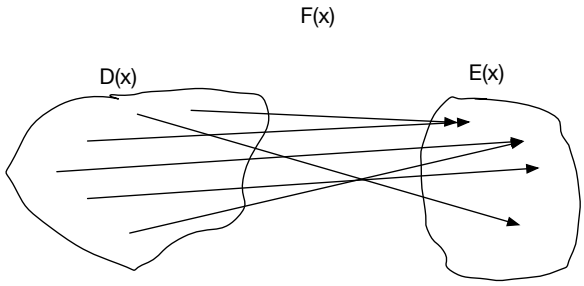


Рис. 5.89. Отображение множества $D(x)$ на множество $E(x)$

Эта абстракция — обязательный элемент, присутствующий в любом современном языке программирования. Можно сказать, что наша цель — реализовать некий *словарь*, в котором мы можем добавлять, искать и удалять *словарные статьи*.

C++, наряду с другими современными языками, предоставляет возможность использовать отображения как обобщение понятия *массив* с помощью синтаксиса индексации:

```
map m<string,int>;
m["Шанхай"] = 24150000;
m["Карачи"] = 23500000;
m["Пекин"]  = 21150000;
m["Дели"]   = 17830000;
...
int BeijingPopulation = m["Пекин"];
...
for (auto x: m) {
```

```
  printf("Population of '%s' is %d\n",
  x.first, x.second);
}
```

Интерфейс абстракции *отображение* есть частный случай абстракции хранилища `CRUD`.

- ***insert(key, value)*** — добавить элемент с ключом *key* и значением *value*.
- *Item* ***find(key)*** — найти элемент с ключом *key* и вернуть его.
- ***erase(key)*** — удалить элемент с ключом *key*.
- ***walk*** — получить все ключи (или все пары *ключ/значение*) в какомлибо порядке.

В дальнейшем под термином ***ключ*** мы понимаем пару ***ключ+значение***, в которой определена операция сравнения по ключу.

Абстракцию *множество* можно рассматривать как частный случай абстракции *отображение*. Например, одна из возможных реализаций отображения — множество с прикреплёнными данными. С другой стороны — можно абстрагировать понятие *множество* как отображение набора ключей на логические переменные: все присутствующие в множестве элементы отображаются на логическую истину, остальные — на логическую ложь. Одна из удобных и эффективных структур данных для представления и множеств, и отображений — бинарное дерево поиска.

5.2 Бинарные деревья поиска

Определение 24. ***Бинарным деревом поиска*** *называется бинарное дерево, в котором все узлы, находящиеся справа от родителя, имеют значения, не меньшие значения в родительском узле, а слева — не большие этого значения.*

Задача. На вход подаётся последовательность чисел. Выходом должно быть двоичное дерево поиска.

Бинарные деревья поиска часто обозначаются аббревиатурой `BST` — Binary Search Tree.

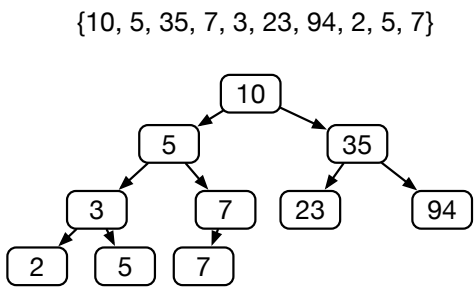


Рис. 5.90. BST: пример

Алгоритм поиска элемента в `BST`, содержащего ключ X.

1. Делаем текущий узел корневым.
2. Переходим в текущий узел C.
3. Если $X = C.Key$, то алгоритм завершён.
4. Если $X > C.Key$ и C имеет потомка справа, то делаем текущим узлом потомка справа. Переходим к п. 2.
5. Если $X < C.Key$ и C имеет потомка слева, то делаем текущим узлом потомка слева. Переходим к п. 2.
6. Ключ не найден. Конец алгоритма.

Алгоритм прост и эффективен, и его сложность определяется только наибольшей высотой дерева.

Наивное построение бинарных деревьев поиска по последовательности ключей заключается в том, что первый поступивший элемент последовательности формирует корневой узел дерева — и каждый последующий элемент занимает соответствующее место после неудачного поиска (если поиск удачен, то дерево уже содержит ключ). Неудачный поиск всегда заканчивается на каком-то узле, у которого в нужном направлении нет потомка.

Алгоритм вставки элемента элемента с ключом X в `BST`.

1. Делаем текущий узел корневым.
2. Переходим в текущий узел C.
3. Если $X = C.Key$, то алгоритм завершён, вставка невозможна.
4. Если $X > C.Key$ и C имеет потомка справа, то делаем текущим узлом потомка справа. Переходим к п. 2.

5. Если $X > C.Key$ и C потомка справа не имеет, то создаём правого потомка с ключом X и завершаем алгоритм.
6. Если $X < C.Key$ и C имеет потомка слева, то делаем текущим узлом потомка слева. Переходим к п. 2.
7. Если $X < C.Key$ и C потомка слева не имеет, то создаём левого потомка с ключом X и завершаем алгоритм.

Дерево на рис. 5.90 — неплохое дерево.

А вот это дерево — отвратительное. Поиск в нём будет требовать $O(N)$.

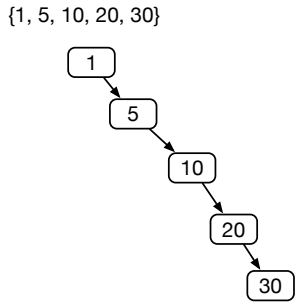


Рис. 5.91. BST: вырожденное дерево

Интересно узнать, насколько часто будут встречаться такие деревья при случайном поступлении элементов из последовательности?

Определение 25. ***Случайное бинарное дерево*** *T размера n — дерево, получающееся из пустого бинарного дерева поиска после добавления в него n узлов с различными ключами в случайном порядке, при условии, что все $n!$ возможных последовательностей добавления равновероятны.*

Определим его среднюю глубину.

Пусть $\bar{d}(N+1)$ — средняя глубина всех узлов случайного дерева с $N+1$ узлами. Как оно получилось? Вначале был добавлен какой-то узел, пусть он имеет значение k. Так как как все события равновероятны, то вероятность того, что первым был добавлен именно узел k есть $p_k = \dfrac{1}{N+1}$.

И слева и справа от этого узла есть поддеревья, возможно нулевой высоты. Они начнутся с высоты 1. В левом поддереве окажутся элементы $\{0, \ldots, k-1\}$, в правом — $\{k+1, \ldots, N\}$.

Средняя высота левого поддерева равна $\bar{d}(k)$, правого — $\bar{d}(N-k)$. Математическое ожидание высоты случайного дерева, следовательно, будет равно

$$\bar{d}(N+1) = \sum_{k=0}^{N} \frac{1}{N+1} \left(1 + \frac{k}{N} \cdot \bar{d}(k) + \frac{N-k}{N} \cdot \bar{d}(N-k)\right);$$

$$\bar{d}(N+1) = \frac{2}{N(N+1)} \sum_{k=0}^{N} k \cdot \bar{d}(k).$$

Используя предел

$$\lim_{n\to\infty} \left(\sum_{k=1}^{n} \frac{1}{k} - \ln n\right) = \gamma = 0.57721...,$$

получаем

$$\lim_{N\to\infty} (\bar{d}(N) - 2\ln N) \to const.$$

Отсюда можно сделать вывод, что и средняя глубина узлов случайного бинарного дерева, и средние времена выполнения операций вставки, удаления и поиска в случайном бинарном дереве есть $\Theta(\log_2 N)$.

5.2.1 Свойства бинарного дерева поиска

Так как все потомки слева от узла имеют значения ключей, меньшие значения ключа самого узла, то наименьший элемент всегда находится в самом низу левого поддерева. Аналогично, наибольший элемент всегда находится в самом низу правого поддерева.

```
tree * minNode(tree *t) {
  if (t == nullptr) return nullptr;
  while (t->left != nullptr) {
    t = t->left;
  }
  return t;
}
```

Методы поиска и вставки, описанные ранее, достаточно просты.

Процедура удаления — сложнее, требуется рассмотреть три случая:

1. У удаляемого узла нет потомков — достаточно удалить этот узел у родителя.
2. Имеется один потомок — переставляем узел у родителя на потомка.
3. Имеется два потомка — находим самый левый лист в правом поддереве и им замещаем удаляемый.

Первый случай. Удаление терминального узла 6: до удаления.

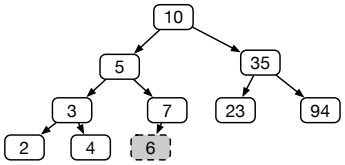


Рис. 5.92. BST: удаляем терминальный узел

Первый случай после удаления. Уничтожается сам узел и связь до него из родителя.

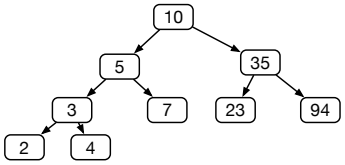


Рис. 5.93. BST: удаляем терминальный узел. Итог

Второй случай. Удаление узла 7, имеющего одного потомка: до удаления.

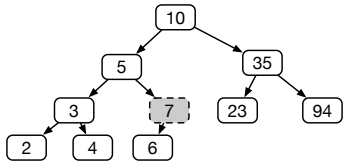


Рис. 5.94. BST: удаление узла с одним потомком

Второй случай после удаления. Единственный потомок узла занял его место.

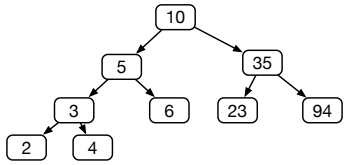


Рис. 5.95. BST: удаление узла с одним потомком. Итог

Третий случай. Удаление узла 10, имеющего двух потомков.

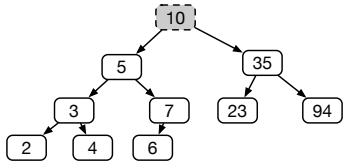


Рис. 5.96. BST: Удаление узла с двумя потомками

Третий случай после удаления. Самый левый потомок узла 35 занял место удаляемого.

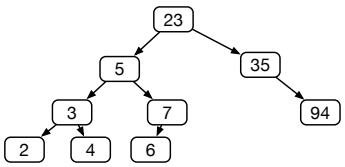


Рис. 5.97. BST: Удаление узла с двумя потомками. Итог

Структура хранилища	вставка	удаление	поиск
Бинарное дерево поиска (наихудшее)	$O(N)$	$O(N)$	$O(N)$
Бинарное дерево поиска (среднее)	$O(\log N)$	$O(\log N)$	$O(\log N)$

5.3 Борьба с дисбалансом

Сложность всех алгоритмов в бинарных деревьях поиска определяется средневзвешенной глубиной. Операции вставки/удаления могут привести к дисбалансу и ухудшению средних показателей. Для борьбы с дисбалансом применяют *рандомизацию* и *балансировку*.

Как мы уже видели, создание `BST` из упорядоченной последовательности чревато крайне несбалансированным деревом. Возникает вопрос: а что будет, если вставлять новые элементы не в терминальные узлы дерева, а заменять ими корень?

Последствия таковы: если вставляемый элемент больше корня, то старый корень сделаем левым поддеревом, а его правое поддерево — нашим правым поддеревом. Аналогично рассуждаем для случая, когда вставляемый элемент меньше корня. К сожалению, и в том, и в другом случае может нарушиться упорядоченность. Чтобы нарушений не происходило, требуется сохранять инвариант упорядоченности. Для этого введём понятие *поворота*, не изменяющего упорядоченные свойства дерева, но меняющего его структуру, то есть высоту поддеревьев.

В следующих функциях мы пользуемся такой возможностью языка `C++`, как ссылка на указатель. Это позволяет нам сильно упростить весь код.

Пусть дерево перед поворотом выглядит так:

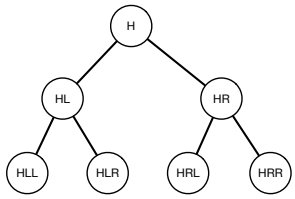


Рис. 5.98. BST: дерево перед поворотом

Обозначения достаточно прозрачны: `H` есть сокращение от слова `Head`, `L` — от `Left` и `R` — от `Right`.

После поворота направо дерево будет выглядеть так (рис. 5.99):

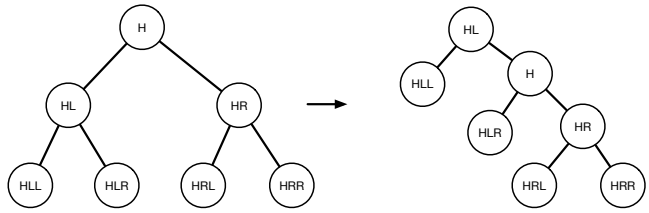


Рис. 5.99. BST: дерево после правого поворота

После поворота налево дерево будет выглядеть так (рис. 5.100):

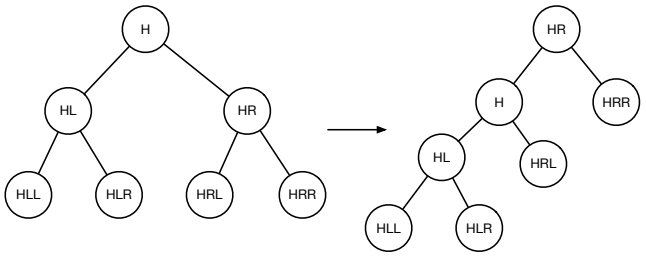


Рис. 5.100. BST: дерево после левого поворота

Обратите внимание на то, что упорядоченность узлов сохранена, что меньшие значения ключей всё ещё находятся в левых поддеревьях, а большие — в правых.

При реализации кода, связанного с деревьями, рекомендуется всегда рисовать где-нибудь на листке бумаги состояния до и после операции. После такой подготовки требуемый код прост.

```
void rotateRight(node* &head) {
  node *temp = head->left;
  head->left = temp->right;
  temp->right = head;
  head = temp;
}

void rotateLeft(node* &head) {
  node *temp = head->right;
  head->right = temp->left;
  temp->left = head;
  head = temp;
}
```

Каждая из функций поворота изменяет свой аргумент значением нового корня поддерева.

Функция вставки нового узла в корень дерева изящна и рекурсивна. Самостоятельно разберитесь, как она работает.

```
void insert(node* &head, item x) {
  if (head == nullptr) {
```

```
    head = new node(x);
    return;
  }
  if (x.key < head->item->key) {
    insert(head->left, x);
    rotateRight(head);
  } else {
    insert(head->right, x);
    rotateLeft(head);
  }
}
```

5.4 Рандомизированное дерево

Итак, у нас есть операции вставки нового ключа как в виде терминального узла, так и в корень дерева. Мы знаем, что вставка узла в терминальный узел приводит к вырожденному дереву для упорядоченной последовательности. Впрочем, если попробуем эту же последовательность вставлять в корень, то получим подобный результат. Ключ к решению задачи — добавление здравого количества случайностей в процедуру вставки. Будем случайным образом выбирать, куда мы собираемся вставить очередной ключ. Операция вставки в корень дерева значительно сложнее операции вставки в терминальный узел, так как она требует $O(\log N)$ операций поворота. Случайность здесь заключается в том, что для дерева, содержащего N вершин, мы бросаем кубик, содержащий $N + 1$ грань, и вставляем в корень только при условии, что выпала единица. Другими словами, вставка очередного узла в корень производится с вероятностью $\frac{1}{N+1}$, в противном случае вставку оставляем обыкновенной, то есть в узел. В этом случае свойства любого дерева будут соответствовать свойствам случайного дерева.

5.5 Декартовы деревья

Случайные бинарные деревья поиска близки к идеальным по сложности ($H = O(\log N)$).

Можно внести ещё более серьёзный элемент случайности, добавив второй ключ, генерируемый случайно.

Определение 26. ***Декартово дерево*** *(treap) есть комбинация бинарного дерева поиска (BST) и бинарной кучи (BH). При поиске информации декартово дерево — BST. Вставка и упорядочивание узлов происходит по отношениям BH.*

При вставке в BST можно получить определяемое комбинаторикой количество различных деревьев, содержащих те же самые элементы. При вставке в BST с вторичным упорядочиванием по отношениям BH получается единственное дерево со свойствами случайного BST.

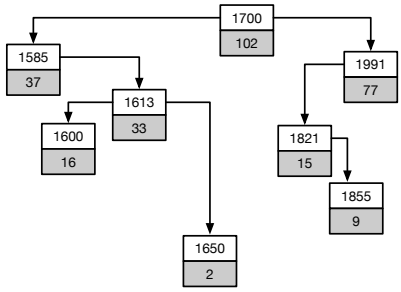


Рис. 5.101. Декартово дерево: пример

5.5.1 Операции над декартовыми деревьями

Операция *find*

Декартово дерево есть BST. Поиск происходит по всем правилам обычного BST. $T(N) = O(\log N)$.

Операция *insert*

Декартово дерево есть BST + BH.

Первичная вставка проводится в BST. При этом могут быть нарушены свойства BH. Если вставленный элемент не нарушает свойства (BH), то вставка завершена. Если свойства BH нарушаются, проводится вращение, поднимающее вставленный элемент. Подъём происходит до тех пор, пока нарушены свойства (BH) .

Здесь и далее мы будем применять выражение для какого-то узла «вращение в сторону родителя». Если искомый узел — левый подузел у родителя, то вращение в сторону родителя есть правый поворот с корнем в родителе. Если искомый узел — правый подузел у родителя, то вращение в сторону родителя есть левый поворот с корнем в родителе.

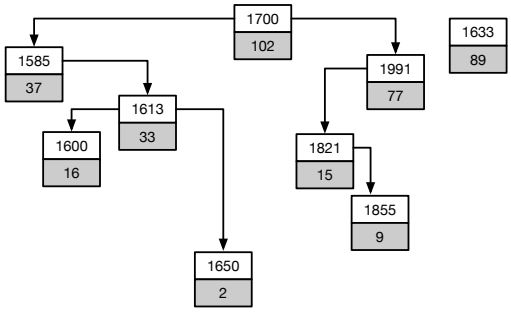


Рис. 5.102. Декартово дерево: вставка элемента `(1633,89)`

Элемент вставлен по правилам `BST`, но не упорядочен по правилам `BH`.

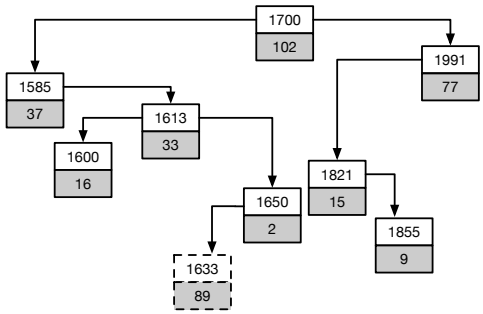


Рис. 5.103. Декартово дерево: вставка по правилам `BST`. Нарушено свойство `BH`

Попытка обмена с родителем по правилам `BH` нарушает свойства `BST`. Как оказалось, сохранить инвариант `BST` сложнее, чем инвариант `BH` (рис. 5.104).

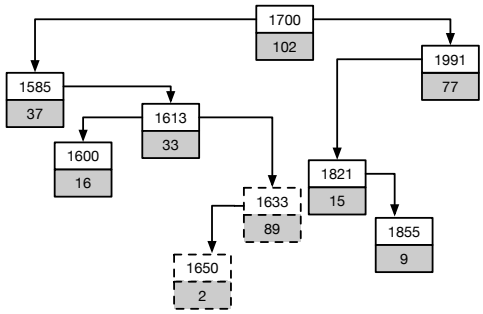


Рис. 5.104. Декартово дерево: попытка обмена по правилам `BH` неудачна

Вращение в сторону родителя не нарушает свойства BST, но свойство BH ещё нарушено (рис. 5.105).

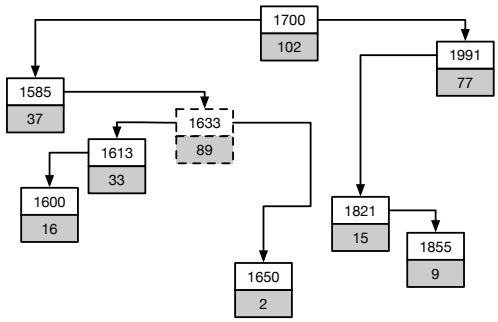


Рис. 5.105. Декартово дерево: вставка. Вращение по правилам BST сохраняет инвариант BST

Ещё одно вращение в сторону родителя — и все свойства восстановлены (рис. 5.106).

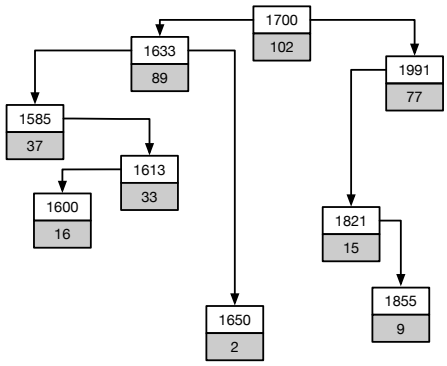


Рис. 5.106. Декартово дерево: вставка, итог

Операция *remove*

Декартово дерево есть `BST` + `BH`.

Так как удаление узлов, отличных от вершин, нетривиально, а удаление вершин — тривиально, задача — сделать удаляемый узел терминальным. Для этого на каждом шаге вращаем удаляемый узел с его ребёнком, имеющим наибольшее значение y, до тех пор, пока он не станет терминальной вершиной. На этапе спуска мы не обращаем внимания на сохранение свойства `BH`, нас интересуют только значения y.

Попытаемся удалить корневой элемент. Элемент `(1633,89)` имеет наибольшее значение y из детей, вращаем его по направлению к родителю (рис. 5.107).

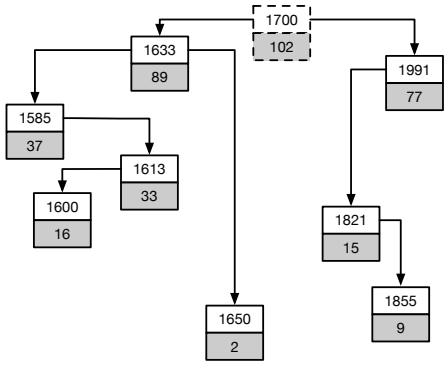


Рис. 5.107. Декартово дерево: удаление корневого элемента, вращение в сторону наибольшего значения y из детей

Теперь новый объект для вращения — узел `(1991,77)` (рис. 5.108).

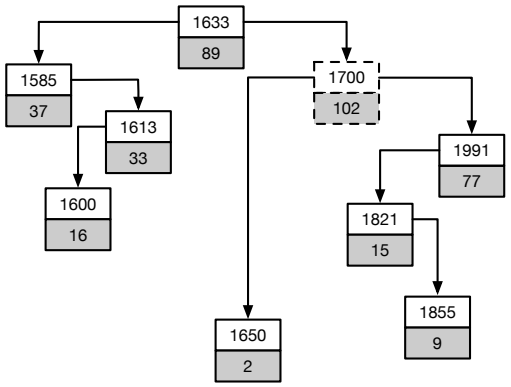


Рис. 5.108. Декартово дерево: вращение с корнем в `(1700,102)` в направлении `(1991,77)`

Следующее направление — узел `(1821,15)` (рис. 5.109).

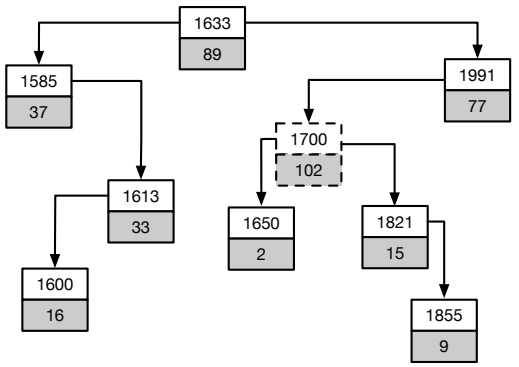


Рис. 5.109. Декартово дерево: вращение с корнем в `(1700,102)` в направлении `(1821,15)`

Последнее направление — узел `(1650,2)` (рис. 5.110).

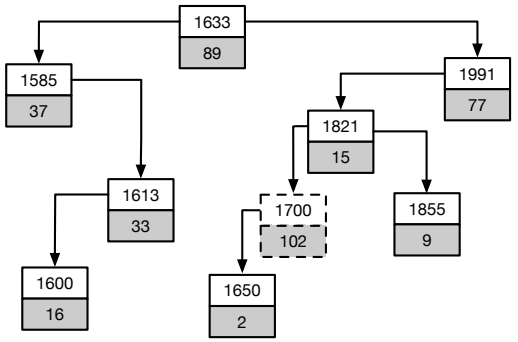


Рис. 5.110. Декартово дерево: вращение с корнем в `(1700,102)` в направлении `(1650,2)`

Удаляемый узел добрался до вершин и может быть удалён (рис. 5.111).

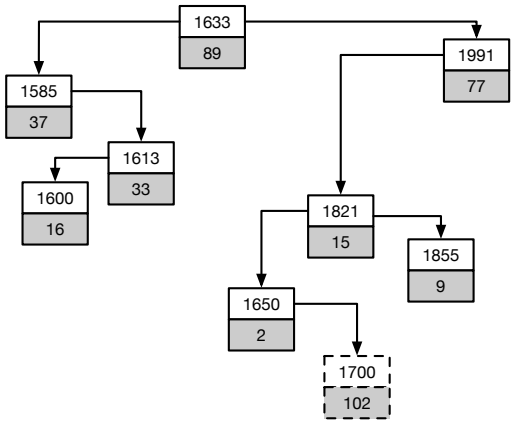


Рис. 5.111. Декартово дерево: терминальный узел удаляется без проблем

Заключительное состояние показано на рис. 5.112).

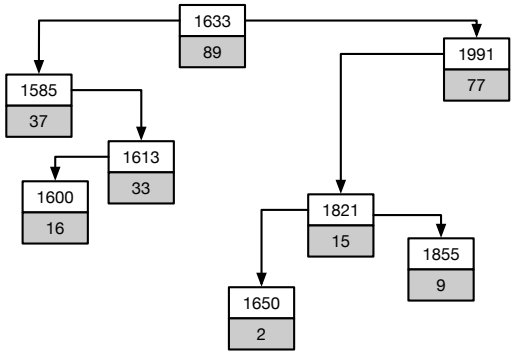


Рис. 5.112. Декартово дерево: удаление. Итог

Декартовы деревья, конечно, имеют гораздо более широкий спектр применений, чем здесь рассмотрено. Наша задача была и познакомиться с этой замечательной структурой данных, и научиться использовать её для реализации деревьев поиска.

5.6 Сбалансированные деревья поиска

Добиться хороших оценок времени исполнения операций с `BST` можно и без использования случайности. Для этого требуется при операциях избегать тех преобразований структуры деревьев, которые приводят к их вырождению. Это можно сделать, измеряя высоты поддеревьев, и, делая повороты при необходимых условиях, *балансировать* деревья.

Поставим более жёсткую задачу: реализовать операции с деревьями, имеющие время в *худшем* $\Theta(\log N)$.

Для этого требуется сохранять высоту дерева H в определённых границах. Чтобы сравнить различные стратегии балансировки, будем сравнивать высоту H, выраженную формулой $H < A \cdot \log_2 N + B$, где A и B — некоторые фиксированные константы. Если обозначить через H_{ideal} высоту идеально сбалансированного дерева, а через H_{algo} — наибольшую из возможных высот при реализации выбранной стратегии балансировки, то в

первую очередь нас будет интересовать константа $A = \dfrac{H_{algo}}{H_{ideal}}$ — отношение этих высот.

Рассмотрим несколько критериев сбалансированности и вычислим для них коэффициенты A и B.

Сбалансированное дерево №1.

Для любого узла количество узлов в левом и правом поддереве N_l, N_r отличается не более, чем на 1. $N_r \leqslant N_l + 1, \quad N_l \leqslant N_r + 1$. Это — идеально сбалансированное дерево.

Пусть $H_{ideal}(N)$ — максимальная высота идеально сбалансированного дерева.

Если N — нечётно и равно $2M + 1$, тогда левое и правое поддеревья должны содержать ровно по M вершин.

$$H_{ideal}(2M + 1) = 1 + H_{ideal}(M).$$

Если N — чётно и равно $2M$. Тогда

$$H_{ideal}(2M) = 1 + \max(H_{ideal}(M - 1), H_{ideal}(M)).$$

Так как $H_{ideal}(M)$ — неубывающая функция, то

$$\begin{aligned} H_{ideal}(2M) &= 1 + H_{ideal}(M), \\ H_{ideal}(N) &\leqslant \log_2 N. \end{aligned}$$

Это означает, что ключевой коэффициент A равен 1.

Сбалансированное дерево №2. Для любого узла количество подузлов в левом и правом поддеревьях удовлетворяют условиям $N_r \leqslant 2N_l + 1, \quad N_l \leqslant 2N_r + 1$.

Примерная сбалансированность количества узлов. Пусть $H(M)$ — максимальная высота сбалансированного дерева со свойством 2. Тогда $H(1) = 0, H(2) = H(3) = 1$.

При добавлении узла один из узлов будет корнем, остальные $N - 1$ распределятся в отношении $N_l : N_r$, где $N_l + N_r = N - 1$.

Не умаляя общности, предположим, что $N_r \geqslant N_l$, тогда $N_r \leqslant 2N_l + 1$.

$$H(N) = \max_{N_l, N_r} (1 + \max(H(N_l), H(N_r))).$$

Функция $H(N)$ — неубывающая, поэтому

$$H(N) = 1 + H(\max(N_r, N_l)).$$

При ограничениях $N_r \leqslant 2N_l + 1$ и $N_l + N_r = N + 1$ получаем

$$H(N) = 1 + H\left(\left\lfloor \frac{2N-1}{3} \right\rfloor\right),$$
$$H(N) \geqslant 1 + H\left(\left\lfloor \frac{2N}{3} \right\rfloor\right),$$
$$H(N) \geqslant \log_{3/2} N + 1 \approx 1.71 \log_2 N + 1.$$

Сбалансированное дерево №3.

Для любого узла высоты левого и правого поддеревьев H_l, H_r удовлетворяют условиям $H_r \leqslant H_l + 1, H_l \leqslant H_r + 1$.

Это — примерная сбалансированность высот. Название этих деревьев — АВЛ-деревья — взято из первых буквы фамилий их изобретателей: Георгия Максимовича Адельсона-Вельского и Евгения Михайловича Ландиса.

Пусть $N(H)$ — минимальное число узлов в АВЛ-дереве с высотой H (минимальное АВЛ-дерево), и левое дерево имеет высоту $H - 1$. Тогда правое дерево будет иметь высоту $H-1$ или $H-2$. Так как $N(H)$ — неубывающая, то для минимального АВЛ-дерева высота правого равна $H - 2$.

Число узлов в минимальном АВЛ-дереве: $N(H) = 1 + N(H-1) + N(H-2)$.

$$\lim_{h\to\infty} \frac{N(h+1)}{N(h)} = \Phi = \frac{\sqrt{5}+1}{2},$$
$$H(N) \approx \log_\Phi (N-1) + 1 \approx 1.44 \log_2 N + 1.$$

Сбалансированное дерево №4. Красно-чёрное дерево (`RBT`) — это сбалансированное бинарное дерево поиска, которое в качестве критерия балансировки использует цвет узлов.

1. Вершины разделены на **красные** и **чёрные**.
2. Каждая вершина хранит поля **ключ** и **значение**.
3. Каждая вершина имеет указатель *left*, *right*, *parent*.
4. Отсутствующие указатели помечаются указателями на фиктивный узел *nil*.
5. Каждый лист *nil* — чёрный.
6. Если вершина — красная, то её потомки — чёрные.

7. Все пути от корня *root* к листьям содержат одинаковое число чёрных вершин. Это число называется **чёрной высотой дерева**, **black height**, $bh(root)$.

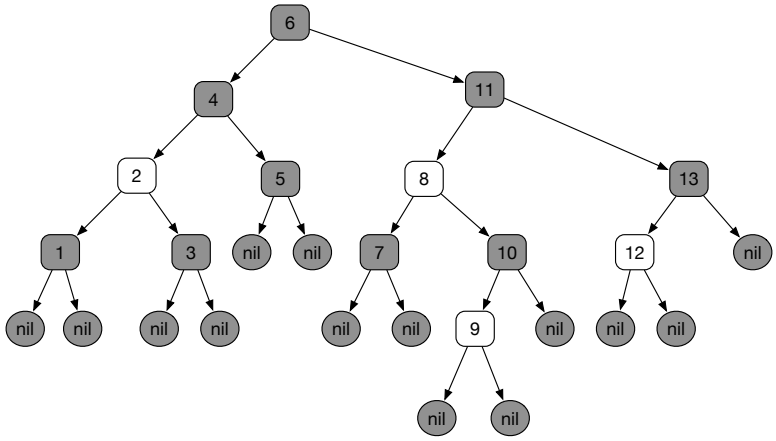


Рис. 5.113. Красно-чёрное дерево, пример. Красные узлы на рисунке изображены отсутствием заливки

Найдём коэффициенты A и B для формулы высоты этого дерева в зависимости от числа узлов.

Для листьев чёрная высота равна нулю, $bh(x) = 0$. Обозначим через $|T_x|$ количество узлов в дереве с чёрной высотой, равной x.

Докажем, что $|T_x| \geqslant 2^{bh(x)}$.

- База индукции: Пусть вершина x является листом. Тогда $bh(x) = 0$ и $|T(x)| = 0 < 2^{bh(x)}$.
- Пусть вершина x не является листом и $bh(x) = k$. Тогда для обоих потомков $bh(l) \geqslant k-1$, $bh(r) \geqslant k-1$, т. к. красный будет иметь высоту k, чёрный — $k-1$.
- Чёрная высота левого и правого поддеревьев отличается от чёрной высоты x не более, чем на единицу и они обе не меньше $k-1$, то есть, $l, r \geqslant k-1$. По предположению индукции $|T_l|, |T_r| \geqslant 2^{k-1}$.
- Общее количество узлов в дереве с вершиной x есть сумма количества

вершин поддеревьев и самого узла, следовательно, $|T_k| = |T_l| + |T_r| + 1 \geqslant 2^k$.

По свойству (6) не менее половины узлов составляют чёрные вершины, $bh(t) \geqslant H_{rb}/2$. Отсюда $N \geqslant 2^{H_{rb}/2} - 1$, и значит:

$$H_{rb} \leqslant 2 \cdot \log_2 N + 1.$$

На семинарах нам предстоит реализовать АВЛ-дерево и сравнить его эффективность со стандартными средствами библиотеки шаблонов `STL` языка `C++`, которая использует для представления абстракций `set` и `map` именно красно-чёрное дерево.

	RB-tree	AVL-tree
Средняя высота	до 1.38H	H
Поиск/вставка	до 1.38t	t
Поворотов при вставке	до 2	до 1
Поворотов при удалении	до 3	до $\log N$
Дополнительная память	1 бит	1 счётчик

5.7 Списки с пропусками

Все рассмотренные нами деревья хорошо подходят для реализации `CRUD`, то есть с их помощью можно создавать отображения и множества со сложностью операций поиска/вставки/замены $O(\log N)$. Современные процессоры в большинстве своём имеют несколько вычислительных ядер — и неплохо бы уметь использовать несколько ядер для ускорения работы программ. Предположим, что мы такую программу написали, и каждый из вычислительных потоков использует один и тот же справочник, содержащий общие данные. Иногда какой-то из потоков в этот справочник заносит новую информацию, иногда информация по ключу меняется. В момент изменения информации к структуре данных из других потоков обращаться нельзя — представьте себе, что обращение происходит в момент поворота дерева!

Классический способ множественного обращения к одной структуре данных — создание критической секции. Теперь при обращении к общим данным поток запрашивает вход в критическую секцию, и, если доступ разрешён, устанавливается замок на последующие входы в критическую секцию. После того, как поток завершил изменения структуры данных, замок снимается — и другие потоки получают возможность обращения к общим данным. Ну а пока один поток захватил замок, другие в это время переходят в состояние ожидания. Коэффициент полезного действия системы

уменьшается. Алгоритмы, использующие критические секции, называются *блокирующими*.

К счастью, имеется возможность увеличить КПД системы, применяя *неблокирующие* алгоритмы. Такие алгоритмы используют *атомарные* операции процессора. Существуют машинные команды типа *Compare-And-Swap*, которые позволяют атомарно обменять две ячейки памяти, возможно, содержащие указатели. При вставке в односвязный список достаточно атомарных операций для замены цепочки указателей. Односвязный список — идеальная структура данных для параллельных неблокирующих операций.

Однако, как мы уже выяснили, почти все операции с односвязными списками имеют сложность $O(N)$. Проблема в том, что при неудачной операции сравнения ключей указатель в списке перемещается ровно на один элемент. Если бы можно было при перемещении пропустить несколько элементов, то количество операций сократилось бы и сложность уменьшилась.

Рассмотрим следующую структуру, которая представляет собой несколько списков, связанных друг с другом тоже списками. Каждый следующий список примерно в два раза короче предыдущего и пропускает примерно половину элементов предыдущего. Каждый элемент, кроме элементов самого нижнего уровня, содержит два указателя на элементы — переход на следующий уровень и переход на следующий элемент в списке. На каждом уровне список упорядочен по значению ключа (рис. 5.114).

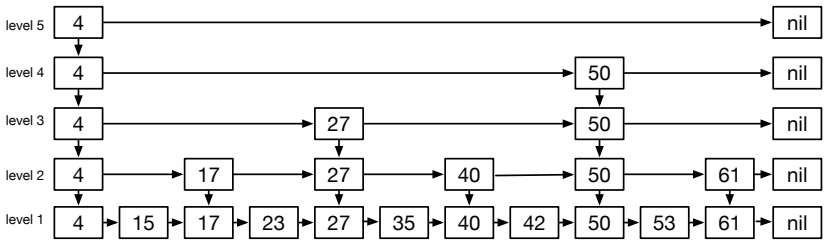


Рис. 5.114. Списки с пропусками. Пример

Поиск существующего элемента прост и начинается с самого верхнего левого элемента. Для каждого элемента имеется ключ — и всегда можно установить значение ключа, следующего по списку. Эти два значения клю-

ча определяют границы поиска ключа. Если искомый ключ находится в границах — происходит переход на уровень ниже, если нет — происходит переход внутри уровня. На следующем рисунке штриховыми линиями показаны значения границ при каждой проверке, пунктирными — маршрут поиска (рис. 5.115).

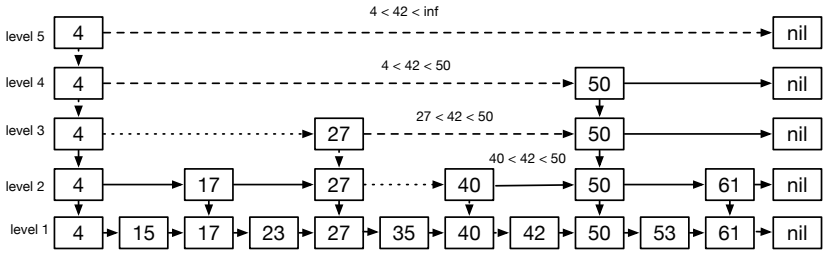


Рис. 5.115. Списки с пропусками: поиск элемента

Если ключа в структуре данных не существует, то последний поиск завершается с определением узла, после которого этот ключ мог бы быть вставлен.

А вот вставка элемента весьма интересна. После завершения неудачного поиска мы создаём узел для вставки — и атомарным образом вставляем его после требуемого узла. Нетрудно убедиться, что в этот момент времени другие потоки могут использовать список с пропусками для поиска информации без блокировки.

Но ведь если вставка будет производиться только в нижний слой, вся выгода списков с пропусками пропадёт? Снова на помощь приходит рандомизация. Каждый раз при вставке узла на какой-то уровень мы кидаем монетку. С вероятностью $\frac{1}{2}$ мы ничего не вставляем — и на этом операция завершается. В противном случае вставляем узел на вышележащий уровень и делаем его текущим. Операция вставки на уровень выше продолжается, пока позволяет монетка. При необходимости создаются новые вышележащие уровни (рис. 5.116).

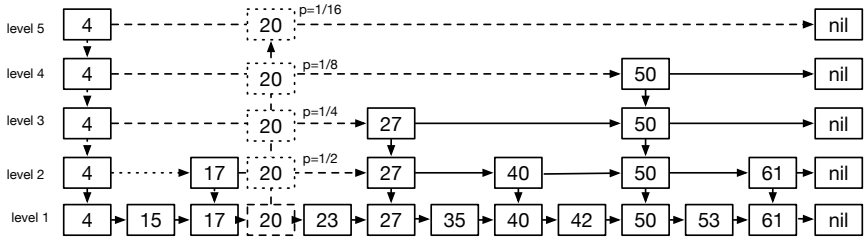


Рис. 5.116. Списки с пропусками: вставка элемента. Бросание монетки

Удаление элемента тоже состоит из двух этапов. Вначале ищется удаляемый элемент и помечается столбец, его содержащий.

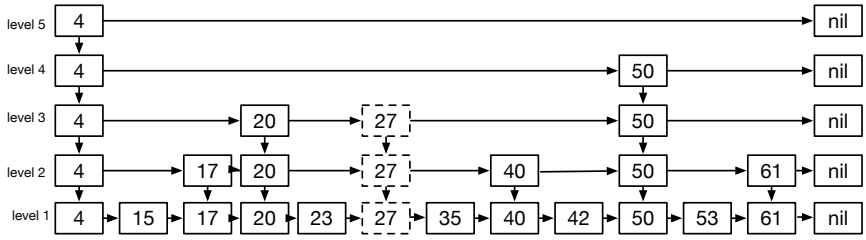


Рис. 5.117. Списки с пропусками: удаление элемента. Первый этап

Затем снизу вверх элемент уделяется из каждой содержащей его строки.

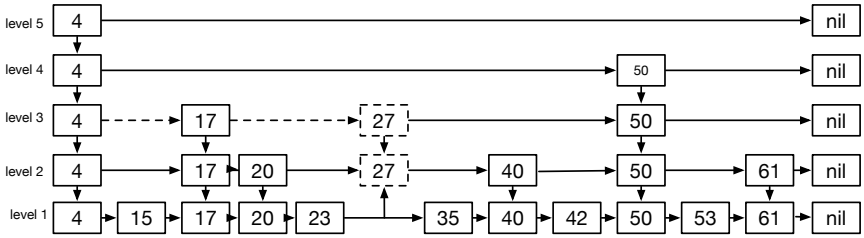


Рис. 5.118. Списки с пропусками: удаление элемента. Очистка списков

Итоговая структура данных не содержит удалённый элемент ни в одной из строк и ни в одном столбце.

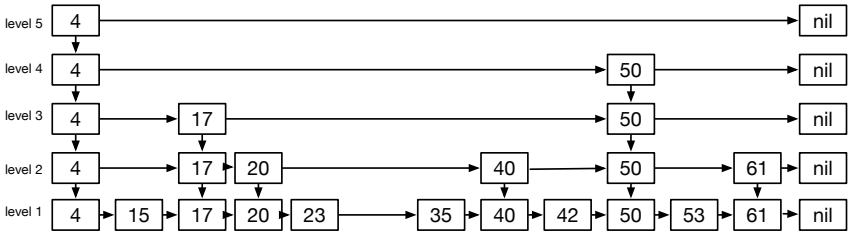


Рис. 5.119. Списки с пропусками: удаление элемента. Итог

При поиске, вставке и удалении элементов применяются различные хитрые уловки, которые помогают совместно работающим вычислительным потокам использовать минимальное количество операций, но это уже тема другой дисциплины — многопоточного программирования систем с общей памятью.

Амортизационная сложность списков с пропусками:
- Вставка — $T(N) = O(\log N)$.
- Поиск — $T(N) = O(\log N)$.
- Удаление — $T(N) = O(\log N)$.

5.8 Внешний поиск. B-деревья

Современные компьютеры в качестве носителей информации в основном используют два типа устройств — накопители, использующие магнитные пластины, `HDD`, и накопители, использующие флэш-память, `SSD`.

На `HDD` информация располагается в *секторах*, которые логически расположены на *дорожках*. Типичный размер сектора — 512/2048/4096 байт. Информация считывается и записывается *головками чтения/записи*. Для чтения/записи информации требуется *подвести* головку чтения записи к нужной дорожке и дождаться подхода нужного сектора. Типичные скорости вращения `HDD` — 5400/7200/10033/15000 оборотов в минуту. Один оборот совершается за время от 1/90 до 1/250 секунды. Операция перехода на соседнюю дорожку составляет примерно 1/1000 секунды.

Внешние сортировки используют многократный последовательный проход по данным, расположенным на носителях информации. Скорость последовательного считывания информации с `HDD` типично 100-250 МБ/сек.

Смена позиции в файле часто требует:
- ожидания подвода головки на нужную дорожку;
- ожидания подхода нужного сектора к головкам чтения/записи.

Операция последовательного чтения 4096 байт занимает $\frac{4096}{100 \times 10^6} \approx 40 \times 10^{-6}$ секунд.

Операция случайного чтения 4096 байт занимает не менее $5 - 10 \times 10^{-3}$ секунд.

На `SSD` информация хранится в энергонезависимой памяти на микросхемах. Операции производятся блоками размером 64-1024 КБ. Время доступа к блоку $\approx 10^{-6}$ секунд. `HDD` и `SSD` используют *буферизацию* для ускорения работы.

Алгоритмы поиска во внешней памяти должны минимизировать число обращений к внешней памяти.

На логическом уровне обращения к устройствам хранения производятся блоками любого размера, кратного 512 байт. На физическом уровне всё сложнее. Размер физического блока — от 64 до 1024 КБ. При операции частичной записи 512 байт:

1. Считывается полный блок (всегда).
2. Заменяется 512 байт в требуемом месте.
3. Записывается полный блок (всегда).

При выровненной записи целого блока никаких добавочных действий не производится.

А так ли необходимы структуры данных, хранящиеся на внешних носителях? Оценим, какое количество записей можно обработать, имея 16 ГБ оперативной памяти, при использовании бинарного дерева поиска, состоящего из данных размером 64 байта, ключа размером 8 байт и указателей `left` и `right` размером 8 байт. Общий размер узла — 88 байт (не считая многочисленных накладных расходов на системные структуры данных, используемых в менеджере памяти для операций `malloc/new`). При заданных условиях можно будет обработать $\frac{16 \times 2^{30}}{88} \approx 195 \times 10^6$ узлов, что не так уж и много.

Определение 27. ***B-дерево*** *— сбалансированное дерево поиска, узлы которого хранятся во внешней памяти.*

1. *Каждый узел содержит:*
   - *количество ключей* n*, хранящихся в узле;*
   - *индикатор листа* `final`*;*
   - `n` *ключей в порядке возрастания;*
   - `n+1` *указатель на детей, если узел не корневой.*
2. *Ключи есть границы диапазонов ключей в поддеревьях.*
3. *Все листья расположены на одинаковой глубине* `h`*.*
4. *Вводится показатель* `t` *— минимальная степень дерева.*
5. *В корневом узле — от 1 до* `2t-1` *ключей.*
6. *Во внутренних узлах — минимум* `t-1` *ключей.*
7. *Во внешних узлах — максимум* `2t-1` *ключей.*
8. *Заполненный узел имеет* `2t-1` *ключ.*

Такие хранилища называют *персистентными*. Из соображений эффективности часть информации загружается в оперативную память.

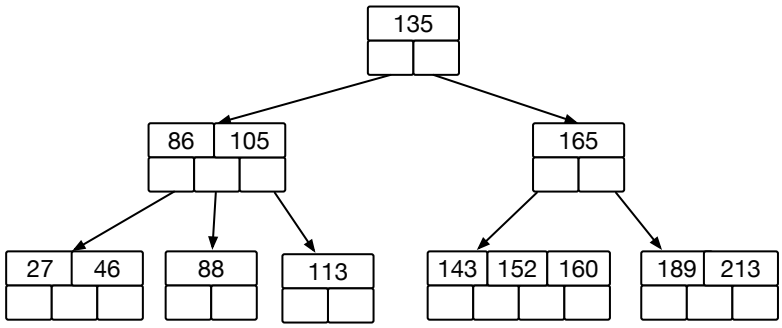


Рис. 5.120. B-дерево. Структура

Высота B-дерева с $n \geqslant 1$ ключами и минимальной степенью $t \geqslant 2$ в худшем случае не превышает $\log_t \frac{n+1}{2}$.

Для операций с внешней памятью используем абстракцию `storage` с операциями `Load` и `Store`. Корневой узел сохраняем в оперативной памяти. Наша цель — минимизировать количество операций обмена с внешней памятью. Размер блоков в операциях `Load` и `Store` можно подобрать в соответствии с физическим размером блока на носителе.

Операция `find` в `B-дереве`:

1. Операцией бинарного поиска ищем самый левый ключ $key_i \geqslant k$.
2. Если $key_i = k$, то узел найден.
3. Исполняем `Load` для дочернего узла и рекурсивно повторяем операцию.
4. Если $final = true$, то ключ не найден.

Количество операций с носителем $T_{load} = O(h) = O(\log_t N)$.

Операция вставки в `B-дерево`:

1. Операцией `find` находим узел для вставки.
2. Если лист не заполнен, сохраняя упорядоченность, вставляем ключ.
3. Если лист заполнен (содержит `2t-1` ключей), разбиваем его на два листа по `t-1` ключу поиском медианы.
4. Медиана рекурсивно вставляется в родительский узел.

Сложность в худшем случае: почти заполненный узел на каждом уровне каждый раз разбивается на два. Тем не менее, количество таких операций

не превосходит высоты дерева

Количество операций: $T_{ext} = O(h) = O(\log_t n)$.

$\mathbf{B}^+$-дерево содержит информацию только в листьях, а ключи присутствуют только во внутренних узлах. Это — очень популярная структура данных, используемая в файловых системах `XFS`, `JFS`, `NTFS`, `Btrfs`, `HFS`, `APFS` и многих других. Ещё одна область использования этой структуры данных — хранение индексов в базах данных `Oracle`, `Microsoft SQL`, `IBM DB2`, `Informix` и др.

5.9 Дерево отрезков

Последний класс деревьев, которые полезно рассмотреть — деревья отрезков.

Пусть нам надо решить задачи:

- многократное нахождение максимального значения на отрезках массива;
- многократное нахождение суммы на отрезке массива.

Мы умеем совершать эти действия за время $O(N)$, где $N = R - L + 1$.

Оказывается, что при определённой подготовке можно сократить время на каждую из операций до $O(\log N)$.

Попробуем воспользоваться бинарными деревьями.

Для примера возьмём массив `{3, 1, 4, 1, 5, 9, 2, 6}`.

Рис. 5.121. Массив — основа дерева отрезков

Попарно соединим соседние вершины, поместив в узел-родитель значение функции `max(left,right)`.

Родитель каждого узла называется *доминирующим узлом*.

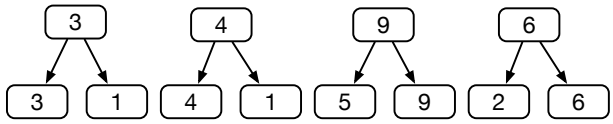


Рис. 5.122. Дерево отрезков: построение. Добавление доминирующих узлов первого уровня

Проделаем эту же операцию с получившимися узлами:

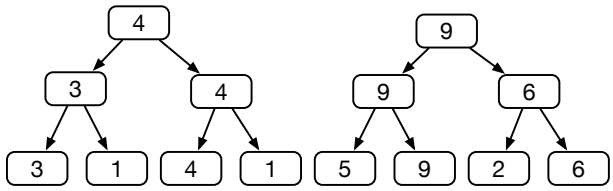


Рис. 5.123. Дерево отрезков: построение. Добавление доминирующих узлов второго уровня

Наконец:

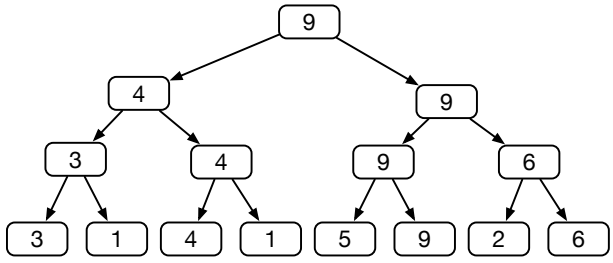


Рис. 5.124. Дерево отрезков: построение. Итог

После построения такого дерева задачу нахождения максимума на отрезке можно решить за $O(\log N)$.

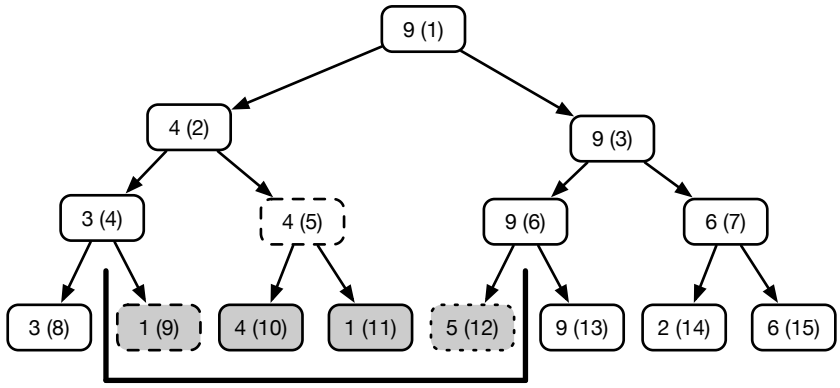


Рис. 5.125. Дерево отрезков: поиск максимума на отрезке

Нам повезло с функцией на отрезке. Во-первых, она коммутативна, то есть $\max(X, Y) = \max(Y, X)$. Во-вторых, она ассоциативна, то есть $\max(\max(X, Y), Z) = \max(X, \max(Y, Z))$. Это означает, что если, например, нам нужно вычислить $\max(x_2, x_3, x_4, x_5)$, то вычисления мы можем производить в любом порядке. Например, $\max(x_3, x_4)$ у нас уже вычислены — это значение находится в доминирующем узле (x_{34}). Ну а вычислять $\max(x_2, x_{34}, x_5)$ можно в удобном для нас порядке. Идея вычисления проста — если на отрезке присутствует пара элементов, имеющая общий доминирующий узел, то результатом вычисления функции для этой пары будет предвычисленное значение доминирующего узла. Элементы, не входящие в такие пары, могут находиться либо строго на левом конце отрезка, либо строго на правом. Что же, тогда их придётся явно учесть отдельно — и это мы сделаем, когда будем реализовывать соответствующие операции.

Нам повезло, элементов в массиве было $8 = 2^3$, поэтому после всех операций получилось полное бинарное дерево. А что будет, если количество элементов не будет степенью двойки? Недостающие элементы можно добавить до ближайшей степени двойки.

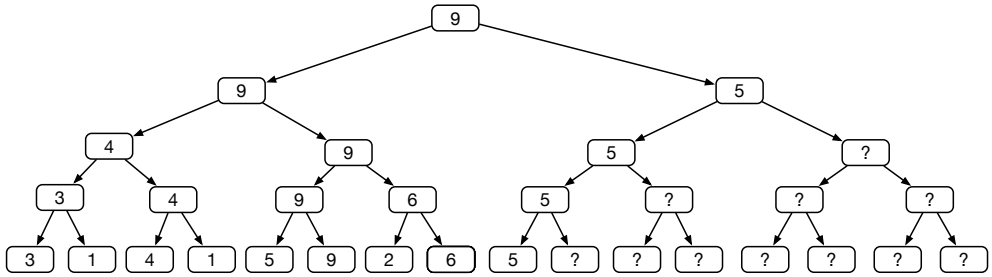


Рис. 5.126. Дерево отрезков: дополнение числа элементов до степени двойки

Что должно находиться в узлах, отмеченными знаками вопроса?

Так как все значения в доминирующих узлах вычисляются с помощью функции $P = \max(L, R)$, то то же самое должно происходить с элементом `'?'`. Это означает, что операция $\max(L, ?)$ должна возвращать L. То есть элемент `'?'` есть $-\infty$.

Для функции `max` $-\infty$ есть *нейтральный элемент*.

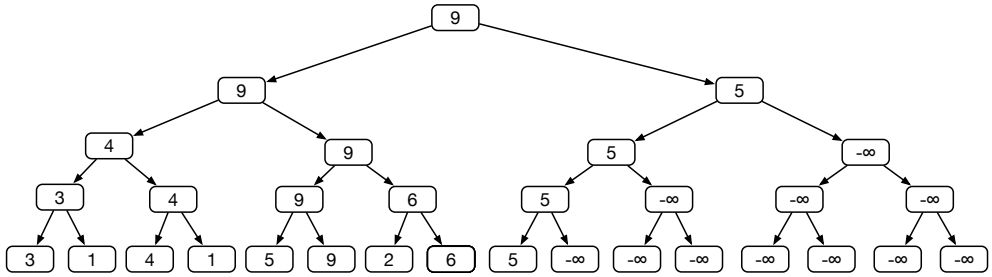


Рис. 5.127. Дерево отрезков: нейтральные элементы

Идея дерева отрезков распространяется на все такие операции (функции), в которых:

$$\begin{aligned} A \circ B = &\quad B \circ A \\ A \circ (B \circ C) = &\quad (A \circ B) \circ C \\ \exists E : A \circ E = &\quad A \end{aligned}$$

Для некоторых операций, удовлетворяющих необходимым требованиям, такие нейтральные элементы существуют.

Операция	Нейтральный элемент
`max`	$-\infty$
`min`	$+\infty$
`+`	0
`*`	1
`xor`	0

При создании дерева отрезков (***Create(size)***) создаётся бинарная куча, инициализированная нейтральными элементами. C есть номер первого элемента в нижнем ряду, представляющем заданный набор, над которым будут далее производиться операции. $C = \min(2^k) : C \geqslant size$ (рис. 5.128).

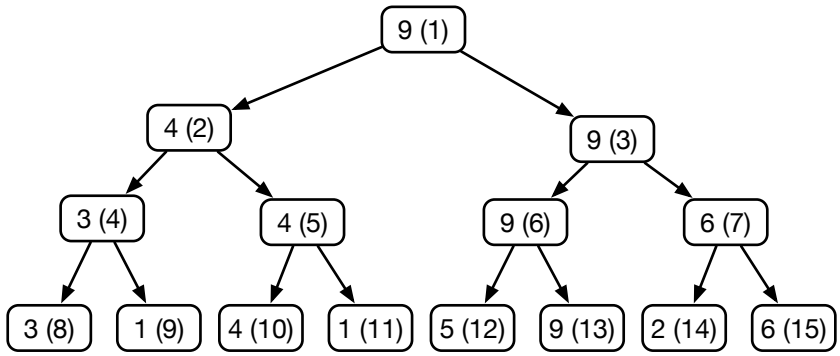


Рис. 5.128. Дерево отрезков: нумерация

Каждая операция вставки элемента по сути заменяет нейтральный элемент, который находился в месте вставки, на нужное значение. Операция

`propagate(i)` рекурсивно обновляет все доминирующие узлы.

Insert/Replace(i, val): `body[i+C]=val; propagate(i);`

Операция нахождения значения функции на интервале $[left, right]$ ***Func(left,right)*** тоже рекурсивна:

```
1. Res = E;
2. if (left % 2 == 1) Op(Res,body[left++]);
3. if (right % 2 == 0) Op(Res,body[right--]);
4. if (right > left) Op(Res,Func(left/2, right/2));
```

Операция создания дерева отрезков требует в худшем случае до $4N$ памяти, а остальные операции имеют логарифмическую сложность.

- Требуемая память: $\min = O(2N)..max = O(4N)$.
- Операция ***Insert/Replace***: $O(\log N)$.
- Операция ***Func*** на любом подотрезке: $O(\log N)$.

5.10 Домашние задания

Задача 21. Запросы сумм

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

В первой строке файла содержатся два числа: количество элементов в массиве `V`: $10 \leqslant N \leqslant 500000$ и количество запросов $1 \leqslant M \leqslant 500000$. Каждый элемент массива лежит в интервале $[0 \dots 2^{32})$.

Каждый запрос — отдельная строка, состоящая из кода запроса, который может быть равен 1 или 2, и аргументов запроса.

Запрос с кодом один содержит два аргумента, начало `L` и конец отрезка `R` массива. В ответ на этот запрос программа должна вывести значение суммы элементов массива от $V[L]$ до $V[R]$ включительно.

Запрос с кодом два содержит тоже два аргумента. Первый из них есть номер элемента массива `V`, а второй — его новое значение.

Количество выведенных строк должно совпадать с количеством запросов первого типа.

Пример

стандартный ввод	стандартный вывод
10 8 1 7 15 8 9 15 15 19 5 19 1 1 8 1 6 8 1 0 6 2 6 6 2 1 6 2 0 9 1 4 7 1 3 6	93 39 70 49 38

Задача 22. Поиск множеств

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	64 мегабайта

В первой строке файла содержится три числа: `N` — количество эталонных множеств, `M` — размер каждого из множеств и `K` — количество пробных множеств.

Каждое из множеств содержит целые числа от 0 до 10^9, числа могут повторяться.

Требуется для каждого из пробных множеств вывести в отдельной строке цифру `'1'`, если это множество в точности совпадает с каким-либо из эталонных множеств, и цифру `'0'`, если оно не совпадает ни с одним, то есть выведено должно быть в точности `K` строк.

$5 \leqslant N \leqslant 50000$
$3 \leqslant M \leqslant 1000$
$5 \leqslant K \leqslant 50000$

Примеры

стандартный ввод	стандартный вывод
10 3 5 6 5 1 7 9 3 2 3 2 7 2 9 9 6 2 6 6 6 9 4 1 8 4 4 8 3 2 1 2 6 9 7 2 1 6 5 3 7 7 4 4 6 3 9 7	1 1 0 0 1
10 7 5 8 4 0 3 6 9 2 3 5 0 4 3 1 1 7 1 0 3 1 2 4 7 1 5 1 5 5 1 3 4 0 0 3 4 0 3 3 3 6 3 9 3 3 4 1 3 1 8 1 1 1 6 8 6 8 2 5 6 8 1 3 9 3 7 5 7 1 4 0 3 1 1 3 3 8 4 1 2 1 1 6 6 8 8 1 1 5 7 5 1 5 3 4 1 3 1 1 8 0 0 1 2 8 2 6	1 1 1 1 0

Задача 23. Телефонная книга

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Необходимо разработать программу, которая является промежуточным звеном в реализации телефонной книги. На вход подаётся $N \leqslant 1000$ команд вида

ADD User Number

DELETE User

EDITPHONE User Number

PRINT User

Согласно этим командам, нужно соответственно добавить пользователя в телефонную книгу, удалить пользователя, изменить его номер и вывести на экран его данные. В случае невозможности выполнить действие необходимо вывести **ERROR**. Добавлять пользователя, уже существующего в телефонной книге, нельзя.

Необходимо вывести протокол работы телефонной книги

Формат входных данных

См. пример.

Формат выходных данных

В случае невозможности выполнения действия требуется вывести **ERROR**

В случае команды **PRINT** User

User Number

Примеры

стандартный ввод	стандартный вывод
9 ADD IVAN 1178927 PRINT PETER ADD EGOR 123412 PRINT IVAN EDITPHONE IVAN 112358 PRINT IVAN PRINT EGOR DELETE EGOR EDITPHONE EGOR 123456	ERROR IVAN 1178927 IVAN 112358 EGOR 123412 ERROR

Задача 24. Анаграммы

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	0.5 секунд
Ограничение по памяти:	256 мегабайт

Как известно, анаграммами называются слова, которые могут получиться друг из друга путём перестановки букв, например `LOOP`, `POOL`, `POLO`. Будем называть все слова такого рода *комплектом*.

На вход программы подаётся число слов $1 \leqslant N \leqslant 100000$. В каждой из очередных N строк присутствует одно слово, состоящее из заглавных букв латинского алфавита. Все слова имеют одинаковую длину $3 \leqslant L \leqslant 10000$.

Требуется определить число комплектов во входном множестве.

Формат входных данных

```
N
W1
W2
...
WN
```

Формат выходных данных

```
NumberOfComplects
```

Пример

стандартный ввод	стандартный вывод
8 BCB ABA BCB BAA BBC CCB CBC CBC	3

Задача 25. Кеширующий сервер

Имя входного файла:	стандартный ввод
Имя выходного файла:	стандартный вывод
Ограничение по времени:	3 секунды
Ограничение по памяти:	64 мегабайта

Дормидонт работает в компании, которая занимается обработкой больших данных. Обрабатываемые данные находятся где-то в распределённой системе. Количество различных данных в системе ограничено — и каждые данные имеют свой номер. Эти данные регулярно требуются различным клиентам и, поскольку время обращения к ним достаточно велико, для ускорения обработки информации Дормидонту поручено написать часть middlware — сервер-посредник, к которому и обращаются теперь клиенты за данными. Так как система — распределённая, а сервер — нет, все требуемые данные на сервер не помещаются. Но он имеет возможность запоминать результаты своих запросов к распределённой системе. Для этого на сервере выделена ограниченная память на N запросов. Важно, что клиент не имеет возможности обращаться к распределённой системе — и результаты запроса к распределённой системе всегда должны оказываться на сервере.

К большой радости Дормидонта оказалось, что самые крупные и значимые клиенты всегда обращаются за одними и теми же данными в одном и том же порядке, так что у него есть последовательность запросов. Дормидонт придумал такой алгоритм, что как можно большее количество запросов исполняется из кеша сервера, без обращения к распределённой системе. Придумаете ли вы что-то подобное?

Формат входных данных

На вход программы подаётся размер памяти под кеширование запросов $1 \leqslant N \leqslant 100000$, количество запросов $1 \leqslant M \leqslant 100000$ и ровно M запросов с номерами $0 \leqslant R_i \leqslant 10^{18}$. Количество различных номеров запросов ограничено и не превосходит 100000.

Формат выходных данных

Требуется вывести одно число: сколько раз пришлось обратиться к распределённой системе за данными, отсутствующими в кеше. В начале работы кеш пуст.

Примеры

стандартный ввод	стандартный вывод
5 15 3 1 4 1 5 9 2 6 5 3 5 8 7 9 3	9

Замечание

В приведённом примере первые три запроса произойдут к данным под номерами 3, 1 и 4, так как их нет в кеше. Следующий запрос, 1, есть в кеше — и обращения к распределённой системе не произойдёт. Запросы 5 и 9 занесут их в кеш. Следующий запрос — 2 — в кеше отсутствует, но мы выкинем из кеша запрос 1, и запрос 2 займёт его место. Далее, запрос 6 вытеснит из кеша значение 2 (у нас есть информация о дальнейших запросах — и из неё мы видим, что запрос под номером 2 больше не повторится и нет причин хранить его далее), после чего следующие три запроса удовлетворятся из кеша. Затем произойдёт ещё два вытеснения — 8 и 7. Итого: 9 обращений к распределённой системе. Нетрудно установить, что меньше сделать нельзя.

Лекция 6

6.1 Обобщённый быстрый поиск

Структуры данных, которые мы изучили в предыдущей лекции, имеют логарифмическую сложность всех основных операций. Аппетит приходит во время еды, и мы хотим уменьшить эту сложность. Двоичный поиск — это хорошо и достаточно быстро, но можно ли искать быстрее? Если посмотрим на толстый бумажный словарь и попытаемся найти там что-нибудь, то не найдём ли мы там подсказку? В некоторых словарях на торце книги имеются отметки для букв. Последуем их примеру.

Для более быстрого поиска в словаре, содержащем названия городов и их население, можно разбить общее хранилище на 33 более маленьких, по одному связному списку на букву (рис. 6.129).

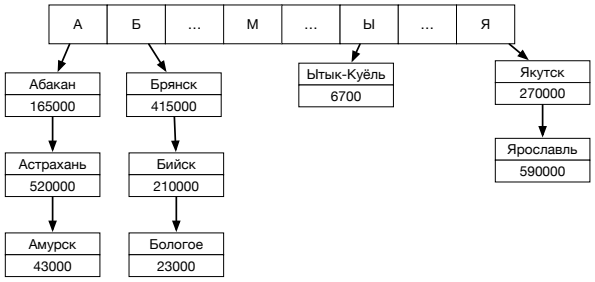


Рис. 6.129. Быстрый поиск: разбиение множества ключей на подмножества в виде связных списков

Собственно говоря, почему мы обязаны использовать именно связные списки, почему, например, не уже изученные нами деревья? Конечно, можно взять и их (рис. 6.130).

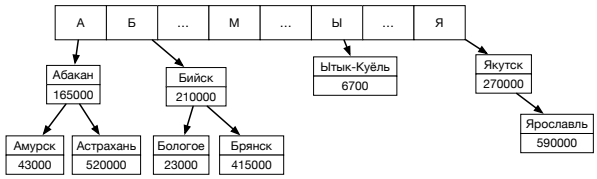


Рис. 6.130. Быстрый поиск: разбиение множества ключей на подмножества в виде деревьев

Итак, здесь — 33 сбалансированных дерева.

Очевидно, что и в том, и в другом случае поиск потребует меньшую сложность. Но такое прямолинейное разбиение не вполне хорошо — на мягкий и твёрдый знак названий нет, для некоторых букв названий совсем мало, некоторые буквы очень популярны.

Основная идея действительно быстрого поиска — разбиение пространства ключей на независимые подпространства (`partitioning`). При независимом разбиении на M подпространств сложность поиска уменьшается.

Для разбиения множества N ключей на примерно равные M подмножеств сложность вычисляется по главной теореме о рекурсии при числе подзадач M, коэффициенте размножения 1 и консолидации $O(1)$.

$$C \cdot O(N) \to \frac{C}{M} O(N),$$

$$C \cdot O(N \log N) \to \frac{C}{M} O(N \log N).$$

При увеличении M время поиска уменьшается

$$\lim_{K \to \infty} T(N, M) = O(1),$$

а требуемая память увеличивается

$$\lim_{K \to \infty} Mem(N, M) = \infty.$$

При $M \approx N$ имеется зона оптимальности — поиск уже будет проводиться за $O(1)$, а вот потребная память ещё не столь велика — $O(N)$.

Примитивное разбиение пространства ключей по первым буквам — не очень хороший вариант. Хотелось бы иметь детерминированный способ разбиения пространства ключей на M независимых подпространств. Условие разбиения — мощность множеств ключей, принадлежащих каждому подпространству, должна быть примерно равна.

$$|K_1| \approx |K_2| \approx \cdots \approx |K_M|;$$

$$\sum_{i=1}^{M} |K_i| = |K|.$$

Эврика! Создаём функцию $H(K)$, удовлетворяющую некоторым условиям.

6.2 Хеш-функции

Определение 28. ***Хеш-функция*** *есть функция преобразования множество ключей K на множество V мощностью M.*

$$H(K) \to V,$$

$$|D(V)| = M.$$

Введём понятие *соперника*, того, кто предоставляет нам ключи. Цель *соперника* — предоставлять ключи таким образом, чтобы значения функции оказались не равновероятными. *Соперник* знает хеш-функцию и может выбирать ключи.

Чтобы быть независимыми от соперника — и для удобства практического применения, хотелось бы для функции $H(K)$ обеспечить следующие свойства:

- **Эффективность.**
$$T(H(K)) \leqslant O(L(K)),$$
где $L(K)$ — мера длины ключа K. Время вычисления хеш-функции не должно быть велико.
- **Равномерность.** Каждое выходное значение равновероятно.
$$p_{H(K_1)} = p_{H(K_2)} = \cdots = p_{H(K_M)}.$$

- **Лавинность.** При незначительном изменении входной последовательности выходное значение должно меняться значительно, иначе соперник может просто подобрать ключ.
- Для борьбы с *соперником* — **необратимость**, то есть невозможность восстановления ключа по значению его функции.

Следствия из требуемых свойств:

- Функция не должна быть близка к непрерывной. Неплохо было бы, если бы для близких значений аргумента получались сильно различающиеся результаты.
- В значениях функции не должно образовываться *кластеров*, множеств близко стоящих точек.

Примеры плохих функций:

- $H = K^2 \mod 10000$ для $K < 100$.
  Функция монотонно возрастает. Пространство значений ключа слишком велико — и часть значений недостижима.
- $H = \sum_{i=0}^{s.size()-1} s[i]$ для строки s.
  Функция даёт одинаковые значения для строк `abcd` и `abdc` и отличающиеся на единицу — для строк `abcd` и `abde`. Сопернику легко найти ключи, которые дают равные значения функции.

Совпадение значений функции для разных значений ключа называется **коллизией**. Большое количество коллизий для данного множества ключей — плохая хеш-функция. Конечно, без коллизий обойтись не удастся, но, если оно больше $\frac{1}{|D(M)|}$, с хеш-функцией какие-то проблемы.

Введём H^* — множество хеш-функций, которые отображают пространство ключей в $m = |D(M)|$ различных значений.

Определение 29. *Множество хеш-функций* ***универсально****, если для каждой пары ключей* $K_i, K_j, i \neq j$ *количество хеш-функций, для которых* $H^*(K_i) = H^*(K_j)$ *не более* $\dfrac{|H^*|}{m}$.

При наличии такого универсального множества борьба с соперником закончится нашей победой, если мы каждый раз будем выбирать случайным образом функцию из этого множества. Ведь если случайным образом выбирается функция из множества H^*, то для случайной пары ключей $K_i, K_j, i \neq j$ вероятность коллизии не должна превышать $\frac{1}{m}$.

А как создать такое универсальное множество? В книге [4] приведена следующая теорема с доказательством:

Пусть множество $Z_p = \{0, 1, \ldots, p-1\}$, *множество* $Z_p^* = \{1, 2, \ldots, p-1\}$, p — *простое число,* $a \in Z_p^*$, $b \in Z_p$. *Тогда множество*

$$H^*(p, m) = \{H(a, b, K) = ((aK + b) \mod p) \mod m\}$$

есть универсальное множество хеш-функций.

Обратим внимание на то, что для хорошей универсальной функции множество Z_p есть множество неотрицательных целых чисел, что не всегда совпадает с нашими запросами: часто требуется получить значение хеш-функции (часто говорят: *получить хеш*) от строк, объектов и прочих сущностей. Вполне достаточно рассматривать значение ключа как последовательность битов, только трактовать эту последовательность придётся как целое число соответствующей разрядности, то есть перейти к операциям над (N)-числами. Для ключей с большой длиной это будет весьма неэффективно. Поэтому на практике часто применяют специальные, не универсальные хеш-функции.

Для строки можно использовать вариант полиномиальной хеш-функции:

$$h = \sum_{i=0}^{n} s_i \times q^i \pmod{HASHSIZE}.$$

Как и все полиномы, её быстрее вычислить по схеме Горнера, причём для хеширования обычно не имеет значения, в каком порядке мы нумеруем коэффициенты полинома — слева направо или справа налево.

```
unsigned
hash_sum(string s, unsigned q, unsigned HASHSIZE)
{
  unsigned sum = 0;
  for (size_t i = 0; i < s.size(); i++) {
    sum = sum * q + s[i];
  }
  return sum % HASHSIZE;
}
```

В книге [5] предлагается хеш-функция, основанная на идеях из теории генерации псевдослучайных чисел.

```
unsigned
hash_sedgewick(string s, unsigned HASHSIZE)
{
  unsigned h, i, a = 31415, b = 27183;
```

```
  for (h = 0, i = 0; i < s.size();
  i++, a = a * b % (HASHSIZE-1)) {
    h = (a * h + s[i]) % HASHSIZE;
  }
  return h;
}
```

Лучшие по статистическим показателям функции — хеш-функции, применяемые в криптографии. К сожалению, у них есть и свои недостатки: у них длинный код и они достаточно медленные.

Одна из хороших и быстрых функций основана на полях Галуа. Это функция `CRC`, очень популярная в архиваторах для контроля целостности данных (есть варианты с различным количеством бит). Она использует специальную таблицу `_table[256]` и её код очень прост:

```
uint32 crc32(uchar *ptr, unsigned length)
{
  uint32 c = 0xFFFFFFFF;
  while (length) {
    c ^= (uint32) (ptr[0]);
    c = (c >> 8) ^ _table[c & 0xFF];
    ptr++;
    length--;
  }
  return c ^ 0xFFFFFFFF;
}
```

Саму таблицу `_table` можно создать функцией генерации таблицы — или иметь уже готовую.

6.2.1 Исследование различных хеш-функций

Насколько плохие хеш-функции плохи? Насколько хорошие функции хороши? Посмотрим картинки. Каждая из картинок даёт распределение относительной встречаемости значения хеш-функции от значения аргумента. В качестве аргументов были взяты названия идентификаторов, встречающихся в одном комплексе программ примерно в 2 миллиона строк на `C++` и `C#`. Идеальная картинка — закрашенный прямоугольник высотой 1.

Первый «подопытный кролик» — функция `hash_sum` для `q=8`. `HASHSIZE` принято за 400.

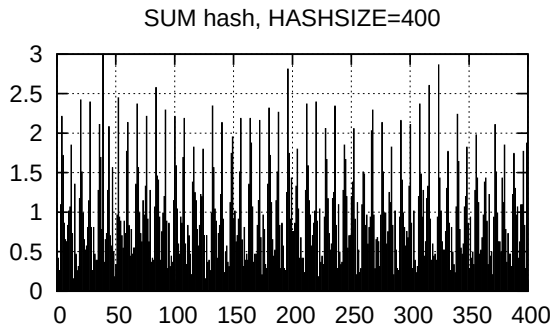


Рис. 6.131. Распределение значений хеш-функции `SUM` для `HASHSIZE=400`

Нельзя сказать, что получилось что-то приличное. Попробуем поменять `HASHSIZE` на 401.

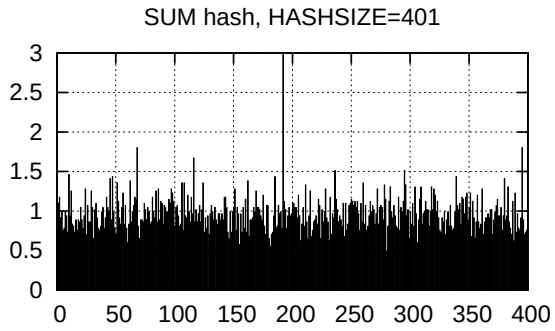


Рис. 6.132. Распределение значений хеш-функции `SUM` для `HASHSIZE=401`

Результат серьёзно улучшился. А что там за пик в районе 190? Не приведёт ли он к проблемам в дельнейшем? Может и привести.

Этот же набор ключей для функции `hash_sedgewick` и `HASHSIZE=400`.

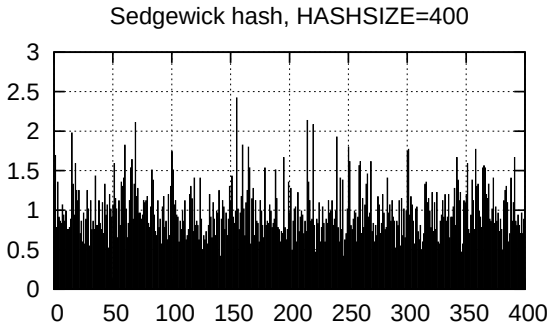


Рис. 6.133. Распределение значений хеш-функции SEDGEWICK для HASHSIZE=400

Неплохо, но что будет, если поменять значение HASHSIZE на 401?

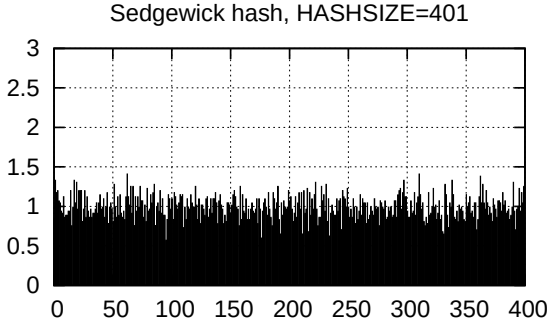


Рис. 6.134. Распределение значений хеш-функции SEDGEWICK для HASHSIZE=401

О, намного лучше! Можно сказать, что график близок к идеалу. Попробуем hash_src для HASHSIZE, равном 400.

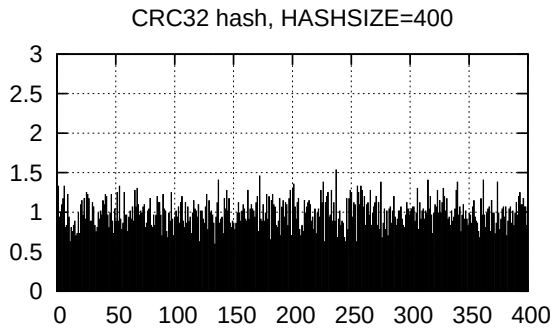


Рис. 6.135. Распределение значений хеш-функции `CRC32` для `HASHSIZE=400`

Интересно, что результат тоже близок к идеальному. А если изменить `HASHSIZE` на 401?

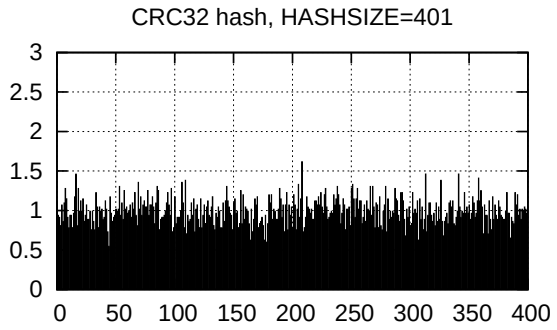


Рис. 6.136. Распределение значений хеш-функции `CRC32` для `HASHSIZE=401`

Всё тоже очень хорошо.

Мы видим, что на результат влияет размер множества значений хеш-функции (`HASHSIZE`). Для простого числа 401 результат почти всегда лучше,

чем для составного 400. Сама хеш-функция тоже очень важна.

Вот небольшая таблица, в которой приведены затраты времени на исполнение программы исполнения хеш-функций для упомянутого набора идентификаторов на одном из ноутбуков.

Алгоритм/набор	include.txt	source.txt
hash_sum	890	786
hash_sedgewick	2873	2312
hash_crc32	912	801

Функция `hash_sedgewick` оказалась самой медленной, так как для каждого символа входной строки потребовалось исполнять операцию нахождения остатка по модулю. А эта операция — одна из самых медленных на современных компьютерах.

6.3 Применение хеш-функций

6.3.1 Вероятностный подход к надёжности

Надёжны ли современные вычислительные системы? Неужели они не отказывают совсем? Увы, сбои при обработке информации происходят, их вероятность мала, но всё же отлична от нуля.

Производитель серверной памяти с коррекцией ошибок `IBM` измерил, что за три года произошло 6 отказов на 10000 серверов, каждый из которых был укомплектован памятью размером 4Гб. Интенсивность отказов — один отказ на 10^{20} обработанных байт.

Какова вероятность отказа при сравнении двух блоков памяти в 4096 байт? Вероятность получения неверного ответа при их равенстве есть $\frac{4096}{10^{20}} \approx 2.5 \cdot 10^{-16}$.

Если мы возьмём хорошую хеш-функцию, выдающую равномерно значения из множества в 2^{64} элементов, то вероятность совпадения значений этой хеш-функции для двух блоков данных размером в 4096 байт — $\frac{4096}{2^{64}} = 2^{-52} \approx 10^{-17.1}$, то есть меньше! Для хеш-функции, выдающей значения из множества 2^{128} элементов, вероятность коллизии будет порядка 10^{-35}, что несравнимо меньше вероятности аппаратного отказа. Таким образом, с вероятностью, максимально близкой к достоверности, можно сказать, что если для двух блоков данных хорошая «длинная» хеш-функция дала одно и то же значение, то эти блоки равны. В дальнейшем мы будем пользоваться этим фактом.

6.3.2 Вероятностные множества

Определение 30. ***Вероятностное множество*** — *структура данных, реализующая функциональность абстракции «множество», имеющая операции* `insert` *и* `find` *с отсутствием гарантии точности результата поиска в этом множестве. Результаты поиска могут быть ложноположительными, если элемент отсутствует, но операция* `find` *вернула истину. Отсутствие элемента всегда определяется точно, то есть ложноотрицательных результатов быть не может.*

6.3.3 Фильтр Блума

Фильтр Блума — один из вариантов реализации вероятностных множеств. В основе его представления лежит битовый массив из m бит, и для его функционирования требуется n различных хеш-функций $h_1, \dots h_n$, равномерно отображающих входные ключи на номера битов (от 0 $m-1$).

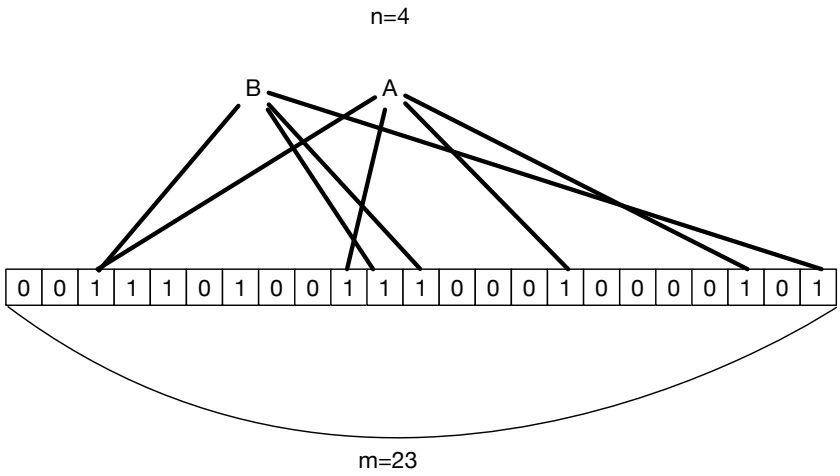


Рис. 6.137. Фильтр Блума: внесены ключи `A` и `B`

На рисунке — фильтр Блума, состоящий из 23 элементов ($m = 23$) и 4-х хеш-функций ($n = 4$). Для ключа `A` значения хеш-функций равны {2, 9, 15, 20}, а для ключа `B` — {2, 10, 11, 22}.

Операция создания нового фильтра Блума **create** обнуляет все m битов.

Операция **insert(key)**: вычисляются все n хеш-функций от **key** — и устанавливаются соответствующие биты в массиве.

При операции **find** вычисляются все n хеш-функций. Если хотя бы один бит в массиве не присутствует, то мы точно знаем, что такого элемента **НЕТ** — если бы он был, все биты, соответствующие хеш-функциям, были бы установлены. А если совпали все биты, то ответ: **МОЖЕТ БЫТЬ** — вполне могло оказаться, что биты установлены другими ключами.

Например, для элементов A и C, не равных друг другу, все их хеши могут совпасть:

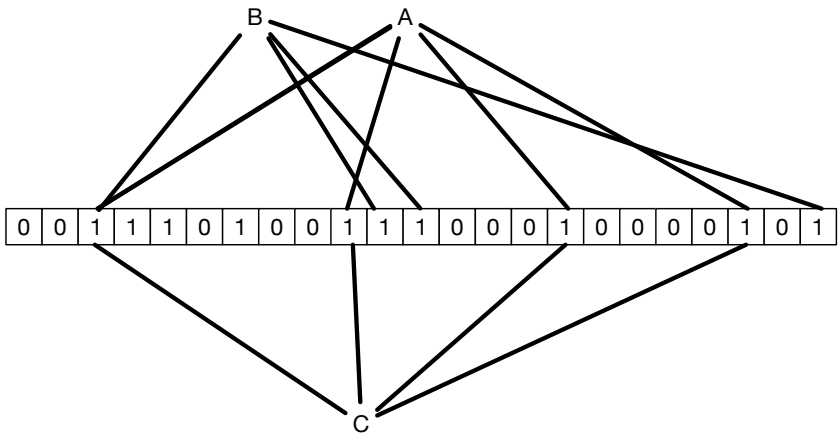


Рис. 6.138. Фильтр Блума: ключ `C` даёт ответ **МОЖЕТ БЫТЬ**

Если фильтром называть то, что помогает что-то не пропускать далее, то данный алгоритм реализует настоящий фильтр, который помогает отсеять заведомо ненужные элементы. Особенности его функционирования: при добавлении элементов количество установленных битов увеличивается — и точность алгоритма уменьшается. В предельном случае, когда все биты установлены, любой запрос даёт ответ **МОЖЕТ БЫТЬ**.

Если количество битов невелико, то фильтр быстро вырождается при увеличении количества вставленных элементов. Та же самая проблема воз-

никает, если велико количество хеш-функций. На практике оптимальное число хеш-функций для m битов и t элементов вычисляется по формуле

$$n = \frac{m}{t} \ln 2.$$

Операцию удаления тоже можно реализовать, но представление множества в виде побитового массива уже не подходит. Подойдёт представление, в котором для каждого из элементов множества имеется счётчик их количества, то есть структура данных *мультимножество*.

Применяют фильтр Блума в случаях, когда операция доступа к элементу, даже для проверки его наличия, достаточно сложная, и есть возможность быстро определить, присутствует ли элемент с таким ключом во вторичной структуре данных. Например, `Google Chrome` по имени сайта может произвести первичную проверку, не известен ли этот сайт как вредоносный. Выигрыш здесь в том, что иногда передавать несколько килобайтов фильтра существенно дешевле, чем каждый раз обращаться к базе данных на сайте.

В `Google BigTable` по заданной строке или столбцу базы данных определяется их наличие в таблице, что многократно уменьшает количество запросов к жёсткому диску за данными.

`Дедупликация` есть способ хранения данных, разбитых на блоки, при котором хранятся только уникальные данные. Каждый объект определяется номерами хранящихся блоков. Если для двух объектов какие-то блоки данных совпали, то в каждом из объектов будут одинаковые ссылки на них. Если какой-то блок изменился, то старая копия остаётся в хранилище, а ссылка теперь указывает на новый уникальный блок. Фильтр Блума позволяет быстро определить отсутствие вновь пришедшего блока данных в хранилище.

6.3.4 Алгоритм Карпа-Рабина

Хеш-функции могут использоваться и для ускорения поиска подстрок в строке.

Задача формулируется так: имеется исходная строка и образец. Определить позицию в исходной строке, содержащую образец.

Упростим задачу.

Пусть множество символов, из которых могут состоять исходная строка и образец, ограничено символами `A`, `B`, `C`, `D`.

Отобразим их в `1`, `2`, `3`, `4`. Почему мы начинаем нумерацию не с нуля? Полиномиальная хеш-функция, которую мы будем применять, не способна различить одиночный нуль или группу нулей.

Пусть строка-образец — pat=`ABAC` или `1213`.

Строка-источник — src=`ACABAACABACAABCA`

A	B	A	C
1	2	1	3

A	C	A	B	A	A	C	A	B	A	C	A	A	B	C	A
1	3	1	2	1	1	3	1	2	1	3	1	1	2	3	1

Выберем *простое* число, немного превышающее мощность алфавита $P = 5$. Составим таблицу `T` степеней числа P по модулю 2^{32}

0	1	2	3	4	5	6	7	8	9
1	5	25	125	625	3125	15625	78125	390625	1953125

10	11	12	13	14	15
9765625	48828125	244140625	1220703125	1808548329	452807053

Предлагаемая хеш-функция от строки `S` в поддиапазоне `[k...r]` вычисляется следующим образом:

$$H(S_{[k,r]}) = \sum_{i=k}^{r} S_{i-k} \cdot P^{i-k} = \sum_{i=k}^{r} S_{i-k} \cdot T_{[i-k]}.$$

Чтобы найти, имеется ли образец длиной 4 в исходной строке, сначала вычислим значение хеш-функции от образца, а затем — от всех подстрок строки `src` длиной 4:

$$H\left(pat_{[0,3]}\right) = H(ABAC) = 0 \cdot 5^0 + 1 \cdot 5^1 + 0 \cdot 5^2 + 2 \cdot 5^3 = 411.$$

Для подстрок:

$$\begin{aligned}
H\left(src_{[0,3]}\right) &= 291, & H\left(src_{[1,4]}\right) &= 183,\\
H\left(src_{[2,5]}\right) &= 161, & H\left(src_{[3,6]}\right) &= 407,\\
H\left(src_{[4,7]}\right) &= 206, & H\left(src_{[5,8]}\right) &= 291,\\
H\left(src_{[6,9]}\right) &= 183, & H\left(src_{[7,10]}\right) &= 411,\\
H\left(src_{[8,11]}\right) &= 207, & H\left(src_{[9,12]}\right) &= 166,\\
H\left(src_{[10,13]}\right) &= 283, & H\left(src_{[11,14]}\right) &= 431.
\end{aligned}$$

Хеш-функция для наших строк выглядит так:

```
unsigned hash(string s, unsigned l, unsigned r, unsigned *ptab) {
  unsigned sum = 0;
  for (unsigned i = l; i < r; i++) {
    sum += (s[i]-'A'+1) * ptab[i-l];
  }
  return sum;
}
```

Наивный поиск подстроки:

```
unsigned hs1 = hash(s1, 0, s1.size(), ptab);
for (size_t i = 0; i < s2.size() - s1.size(); i++) {
  unsigned hs2 = hash(s2, i, i+s1.size(), ptab);
  if (hs2 == hs1) {
    bool ok = true;
    for (size_t j = 0; ok && j < s1.size(); j++) {
      if (s1[j] != s2[i+j]) {
        ok = false;
        break;
      }
    }
    if (ok) {
      printf("match at: %u\n", i);
    }
  }
}
```

Примем, что `N=src.size()`, `M=pat.size()`. Сложность алгоритма тогда составит $O(NM)$, что слишком много.

Мы ведь всё делали правильно, почему же такой разочаровывающий результат? Очень просто. Мы произвели много избыточных действий.

Очевидно, что

$$H\left(s_{[0,4]}\right) = s_0 + s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3.$$

Попробуем применить индукцию и вычислить $H\left(s_{[1,4]}\right)$.

$$H\left(s_{[1,5]}\right) = s_1 + s_2 \cdot p^1 + s_3 \cdot p^2 + s_4 \cdot p^3.$$

Умножим на p^1:

$$H\left(s_{[1,5]}\right) \cdot p^1 = s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3 + s_4 \cdot p^4.$$

Сравним с

$$H\left(s_{[0,5]}\right) = s_0 + s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3 + s_4 \cdot p^4.$$

$$\boxed{H\left(s_{[k,l]}\right) \cdot p^k = H\left(s_{[0,l]}\right) - H\left(s_{[0,k]}\right).}$$

Выражение показывает, что достаточно вычислить значения хеш-функции от всех собственных префиксов строки `src`.

$$\begin{aligned}
H\left(src_{[0,0]}\right) = &\quad 0, \quad H\left(src_{[0,1]}\right) &&= 1\\
H\left(src_{[0,2]}\right) = &\quad 16, \quad H\left(src_{[0,3]}\right) &&= 41\\
H\left(src_{[0,4]}\right) = &\quad 291, \quad H\left(src_{[0,5]}\right) &&= 916\\
H\left(src_{[0,6]}\right) = &\quad 4041, \quad H\left(src_{[0,7]}\right) &&= 50916\\
H\left(src_{[0,8]}\right) = &\quad 129041, \quad H\left(src_{[0,9]}\right) &&= 910291\\
H\left(src_{[0,10]}\right) = &\quad 2863416, \quad H\left(src_{[0,11]}\right) &&= 32160291\\
&\quad \ldots
\end{aligned}$$

```
int karp_rabin(string s1, string s2, vector<unsigned> ptab) {
  unsigned hs1 = hash(s1, 0, s1.size(), ptab);
  vector<unsigned> htab(s2.size());
  for (size_t i = 1; i < s2.size(); i++)
    htab[i] = htab[i-1] + (s2[i-1] - 'A' + 1)*ptab[i-1];
  for (unsigned i = 0; i < s2.size() - s1.size(); i++) {
    unsigned hs2 = htab[i+s1.size()] - htab[i];
    if (hs2 == hs1) {
      bool ok = true;
      for (size_t j = 0; j < s1.size(); j++)
        if (s1[j] != s2[i+j]) {
          ok = false;
          break;
        }
      if (ok)
        return i;
```

```
    }
    hs1 *= 5;
  }
  return -1;
}
```

Важно, что при совпадении хеш-функций для фрагментов мы должны сравнить эти фрагменты, то есть, применение хеш-функций даёт нам подсказки, где можно было бы найти совпадающие подстроки.

Приведённый алгоритм может показаться избыточно сложным: ведь есть более простые алгоритмы Z- и префикс-функций, которые помогают быстро найти подстроку в строке. Оказывается, применённая здесь функция имеет свойство *rolling-hash*, то есть достаточно дёшево обходятся операции добавления единичных элементов строки в её конец и удаления единичных элементов из начала строки. Ни Z-, ни префикс-функции этим свойством не обладают.

6.4 Хеш-таблицы

Простая хеш-таблица есть массив пар *{ключ,значение}*. Пусть размер этого массива будет `HASHSIZE`, и хеш-функция от ключа будет давать числа в диапазоне `[0..HASHSIZE)`. Тогда применение хеш-функции к ключу даст нам индекс в этом массиве, который поможет нам найти пару *{ключ,значение}*. Каким образом мы будем искать эту пару далее, зависит от организации хеш-таблицы.

В простейшем варианте мы можем сразу найти необходимую пару. Например, для приведённой таблицы 6.139, если `Hash("Брянск")=5` и в 5-м элементе таблицы находится пара с ключом `"Брянск"`, то поиск завершился успехом после первой же попытки.

Часто в таблице находятся не сами пары *{ключ,значение}*, а указатели на них, или на голову связного списка, содержащего пары. `NULL` в элементе таблицы означает, что элемент пуст (таб. 6.140).

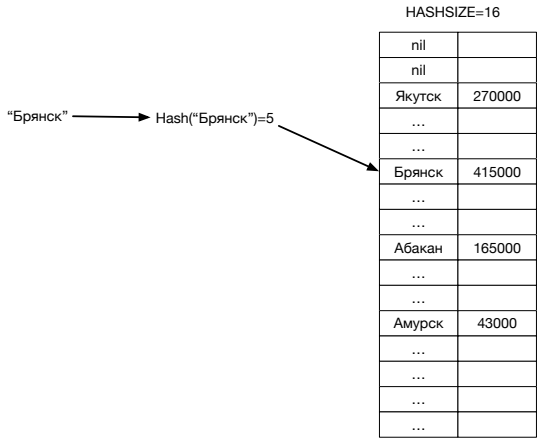


Рис. 6.139. Хеш-таблица: пример

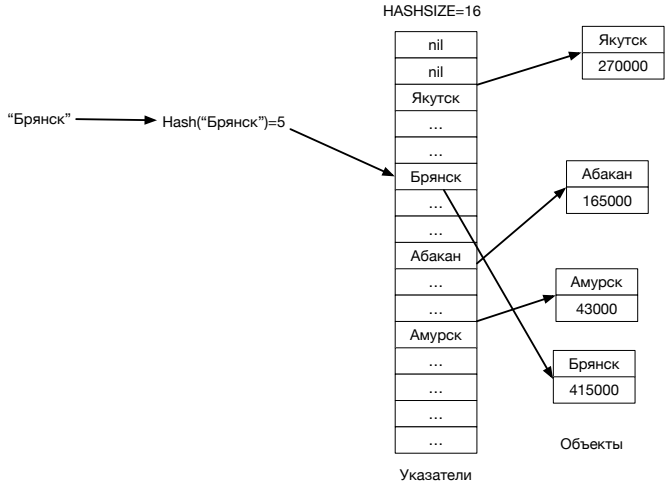


Рис. 6.140. Хеш-таблица с хранением указателей на пары ключ/значение

Введём несколько терминов. Если известны количество элементов в контейнере C и размер массива M, то $\alpha = \frac{C}{M}$ — *коэффициент заполнения*, *fill-factor* или *load-factor*. α — главный показатель хеш-таблицы.

Операция `create` для хеш-таблицы часто имеет аргумент, равный начальному количеству элементов массива. Массив заполняется либо `nullptr`, либо парами с невозможным значением ключей — нам всегда необходимо знать, свободен ли *слот* для размещения пары.

Добавление элементов (операция `insert`) требует поиска (операции `find`).

Если после нахождения значения хеш-функции от вновь прибывшего ключа оказывается, что запись с таким значением хеш-функции уже есть (например, `Hash("Якутск") = 2` и `Hash("Мышкин") = 2`), то говорят, что произошла *коллизия*. Коллизий хотелось бы избежать, так как без них операции поиска и вставки имели бы сложность $O(1)$.

Для борьбы с коллизиями есть несколько способов организации хеш-таблиц. Это — *прямая* (или *закрытая*) адресация или *открытая* адресация. При выборе стратегии с открытой адресацией можно использовать *рехеширование*, об этом чуть попозже.

6.4.1 Хеш-таблицы с прямой адресацией

При коллизии во время создания элемента создаётся связный список конфликтующих. Технически можно создать любую поисковую структуру данных. Обычно стараются, чтобы количество элементов во вторичной структуре данных оставалось небольшим, поэтому простой односвязный список — хороший выбор (рис. 6.141).

Операции поиска и вставки

1. При поиске вычисляется значение хеш-функции от ключа.
2. По этому значению определяется место поиска — вторичная поисковая структура данных.
3. Если вторичной структуры нет, то нет и элемента, который мы ищем. Теперь, если это требуется, то создаётся вторичная структура данных и элемент вставляется в неё.
4. Иначе элемент ищется во вторичной структуре и вставляется туда при необходимости.

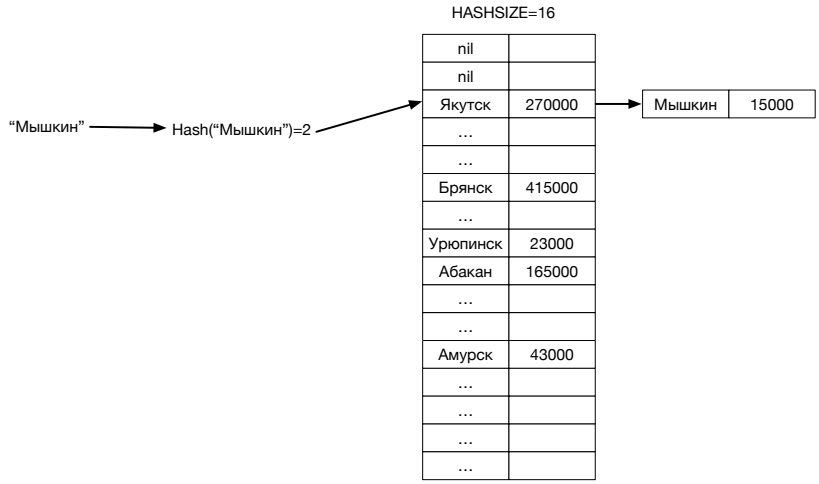


Рис. 6.141. Хеш-таблица с прямой адресацией: имеются вторичные структуры поиска

Операция удаления

1. При удалении вычисляется хеш-функция от ключа.
2. Определяется место нахождения ключа — вторичная поисковая структуре данных.
3. Если вторичной структуры нет, то нет и элемента.
4. Иначе элемент удаляется из вторичной структуры.
5. Если вторичная структура пуста, удаляется сама структура и точка входа в неё.

6.4.2 Хеш-таблицы с открытой адресацией

При другой организации хеш-таблиц вторичные структуры данных не используются — и все пары ключ/значение хранятся в самой таблице. Это означает, что требуется удобный способ разрешения коллизий.

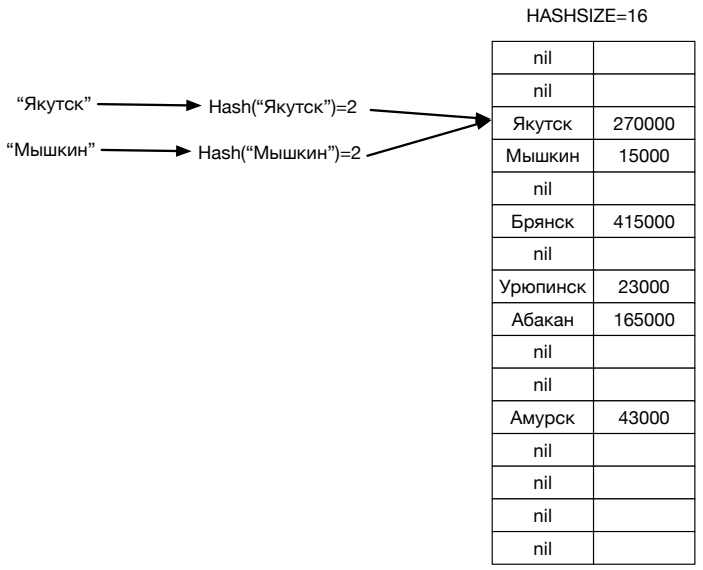


Рис. 6.142. Хеш-таблицы с открытой адресацией: пары ключ/значение хранятся в самой таблице

Процедуры поиска по ключу и вставки здесь немного различаются. Дело в том, что требуется понимать, что делать с удалённым элементом.

Операция поиска по ключу

1. При поиске существующего элемента вычисляется хеш-функция от его ключа.
2. По значению хеш-функции определяется место поиска — индекс в хеш-таблице.
3. Если по индексу ничего нет, то нет и элемента, алгоритм завершён.
4. Иначе по индексу находится элемент с нашим ключом (требуется операция сравнения ключей)— элемент найден.
5. Если по индексу находится элемент с другим ключом или элемент помечен удалённым, увеличиваем индекс на единицу (возвращаясь в начало таблицы при необходимости) и переходим к пункту 3.

6. Следующий индекс вычисляется по формуле $(index + 1) \mod M$.

Операция вставки по ключу

1. При вставке нового элемента вычисляется хеш-функция.
2. По значению хеш-функции определяется место поиска — индекс в хеш-таблице.
3. Если по индексу находится пустой элемент или имеется элемент, помеченный как удалённый, то мы нашли подходящее место — вставляем по индексу элемент.
4. Если по индексу уже присутствует элемент с искомым ключом — не трогая ключа, меняем данные и выходим.
5. Если по индексу элемент с другим ключом, то индекс увеличиваем на единицу и переходим к пункту 3.
6. Следующий индекс вычисляется по формуле $(index + 1) \mod M$.

Рис 6.143 показывает, почему мы требуем свойства равномерности от хеш-функции: малейшая неравномерность при генерации значений приводит к большим *кластерам* коллизий.

Операция удаления

1. При удалении вычисляется хеш-функция от ключа.
2. Определяется место поиска — индекс в хеш-таблице.
3. Если по индексу ничего нет, то нет и элемента.
4. Иначе, если по индексу расположен элемент с требуемым ключом, то элемент найден. Помечаем его удалённым и заканчиваем алгоритм.
5. Если по индексу расположен элемент с другим ключом, индекс увеличивается на единицу — и мы переходим к пункту 3.
6. Следующий индекс вычисляется по формуле $(index + 1) \mod M$.

Когда *fill-factor* начинает превосходить 0.7-0.8, отношение количества сравнения ключей к количеству производимых запросов становится слишком велико, поэтому, какой бы начальный размер таблицы мы ни выбрали, при регулярных операциях вставки возникает момент, когда таблицу необходимо расширить.

Расширение хеш-таблицы:

- Создаётся другой массив пустых элементов либо указателей на них нужного размера.
- Из оригинального массива в порядке увеличения индексов извлекаются элементы и вставляются в новый массив (таблицу).

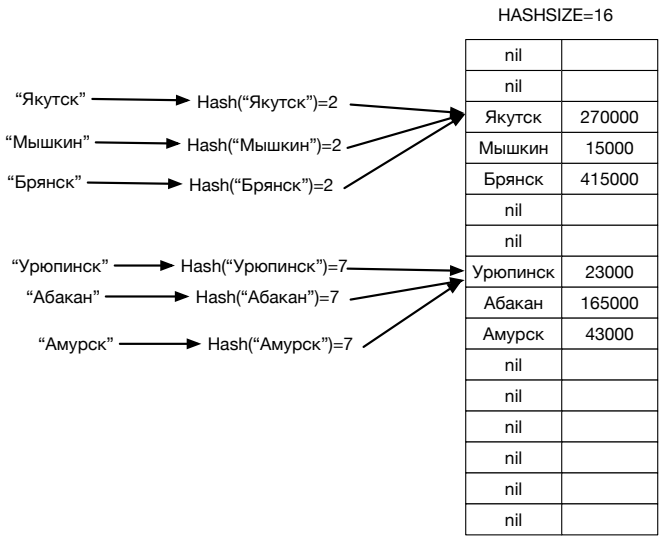


Рис. 6.143. Хеш-таблица с кластеризацией коллизий

• Старый массив удаляется.

Очевидно, что расширение таблицы — дорогая операция, которая должна произвести $O(N)$ операций вставок в новое хранилище для таблицы.

Можно посчитать амортизированную сложность N операций вставки в таблицу, если положить, что каждый раз новое хранилище будет в $k > 1$ раз больше предыдущего. Примем, что начальный размер хеш-таблицы был M, количество операций сравнения ключей есть $O(1)$ для операций вставки, а операция расширения таблицы будет проводиться при достижении фактора заполнения в 0.5.

Тогда до первого расширения будет проведено $\frac{M}{2}$ операций вставки, после чего будет создана новая копия хранилища за $k \cdot M$ операций, после этого будет произведено $\frac{M}{2}$ вставок из старого хранилища. Итого — $\frac{M}{2} + k \cdot M + \frac{M}{2}$ операций, что для вставленных M элементов даст $O(1)$ на одну операцию.

Мы рассмотрели простой способ нахождения пустого элемента, увеличивая номер позиции-кандидата каждый раз на единицу, $K = 1$ (вспомните формулу $(index + 1) \mod M$). При не очень удачной хеш-функции в таблице быстро образуются *кластера* из элементов, ключи которых оказались в коллизии. Если мы обобщим эту формулу до $(index + K) \mod M$, то, вроде бы, ничего не изменится. Но K можно сделать функцией от ключа, то есть $K = H_2(key)$. Оказывается, в этом случае кластеризация уменьшается, и коэффициент амортизации уменьшается вместе с ней. Это — *рехеширование.*

6.4.3 Рекомендации по организации хеш-таблиц

Мы рассмотрели два основных способа организации хеш-таблиц — с открытой и закрытой адресацией. Выбор организации для конкретного применения зависит от многих факторов — размера ключа и данных, сложности операции сравнения ключей, архитектуры конкретной вычислительной системы. Практика показывает, что при закрытой адресации ухудшается локальность обращения к кэш-памяти, что часто приводит к потере производительности. Важность хорошей хеш-функции трудно переоценить. В практике автора встречались примеры, когда замена одной, вроде бы хорошей и быстрой хеш-функции, на другую, более медленную, дала прирост производительности в 2.5 раза. Последняя рекомендация — не допускать большего фактора заполнения, чем 0.5-0.6. При превышении этого значения количество сравнений ключей на одну операцию резко увеличивается. Впрочем, про математическое ожидание количества поисков ключа для различной организации хеш-таблиц и различного фактора заполнения можно прочитать в бессмертной книге [3].

6.5 Хеш-таблицы во внешней памяти

Хеш-таблицы, как оказывается, очень удобны и для работы с данными, для хранения которых не хватает оперативной памяти.

Задача. Имеется $5 \cdot 10^9$ записей, состоящих из уникального ключа размером 1000 байтов и данных размером 10000 байтов.

В настоящее время данные располагаются в 5000000 файлов, в каждом из которых по 1000 строк.

Требуется организовать данные так, чтобы обеспечить быстрый поиск

по ключу. Ожидается небольшое количество операций добавления и удаления после формирования структур данных этой поисковой системы.

Общий размер превышает 2000 GB. Ожидается, что количество операций поиска будет велико. Допустимо хранение результатов преобразования данных на устройстве с произвольным доступом.

Для решения задачи можно использовать B-деревья. Но лучшим выбором будут хеш-таблицы во внешней памяти. Одним проходом по исходным данным формируется таблица, после чего файл, содержащий хеш-таблицу, может использоваться для многократного поиска, возможно, в другом приложении.

Один из возможных вариантов хранения данных в файле приведён на рисунке 6.144.

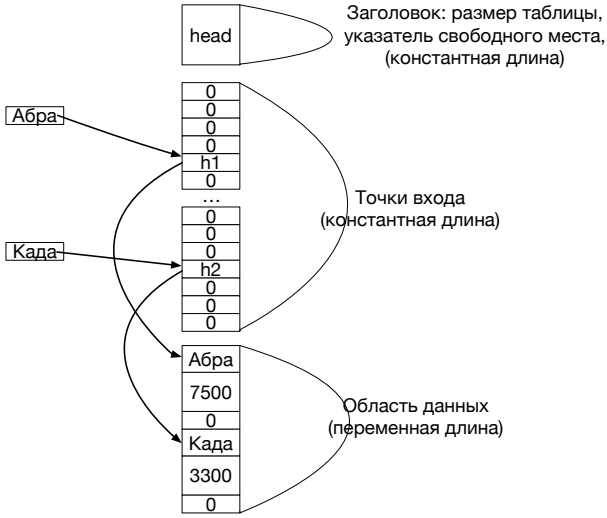


Рис. 6.144. Хеш-таблица во внешней памяти. Возможный формат организации

Весь файл разбит на три зоны.

Первая зона — фиксированного размера, заголовок. В нём можно хранить такую информацию, как размер хеш-таблицы `HASHSIZE`, начало и раз-

мер второй зоны, количество записей в таблице, смещение в файле, по которому можно записывать новые данные, и проч.

Вторая зона содержит точки входа — смещение в файле, по которому начинается поиск требуемой записи. В этой зоне находится `HASHSIZE` входов.

Третья зона содержит сами записи. Каждая запись содержит три поля — ключ, данные и точку продолжения поиска при коллизии.

Для эффективности ввода/вывода все зоны начинаются с позиций, кратных размеру блока на диске или размеру страницы виртуальной памяти. Это число — всегда степень двойки, и одно из типичных значений составляет 4096 байт. Возможный размер адреса поиска определяется максимальным размером файла с хеш-таблицей. Он может составлять, например, 8 байт для удобства.

Операция поиска в такой хеш-таблице может производиться, например, таким образом:

1. Находится значение хеш-функции от ключа `H`. Это число — номер записи во второй зоне.
2. Считывается запись под номером `H` из второй зоны как число `L`.
3. Полагается, что, начиная со смещения `L`, в файле находится запись `R`. Если сравнение ключа записи `R.key` и ключа поиска завершилось успехом, запись найдена и можно возвратить поле данных `R.data`.
4. Если ключи не сравнились, то анализируется поле `R.overflow`. Пустое значение этого поля означает, что такой записи в таблице нет.
5. Если поле `R.overflow` не пусто, оно содержит новый адрес `L`, после чего переходим к пункту 3.

Алгоритм вставки в хеш-таблицу должен вести учёт свободного места, куда можно разместить очередную запись. Если при вставке записи коллизий не произошло, то всё достаточно просто, требуется изменить запись во второй зоне. При коллизии потребуется пробежать по цепочке коллизий — и в последней изменить значение поля `R.overflow` на указатель свободного места, куда и будет вставлена новая запись. Способ удаления записи по ключу предлагается разработать самостоятельно.

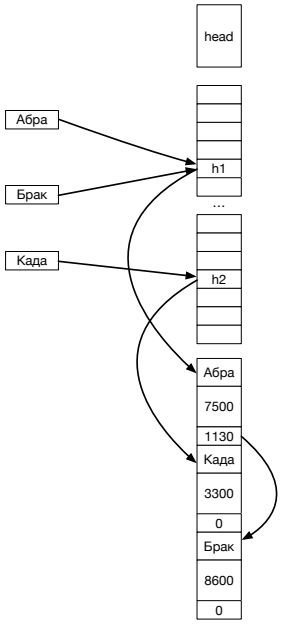


Рис. 6.145. Хеш-таблица во внешней памяти: записи переполнения

6.6 Домашние задания

Задача 26. Подстроки

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Входной файл состоит из одной строки `I`, содержащей малые буквы английского алфавита.

Назовём подстрокой длиной `L` с началом `S` множество непрерывно следующих символов строки.

Например, строка

abcab
содержит подстроки
длины 1: a, b, c, a, b
длины 2: ab, bc, ca, ab
длины 3: abc, bca, cab
длины 4: abca, bcab
длины 5: abcab.

В строках длины 1 есть два повторяющихся элемента — a и b. Назовём весом подстрок длины L произведение максимального количества повторяющихся подстрок этой длины на длину L.

В нашем случае вес длины 1 есть 2 ($2 \cdot 1$), длины 2 есть 4 ($2 \cdot 2$), длины 3 — 3 ($1 \cdot 3$), длины 4 — 4 и длины 5 — 5.

Требуется найти наибольший из всех весов различных длин.

Примеры

стандартный ввод	стандартный вывод
aabaabaabaabaa	24
abcab	5

Замечание

Длина входной строки превышает 10000 символов.

Задача 27. Большая книжка

Имя входного файла:	стандартный ввод
Имя выходного файла:	стандартный вывод
Ограничение по времени:	5 секунд
Ограничение по памяти:	4 мегабайта

Заказчику понравилось решение нашей задачи по созданию записной книжки, и он предложил нам более сложную задачу: создать простую базу данных, которая хранит много записей вида ключ:значение. Для работы с книжкой предусмотрены 4 команды:

ADD KEY VALUE — добавить в базу запись с ключом KEY и значением VALUE. Если такая запись уже есть, вывести ERROR.

DELETE KEY — удалить из базы данных запись с ключом KEY. Если такой записи нет — вывести ERROR.

UPDATE KEY VALUE — заменить в записи с ключом KEY значение на VALUE. Если такой записи нет — вывести ERROR.

`PRINT KEY` — вывести ключ записи и значение через пробел. Если такой записи нет — вывести `ERROR`.

Количество входных строк в файле с данными не превышает `300000`, количество первоначальных записей равно половине количества строк (первые `N/2` команд есть команды `ADD`).

Длины ключей и данных не превосходят `4096`. Ключи и данные содержат только буквы латинского алфавита — и цифры и не содержат пробелов.

Особенность задачи: все данные не поместятся в оперативной памяти, и поэтому придётся использовать внешнюю.

Формат входных данных

См. в примерах.

Формат выходных данных

См. в примерах.

Примеры

стандартный ввод	стандартный вывод
15 ADD RWJSN JFTF ADD ZDH GOON ADD FCDS TCAY ADD HMGVI BWK ADD JTDU TLWWN ADD IXRJ ERF ADD IAOD GRDO PRINT IXRJ PRINT JTDU PRINT IXRJ UPDATE ZDH IOX PRINT ZDH ADD GVWU RTA DELETE ZDH ADD FCDS IVFJV	IXRJ ERF JTDU TLWWN IXRJ ERF ZDH IOX ERROR

Задача 28. Сопоставление по образцу

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Известно, что при работе с файлами можно указывать метасимволы `*` и `?` для отбора нужной группы файлов, причём знак `*` соответствует любому множеству, даже пустому, в имени файла, а символ `?` — ровно одному символу в имени.

Первая строка программы содержит имя файла, состоящее только из заглавных букв латинского алфавита (`A-Z`), а вторая — образец, содержащий только заглавные буквы латинского алфавита и, возможно, символы `*` и `?`. Строки не превышают по длине 700 символов. Требуется вывести слово YES или NO в зависимости от того, сопоставляется ли имя файла указанному образцу.

Формат входных данных

`SOMETEXT`

`PATTERN`

Формат выходных данных

`YES`

или

`NO`

Примеры

стандартный ввод	стандартный вывод
ABRACADABRA ABRA*ABRA	YES
FOOBAR F??B??	YES
FOOBAR F*??O*	NO

Задача 29. Точные квадраты

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	256 мегабайт

Можете ли вы по десятичному представлению натурального числа определить, является ли это число полным квадратом? А если в числе много десятичных знаков?

Формат входных данных

Первая строка содержит $5 \leqslant N \leqslant 10^6$ — количество тех чисел, которые нужно проверить. В последующих N строках — десятичные представления натуральных чисел количеством десятичных цифр в представлении не более 100.

Формат выходных данных

Для каждого из чисел, являющихся полным квадратом, вывести его номер. Нумерация начинается с единицы.

Примеры

стандартный ввод	стандартный вывод
9 215225 264996 136161 8809 189041 870489 203601 339456 917764	3 6 9
8 690380618689483212662971952 64 480003042146133517019980195 84 276155065137459780897404545 96 555769327218234611076604674064 411163477046152583020160369764 432490594212800951537070222500 441051552494387213399506150481 837628563296051442316369723225	3 4 5 8

Задача 30. Такси-1

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	32 мегабайта

В некотором очень большом городе руководство осознало, что автономные такси, то есть такси без водителя — большое благо, и решило открыть 10 станций по аренде таких такси. Были получены данные о том, откуда клиенты могут заказывать машины. Было замечено, что если станция находится от клиента на расстоянии, не большем, чем некое число R, то клиент будет арендовать машину именно на этой станции, причём, если таких станций несколько — клиент может выбрать любую. Для экономии станции решено строить только в местах возможного расположения клиентов. Задача заключается в том, чтобы определить места наилучшей постройки, то есть такие, которые могут обслужить наибольшее количество клиентов.

Формат входных данных

В первой строке входного файла — два числа, количество клиентов и значение параметра R. В каждом из последующих N — два числа, координаты X_i и Y_i i-го клиента (нумерация ведётся с нуля).

Формат выходных данных

В выходном файле должно присутствовать не более 10 строк. Каждая строка должна содержать номер клиента, у которого выгоднее всего строить станцию, и количество обслуживаемых этой станцией клиентов, отличное от нуля. Выводимые строки должны быть упорядочены от наибольшего количества обслуживаемых клиентов к наименьшему. Если две и более станции могут обслужить одинаковое число клиентов, то выше в списке должна находиться станция с меньшим номером.

Примеры

стандартный ввод	стандартный вывод
5 3 0 0 2 -2 5 3 -2 2 5 1	0 2 1 1 2 1 3 1 4 1
10 3.000000 3.168070 1.752490 0.500730 6.436580 0.089300 0.112720 2.275440 7.508780 0.779230 4.377090 0.644400 1.381650 1.844920 1.430420 8.079870 5.225030 7.823270 5.317290 1.788400 5.426120	5 4 1 3 4 3 6 3 9 3 0 2 2 2 3 2 7 1 8 1

Лекция 7

В этой лекции мы рассмотрим метод динамического программирования как основу решения большого количества задач разного класса.

7.1 Задача о количестве маршрутов

Знакомые с ЕГЭ по информатике узна́ют следующую задачу.

Задача. Имеется несколько городов — и между некоторыми из них проведены односторонние дороги так, что выехав из города, вернуться туда невозможно. Требуется найти количество маршрутов из одного пункта в другой.

Забегая вперёд, скажем, что дорожная сеть в данной задаче представлена в виде направленного ациклического графа. Более того, задача динамического программирования часто имеет дело именно с такими графами.

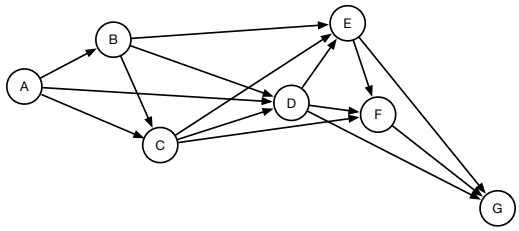


Рис. 7.146. Простая дорожная сеть

В этой задаче нужно найти количество маршрутов из пункта A в пункт G.

Введём функцию от узла: пусть $F(i)$ будет равно количеству маршрутов из A до i. Тогда $F(G) = F(F) + F(E) + F(D)$.

Аналогично:

- $F(F) = F(E) + F(D) + F(C)$,
- $F(E) = F(D) + F(C) + F(B)$,
- $F(D) = F(C) + F(B) + F(A)$,
- $F(C) = D(B) + F(A)$,
- $F(B) = F(A)$,
- $F(A) = 1$.

Это — рекурсивная задача. Каждая задача разбивается на подзадачи. После решения подзадачи требуется консолидировать результаты. Каждая из подзадач решается аналогично основной, но сама задача — несколько проще, чем главная. В данном случае города подзадач находятся немного ближе к начальному пункту, чем города задач. Имеются подзадачи, не требующие рекурсии, например, подзадача для начального пункта уже имеет решение — имеется ровно один маршрут из города в него же.

Так как задача — рекурсивная, для неё можно построить дерево рекурсии. К сожалению, даже для этой задачи оно велико.

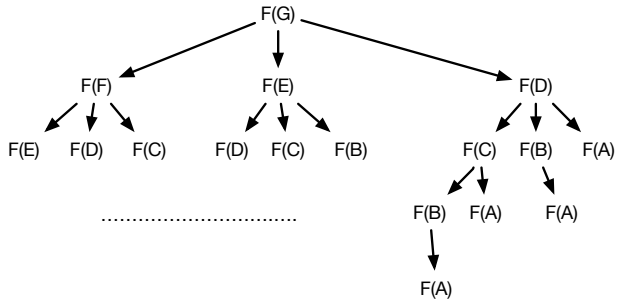


Рис. 7.147. Дерево рекурсии для решения задачи

Однако, рассматривая это дерево, мы замечаем важный факт: подзадачи частично совпадают.

Это означает, что исходная задача — задача динамического программирования.

Сформулируем условия появления задачи *динамического программирования.*

- Задача может быть разбита на произвольное количество подзадач.
- Решение полной задачи зависит только от решения подзадач.
- Каждая из подзадач по какому-либо критерию *немного* проще основной задачи.
- Часть подзадач и подзадач у подзадач совпадает.

Первый способ, который мы рассмотрим — рекурсивное решение с запоминанием результатов. Второй способ — восходящее решение в правильном порядке.

Кстати, а нельзя ли решить эту задачу *без* использования динамического программирования (*ДП*)? Мы же знаем про метод *разделяй и властвуй.*

Вспомним условия применения метода *разделяй и властвуй (РВ)*:

1. Задача может быть разбита на произвольное количество b подзадач.
2. Решение полной задачи зависит только от решения подзадач.
3. Каждая из подзадач проще основной задачи не менее, чем в b раз.

Основной способ решения задач методом РВ — рекурсия или эквивалентная ей итерация.

Различие между методами ДП и РВ одно — пункт 3, но оно фундаментально и приводит к невозможности быстрого решения задачи ДП методом чистой рекурсии.

7.1.1 Принцип Беллмана и уравнение Беллмана

Термин *динамическое программирование* появился в 1940-х годах и был связан с задачами управления. Сам Беллман сформировал принцип оптимальности так:

«Каковы бы ни были первоначальное состояние и решение, последующие решения должны составлять оптимальное поведение относительно уже решённого состояния.»

Принцип Беллмана для динамических управляемых систем таков:

- Пусть имеется управляемая система.
- S — её текущее состояние.
- $W_i = f_i(S, x_i)$ — функция выигрыша (стоимости) при использовании управления x на i-м шаге.

- $S' = \varphi_i(S, x_i)$ — состояние, в которое переходит система при воздействии x.

Независимо от значения S, нужно выбрать управление на этом шаге — так, чтобы выигрыш на данном шаге плюс оптимальный выигрыш на всех последующих шагах был максимальным.

$$W_i(S) = \max_{x_i} \left\{ f_i(S, x_i) + W_{i+1}\left(\varphi_i(S, x_i)\right) \right\}.$$

Ключ к решению задач динамического программирования — составление уравнения Беллмана. Это не так сложно, как кажется. Рассмотрим пример.

7.1.2 Уравнение Беллмана для задачи о количестве маршрутов

Пусть A — матрица смежности для городов,

$$A[i,j] = \begin{cases} 1, & \text{если имеется прямой путь из } i \text{ в } j \\ 0, & \text{если не имеется прямого пути из } i \text{ в } j. \end{cases}$$

Тогда уравнение Беллмана для задачи о количестве маршрутов будет следующим:

$$F(x) = \sum_{i=1}^{k} F(i) \cdot A[i,x].$$

Для дорожного графа на рис. 7.146 матрица смежности такова:

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \end{array} \begin{array}{c} \begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Теперь, при наличии такой матрицы, рекурсивное решение задачи становится элементарным.

```
int f(int x, int **a, int k) {
  int ret = 0;
  for (int i = 0; i < kk; i++) {
    ret += f(i,a,k) + a[i][x];
  }
  return ret;
}
```

Это — пока первый этап решения задачи, но без него задачу не решить. Попрактикуемся в составлении уравнения Беллмана для различных задач.

7.2 Задача о подпоследовательности

Задача. Имеется последовательность чисел $a_1, a_2, \ldots, a_n$. Подпоследовательность $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ называется возрастающей, если

$$1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$$

и

$$a_{i_1} < a_{i_2} < \cdots < a_{i_k}.$$

Требуется найти максимальную длину возрастающей подпоследовательности.

Например, для последовательности $a_i = \{10, 4, 13, 7, 3, 6, 17, 33\}$ одна из возрастающих подпоследовательностей есть $\{4, 7, 17, 33\}$.

Нарисуем граф задачи. Соединим направленными рёбрами элементы, которые могут быть друг за другом в подпоследовательности.

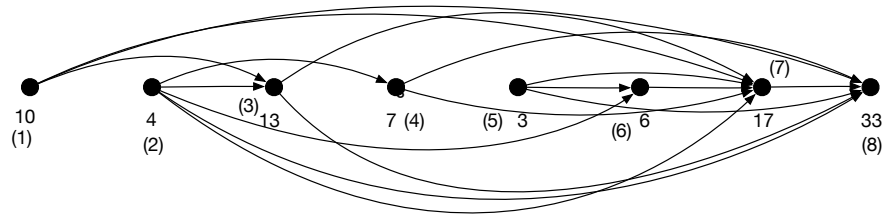


Рис. 7.148. Задача о максимальной возрастающей последовательности

Задача оказалась похожей на предыдущую, только теперь требуется найти не количество путей, а длину наибольшего пути.

В задаче на количество путей консолидация подзадач имела вид

$$R_i = \sum_{i=1}^{N_{R_{i-1}}} R_{i-1}.$$

Обозначим за N_k количество путей к вершине k а за L_k — длину наибольшего пути от вершины 1 до вершины k. Наличие пути из вершины i вершину j, где $i < j$ определяется условием $a_i < a_j$, Обозначим его c_{ij}, которое равно единице, если условие удовлетворяется и нулю в противоположном случае. Тогда в этой задаче длина наибольшего пути к вершине i (уравнение Беллмана) есть максимум из наибольших путей к предыдущим вершинам плюс единица:

$$L_i = 1 + \max_{j=1,i-1} L_j \cdot c_{ji}.$$

Задача и подзадачи выглядят так:

$$L_8 = 1 + \max(L_1, L_2, L_3, L_4, L_5, L_6, L_7),$$

$$L_7 = 1 + \max(L_1, L_2, L_3, L_4, L_5, L_6),$$

$$L_6 = 1 + \max(L_2, L_5),$$

...

Рекурсивно эта задача решается следующей программой:

```
int f(int a[], int N, int k) { // k - номер элемента
  int m = 0;
  for (int i = 0; i < k-1; i++) { // для всех слева
    if (a[i] < a[k]) { // есть путь?
      int p = f(a,N,i); // его длина
      if (p > m) m = p; // m = max(m,p)
    }
  }
  return m+1;
}
```

Для последовательности $\{1, 4, 2, 5, 3\}$ решение будет таким:

- $f_5 = 1 + \max(f_1, f_3)$
- $f_4 = 1 + \max(f_1, f_2, f_3)$
- $f_3 = 1 + \max(f_1)$

- $f_2 = 1 + \max(f_1)$
- $f_1 = 1$.

Возвращаемся назад

- $f_2 = 1 + \max(f_1) = 2$
- $f_3 = 1 + \max(f_1) = 2$
- $f_4 = 1 + \max(f_1, f_2, f_3) = 3$
- $f_5 = 1 + \max(f_1, f_3) = 3$.

Решение есть $\max(f_1, f_2, f_3, f_4, f_5) = 3$.

Чтобы повторно не решать решённые подзадачи, введём массив c размером N, хранящий значения вычисленных функций. Начальные значения его равны нулю — значению, которое не может быть верным решением любой из подзадач. Это позволит нам определить, решали ли мы эту подзадачу или нет. Если нет — запускаем решение для требуемого аргумента, и после получения результата сохраняем его.

```
int f(int a[], int N, int k, int c[]) {
  if (c[k] != 0) return c[k]; // Уже решали для k
  int m = 0;
  for (int i = 0; i < k-1; i++) { // для всех слева
    if (a[i] < a[k]) { // есть путь?
      int p = f(a,N,i); // его длина
      if (p > m) m = p; // m = max(m,p)
    }
  }
  return c[k] = m+1; // заносим в кэш и возвращаем
}
```

Такое запоминание результатов промежуточных подзадач часто называют *мемоизацией*, хотя такой метод по сути есть *кеширование* значений функции. В дальнейшем мы будем называть массив, помогающий нам избежать повторных вычислений, *кеш-таблицей*.

7.3 Ещё раз о рекурсии

Вернёмся к задаче о банкомате, которую мы разбирали во второй лекции. Напоминаем её условие:

Задача. В банкомате имеется неограниченное количество банкнот $(b_1, b_2, \ldots, b_n)$ заданных номиналов. Нужно выдать требуемую сумму денег наименьшим количеством банкнот.

Как мы уже выяснили, жадное решение возможно не для всех наборов входных данных. Например, при $b = \{1, 6, 10\}$ и $x = 12$ жадное решение даст ответ 3 $(10 + 1 + 1)$, хотя существует более оптимальное решение 2 $(6 + 6)$.

Сделаем задачу о банкомате похожей на задачу о количестве маршрутов.

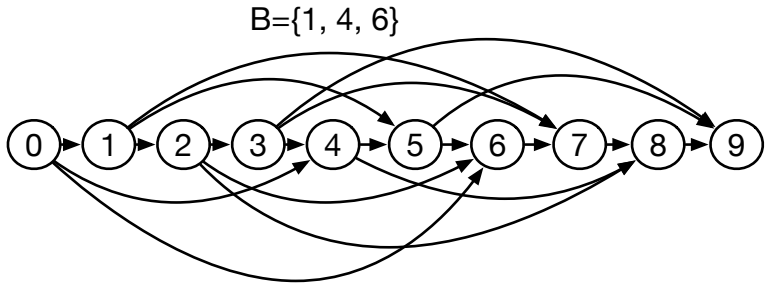


Рис. 7.149. Задача о банкомате

Для каждой суммы проведём стрелку к узлу с новой разрешённой суммой. Уравнение Беллмана непосредственно вытекает из рисунка:

$$f(x) = \begin{cases} \min\limits_{i=1,n}\{f(x - b_i)\} + 1, & \text{если } x > 0 \\ 0, & \text{если } x = 0 \\ \infty, & \text{если } x < 0 \end{cases}$$

А по уравнению Беллмана просто написать рекурсивный код.

```
const int GOOGOL = 999999999;
int f(int x, int *b, int n) {
  if (x < 0) return GOOGOL;
  if (x == 0) return 0;
  int min = GOOGOL;
  for (int i = 0; i < n; i++) {
    int r = f(x - b[i], b, n);
    if (r < min) min = r;
  }
```

```
  return min + 1;
}
```

Опять же, если решать эту задачу рекурсивно, то дерево рекурсии безобразно разрастается.

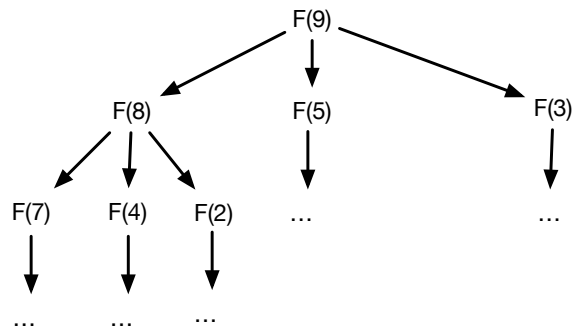


Рис. 7.150. Дерево вызовов при рекурсивном решении задачи

Несмотря на то, что глубина рекурсии не превосходит $\frac{x}{\min\limits_{i=1,n} b_i}$, количество рекурсивных вызовов растёт по экспоненте. Если при $b = \{1, 4, 6\}$ и $x = 30$ количество рекурсивных вызовов составит 285709, то для $x = 31$ — уже 418729, а для $x = 40$ оно равно 597124768.

Опять задача такова, что аргументом уравнения Беллмана является небольшое число. Следовательно, заведение массива, сохраняющего результаты и индексируемого этим числом, должно помочь. Единственная деталь — для того, чтобы понять, вычислялось ли для данного аргумента значение функции, нужно при инициализации массива занести в него подходящие элементы. В данной задаче число -1 не может быть решением, и это будет начальным значением всех элементов массива.

```
const int GOOGOL = 999999999;
int f(int x, int *b, int n, int *cache) {
  if (x < 0) return GOOGOL;
  if (x == 0) return 0;
```

```
  if (cache[x] >= 0) return cache[x];
  int min = GOOGOL;
  for (int i = 0; i < n; i++) {
    int r = f(x - b[i], b, n, cache);
    if (r < min) min = r;
  }
  return cache[x] = min + 1;
}
```

7.4 Декомпозиция задачи

Должное разбиение на подзадачи, декомпозиция, — вероятно, главная сложность при решении задачи методом динамического программирования. Для задачи о количестве путей до точки i подзадачей было определение количества путей до точек, находящихся в одном шаге от i. При условии отсутствия замкнутых маршрутов размер подзадачи всегда был несколько меньше размера задачи. Подзадача решалась тем же методом, что и задача. Наличие таких условий натолкнуло нас на мысль, что задачу можно решить методом динамического программирования.

Давайте сформулируем общий план определения, подходит ли для решения задачи метод динамического программирования.

Задача напрашивается на решение методом динамического программирования, если:

1. можно выделить множество подзадач;
2. имеется порядок на подзадачах, то есть можно показать, что задача с одними аргументами заведомо должна быть решена раньше задачи с другими аргументами; можно сказать, что один набор аргументов *меньше* другого набора;
3. имеется рекуррентное соотношение решения задачи через решения подзадач;
4. рекуррентное соотношение есть рекурсивная функция с целочисленными аргументами — или с аргументами, сводящимися к целочисленным.

При динамическом программировании:

1. принципиально исключаются повторные вычисления в любой рекурсивной функции, если есть возможность запоминать значения функции для аргументов, меньших текущего;

2. снижается время выполнения рекурсивной функции до времени, порядок которого равен сумме времён выполнения всех функций с аргументом, меньшим текущего, если затраты на рекурсивный вызов постоянны.

Правильная декомпозиция задачи — ключ к её решению. Вернёмся к задаче о рюкзаке, но немного упростим её таким образом, чтобы её можно было решать методом динамического программирования.

Задача. Имеется N предметов, каждый из которых имеет вес $H_i \in \mathbb{N}$ и стоимость $C_i \in \mathbb{N}$. Найти комбинацию предметов, имеющую наибольшую суммарную стоимость, суммарный вес которых не превышает V.

Решение задачи. Декомпозиция должна состоять в уменьшении сложности задачи, то есть уменьшении какого-либо параметра. Что есть подзадача меньшего размера? Наполнение рюкзака меньшего размера? Наполнение рюкзака меньшим количеством предметов?

При любой выбранной декомпозиции требуется ответ на вопросы:

1. какие ресурсы потребуются для запоминания результатов подзадач?
2. если имеется решение подзадач, можно ли на основе этого получить решение задачи?

Рассмотрим подзадачу «рюкзак меньшего размера». Размер памяти для результатов есть размер рюкзака. Если мы знаем результаты для всех меньших рюкзаков V_k, то поможет ли это решить задачу?

Для подзадачи «рюкзак с меньшим количеством предметов» размер памяти для результатов есть количество предметов. Если мы знаем результаты для всех подзадач с меньшим количеством предметов, то поможет ли это решить задачу?

Задача «рюкзак меньшего размера» напоминает нам задачу о банкомате. Но, увы, это всё же различные задачи. Важно, что в задаче про банкомат имеется неограниченный запас купюр каждого номинала, поэтому выбор каждой купюры не влияет на множество всех возможных ходов. А ведь если количество предметов ограничено, то часть входных данных задачи — множество предметов, которые можно взять. Поэтому успех при решении задачи о банкомате не помогает при решении этой задачи — после выбора одного из предметов для укладывания в рюкзак множество для выбора изменяется.

Но может быть, это не так страшно? Пусть аргументами подзадачи будут оставшееся множество предметов S и оставшееся место в рюкзаке L. H_i — веса предметов, C_i — их стоимости.

Тогда уравнение Беллмана принимает следующий вид:

$$F(S,L) = \begin{cases} \max\limits_{e \in S} \left(F(S-e, L-H(e)) + C(e)\right), & \text{если } L > 0 \\ 0, & \text{если } L = 0 \\ -\infty, & \text{если } L < 0 \end{cases}$$

Чтобы задача решалась методом ДП, размер пространства аргументов должен быть невелик. В данном случае он составляет

$$D = O(2^N \cdot L_0) = O(2^N),$$

что слишком много.

Попытаемся произвести декомпозицию по количеству предметов.

Мы берёмся за решение задачи с $K+1$ предметами, зная решения всех задач с K предметами. Аргументами подзадачи являются количество предметов K и оставшееся место в рюкзаке L.

Чтобы составить уравнение Беллмана, мы должны сделать индуктивный переход от множества из K предметов к множеству из $K+1$ предмета. Если у нас есть решение задачи для K предметов, то $(K+1)$-й предмет мы можем либо взять, либо не брать.

$$F(K,L) = \begin{cases} \max\limits_{i=1}^{K-1} \left(F(K-1, L-H_i) + C_i, F(K-1, L)\right) & \text{если } L > 0 \\ 0, & \text{если } L = 0 \\ -\infty, & \text{если } L < 0 \end{cases}$$

Размер пространства аргументов этой задачи оказывается существенно меньше, чем при другом способе декомпозиции.

$$D = O(N \cdot L_0)$$

Время решения задачи пропорционально пространству решений. Оказывается, и задачу о рюкзаке при ряде условий можно решить за полиномиальное время.

7.5 Восстановление решения

7.5.1 Задача о банкомате: нахождение банкнот

Решая задачу о банкомате, мы получили то, что хотели — минимальное количество банкнот. Однако если бы это была задача для реального банко-

мата[13], то решение осталось бы неполным — мы ведь не получили, какие именно банкноты требуется выдать.

Находя решение подзадачи, мы всегда имеем возможность запомнить не только ответ, но и оптимальный шаг. Нетрудно, например, просто запоминать историю получения решения.

7.5.2 Задача о банкомате: восстановление решения

Давайте сохранять цепочку вызовов рекурсивной функции, имея список банкнот для каждого промежуточного решения. Так как промежуточных решений может быть N — и каждое из решений имеет различную длину, то для их хранения потребуется N векторов.

Задача о банкомате: сохраняем список

```
const int GOOGOL = 999999999;
int f(int x, int *b, int n,
  int *cache, vector<vector<int> > &solution) {
  if (x < 0) return GOOGOL;
  if (x == 0) return 0;
  if (cache[x] >= 0) return cache[x];
  int min = GOOGOL, best = -1;
  for (int i = 0; i < n; i++) {
    int r = f(x - b[i], b, n, cache, solution);
    f (r < min) {
      min = r; best = b[i];
    }
  }
  solution[x] = solution[x - best];
  solution[x].push_back(best);
  return cache[x] = min + 1;
}
```

Довольно громоздко, не правда ли? К счастью, есть более простые способы. Но для них потребуется значение таблицы решений для аргументов.

[13]Нет-нет, там такие задачи решать не потребуется, все наборы реальных банкнот всегда позволяют решать задачу жадным алгоритмом.

Задача о банкомате: восстанавливаем решение

Доведя решение задачи до ответа и имея кэш-таблицу, можно восстановить решение без знания истории.

Предположим, что мы знаем, что точный ответ при заданных начальных значениях есть 7. Тогда возникает вопрос: какой предыдущий ход мы сделали, чтобы попасть в заключительную позицию? Введём термин *ранг* для обозначения наименьшего числа ходов, требуемого для достижения текущей позиции из начальной. Тогда каждый ход решения всегда переходит в позицию с рангом, большим строго на единицу, и это позволяет нам придумать следующий алгоритм:

1. Решение основной задачи дало в ответе k, это — ранг заключительной позиции.
2. Делаем позицию текущей.
3. Если текущая позиция имеет ранг 0, то это — начальная позиция и алгоритм завершён.
4. Рассматриваем все позиции, ведущие в текущую, и выбираем из них произвольную с рангом $k-1$.
5. Запоминаем ход, который привёл из позиции ранга $k-1$ в ранг k.
6. Понижаем ранг — $k \to k-1$ и переходим к 2.

```
vector<int> buildSolution(int x, int *b, int n, int *cache) {
  vector<int> ret;
  for (int k = cache[x]; k >= 0; k--) {
    for (int i = 0; i < n; i++) {
      int r = x - b[i];
      if (r >= 0 && cache[r] == k-1) {
        x = r;
        ret.push_back(b[i]);
        break;
      }
    }
  }
  return ret;
}
```

Функция `buildSolution` возвращает вектор, содержащий номиналы купюр, требуемых для выдачи необходимой суммы. Хотя вектор, содержащий номиналы купюр, формируется в обратном порядке, именно для этой задачи это оказывается несущественным.

Третья возможность восстановить решение — добавить ещё один кеш,

bestnote, сохраняющий номинал лучшей банкноты при лучшем решении.

```
const int GOOGOL = 999999999;
int f(int x, int *b, int n, int *cache, int *bestnote) {
    if (x < 0) return GOOGOL;
    if (x == 0) return 0;
    if (cache[x] >= 0) return cache[x];
    int min = GOOGOL, best = -1;
    for (int i = 0; i < n; i++) {
        int r = f(x - b[i], b, n, cache,bestnote);
        if (r < min) {
            min = r;
            bestnote[x] = r;
        }
    }
    return cache[x] = min + 1;
}
```

Теперь для того, чтобы получить нужные для размена банкноты для суммы в x, достаточно пробежаться по кешу банкнот:

```
while (x > 0) {
   // вывод bestcache[x];
   x -= bestcache[x];
}
```

7.6 Динамическое программирование и игры

По правилам шахмат, бесконечных партий не существует, поэтому шахматы — конечная игра с наличием цели. Деревья игры заканчиваются либо в позициях с оценкой $+\infty$, если выигрывают белые, либо с оценкой 0, если заключительная позиция — ничья, либо $-\infty$, если выигрывают чёрные. Как это ни печально звучит для шахматистов, но в каждой конкретно заданной позиции результат обоюдно лучшей игры предопределён, как и в крестиках-ноликах на доске 3×3. Сложность лишь заключается в большом размере дерева игры, поэтому неизвестна оценка начальной позиции, есть ли у белых выигрыш при обоюдно лучшей игре, или чёрные всегда могут свести партию вничью. Единственное, что пока доказано — что при обоюдно лучшей игре белые проиграть не могут. Малое дерево игры в крестиках-ноликах позволяет с определённостью сказать, что начальная позиция при лучшей игре обоих сторон ничейна.

Наличие заключительных позиций и дерева игры позволяет заявлять о том, что задача определения лучшего хода в шахматной позиции — задача динамического программирования. Давайте попробуем наметить способы её решения.

Назовём позициями ранга 0 те позиции, в которых игра завершилась с каким-либо результатом — это может быть мат королю одной из сторон, пат, или позиция, в которой остались одни короли. Например, следующая позиция — заключительная ранга 0 с оценкой $+\infty$.

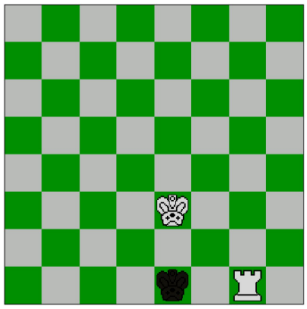


Рис. 7.151. Игра в шахматы: позиция ранга 0

Позиции ранга 1 — те позиции, которые при очередном ходе могут привести к позициям ранга 0. Если в позиции был ход белых и существует ход, который приводит к позиции ранга 0 с оценкой «выигрыш белых», то лучшим выбором будет совершить этот ход.

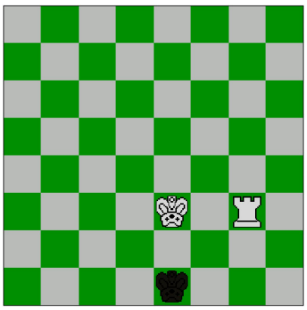


Рис. 7.152. Игра в шахматы: позиция ранга 1

Позиции ранга 2 — те позиции, которые при очередном ходе могут привести к позициям ранга 1. Если в позиции при ходе чёрных все ходы ведут в позиции с оценкой «выигрыш белых», то лучшим выбором будет сделать ход, максимально затягивающий сопротивление. Например, если в позиции имеется два хода, оба из которых неизбежно проигрывают, но один из них ведёт в позицию ранга 17, а другой — в позицию ранга 5, то лучше избрать первый.

Рис. 7.153. Игра в шахматы: позиция ранга 2

Шахматы — одна из игр, решаемых динамическим программированием. Существует большое множество позиций, особенно с небольшим количеством фигур, для которых известен их ранг. Для всех позиций с небольшим

числом фигур подсчитан их ранг. Например, известны позиции с рангом 1097, при обоюдно лучшей игре на 549-м ходу белые ставят мат.

Таблицы рангов — полный аналог кеш-таблиц в рассмотренных нами задачах. Табличный подход в шахматах изобрёл Кен Томпсон (один из изобретателей операционной системы `UNIX`) в 1970-х годах, а эффективную по памяти реализацию с помощью динамического программирования — новосибирский математик Евгений Налимов. Уже готовые таблицы с рангом позиций называются таблицами Налимова.

Для повседневного применения они, тем не менее, достаточно велики. Размер таблиц с 7-ю фигурами составляет 140 ТБ и получены они в 2012 году на компьютере «Ломоносов» ВМК МГУ. Наилучшая игра обеих сторон заключается в том, чтобы каждым очередным ходом переходить в позицию с тем же результатом и рангом, меньшим ровно на единицу.

7.7 Уход от рекурсии. Восходящее решение

Вернёмся к задаче о возрастающей подпоследовательности. При нисходящем решении нам требуется находить максимум из всех значений функций от $F(1)$ до $F(N)$, для чего рекурсивный вызов должен опуститься от $F(N)$ до $F(1)$. Для больших N уровень рекурсивных вызовов может превысить разумные рамки (размер стека в программах ограничен). Можно ли обойтись без рекурсии при решении задачи ДП, определяется структурой хеш-таблицы.

При *восходящем решении* мы пробуем решать подзадачи *до того*, как они будут поставлены. Решая задачи в возрастающем порядке, мы достигаем следующих целей:

1. Гарантируется, что все задачи, решаемые позже, будут зависеть от ранее решённых.
2. Не потребуются рекурсивные вызовы.

Обязательное условие: монотонность аргументов при решении подзадач, если $f(k)$ — подзадача для $f(n)$, то $M(k) < M(n)$, где $M(x)$ есть некая метрика сложности решения.

Для задачи о максимальной возрастающей подпоследовательности порядок решения будет таким:

- F(1) = 1
- F(2) = 1
- F(3) = max(F(1),F(2)) + 1 = 2
- ...

- F(8) = max(F(1),F(2),F(3),F(4),F(5),F(6),F(7))+1

После получения значений $F(i)$ не требуется вызывать функцию, можно использовать уже вычисленное значение.

Восходящее решение — отнюдь не панацея. Например, пусть решение задачи определяется таким образом:

$$F(n) = \max(F((n+1)/2), F(n/3)) + 1$$

$$F(0) = F(1) = 1$$

Это описывает какую-то последовательность. Нисходящий способ для $F(8)$ требует следующих вызовов:

- F(8) = max(F(4),F(2))+1
- F(4) = max(F(2),F(1))+1
- F(2) = max(F(1),F(0))+1=2
- F(4) = 3
- F(8) = 4

Восходящее решение здесь существенно более трудоёмко:

- F(0)=F(1)=1
- F(2)=max(F(1),F(0))+1=2
- F(3)=max(F(2),F(1))+1=3
- F(4)=max(F(2),F(1))+1=3
- F(5)=max(F(3),F(1))+1=3
- F(6)=max(F(3),F(2))+1=3
- F(7)=max(F(4),F(2))+1=4
- F(8)=max(F(4),F(3))+1=4

Значения F(3),F(5),F(6),F(7) можно было бы и не вычислять, так как они не используются в дальнейшем. Но ведь мы этого заранее не знаем.

Поставленная задача более подходила под рекурсивный алгоритм с запоминанием промежуточных результатов (*memoizing*). Нисходящее решение $F(N)$ имеет сложность $O(\log N)$, а восходящее — сложность $O(N)$.

С другой стороны, рассмотрим задачу, которая может быть решена следующим уравнением Беллмана:

$$F(x) = \begin{cases} 1, & \text{если } x \in \{0,1,2\} \\ 0, & \text{если } x < 0 \\ F(x-1) + F(x-3), & \text{если } x > 2 \end{cases}$$

При вычислении F(100000) в нисходящем порядке размер стека будет равен 100000, что, возможно, приведёт к аварийному завершению программы.

Здесь мы различаем понятия «алгоритм», который корректен, и «программа», которая исполняется «исполнителем». Для хранения аргументов и локальных переменных каждого рекурсивного вызова потребуется всё возрастающее количество памяти.

Последовательность будет выглядеть так:

- $\{1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, 88, 129\}$
- $F(998) \approx 2.89273 \cdot 10^{165}$
- $F(999) \approx 4.23951 \cdot 10^{165}$
- $\lim\limits_{n\to\infty} \dfrac{F(i+1)}{F(i)} \approx 1.46557$. Для хранения 999-го элемента $F(999)$ потребуется по меньшей мере $\log_2 4.23951 \cdot 10^{165} \approx 559$ бит ≈ 69 байтов, не считая управляющей информации.

Для восходящего решения задачи достаточно небольшого числа локальных переменных.

7.7.1 Восходящее решение vs нисходящее решение

Восходящее решение по корректности эквивалентно нисходящему, но при его построении необходимо обеспечить вычисление в соответствующем порядке. При простом целочисленном аргументе вычисления можно проводить в порядке увеличения аргумента. Восходящее решение требует меньшего размера стека, но может решать задачи, ответ на которые не понадобится при решении главной задачи.

7.8 Отображения и ДП

До сих в пор в этой главе решались задачи, требующие одного или двух целочисленных аргументов. Увы, не все задачи такие.

7.8.1 Задача о покрытии

Задача. Имеется прямоугольник размером 5×6. Сколькими способами его можно замостить фигурами 1×2 и 1×3? Симметрии и повороты различаются.

Решение задачи. Эту задачу можно решить методами комбинаторики[14], но мы попробуем решить её методом ДП.

[14] Многие задачи комбинаторики естественным образом решаются методом ДП.

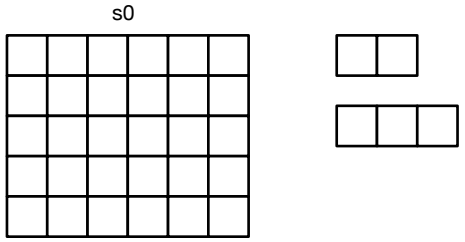


Рис. 7.154. Задача о покрытии

Обозначим через $f(s)$ количество разбиений фигуры s на требуемые фрагменты. Тогда $f(s_0) = f(s_1) + f(s_2) + f(s_3) + f(s_4)$, где s_i — подфигуры, получающиеся вычитанием одного из фрагментов, содержащих крайний левый верхний квадрат. Так как все разбиения фигуры должны содержать этот квадрат, изменим порядок таким образом, чтобы включить его первым ходом.

Возможные подзадачи:

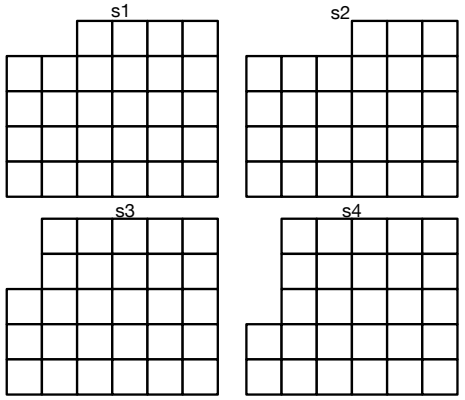


Рис. 7.155. Задача о покрытии: позиции после различных первых ходов

$$f(s_1) = f(s_{11}) + f(s_{12}) + f(s_{13}) + f(s_{14}).$$

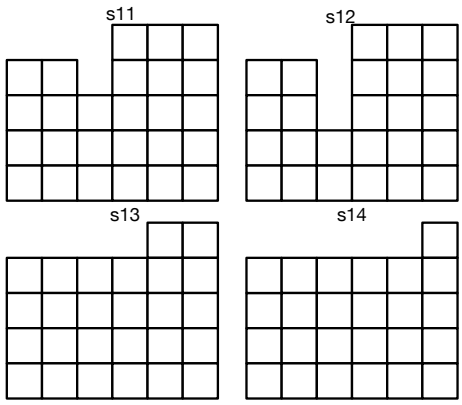


Рис. 7.156. Задача о покрытии: позиции после второго хода

Задача ли это динамического программирования — или это простой рекурсивный перебор? При предложенном алгоритме решения одна подзадача может возникнуть при различных путях:

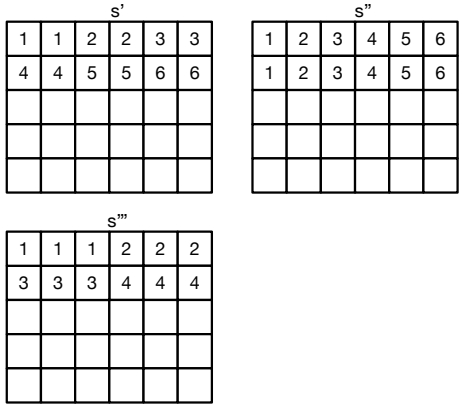


Рис. 7.157. Задача о покрытии: совпадающие подзадачи

Решение таких подзадач не зависит от истории их получения. Если имеется отображение аргументов подзадачи (позиции) на результат, то задача может быть решена методом динамического программирования.

Аргументом в функции, определяющей уравнение Беллмана, в данном случае является абстрактный тип «фигура», над которым определены операция «вырезать». Что есть $f(s)$, если s не является числом? s есть объект, который мы должны использовать в виде ключа key в отображении. $value$ в отображении есть значение, хранимое по этому ключу.

Теперь задача конкретизируется — нужно создать *взаимно однозначное* соответствие объекта s какому-то набору битов.

Объект «фигура» в данной задаче хорошо подходит под такое преобразование. Можно пронумеровать все 30 квадратиков и каждому из номеров присвоить 1, если он присутствует, и 0, если отсутствует. Результатом является строка в 30 битов. Это хорошо подходит для ключа, но размер множества возможных ключей есть 2^{30}, что слишком много для восходящего метода.

Воспользуемся изученной абстракцией «отображение». Каждая из позиций является ключом в отображении, задача решается нисходящим динамическим программированием с мемоизацией. При решении задачи из таблицы решений по ключу, соответствующему текущей позиции, извлекается значение. Если такого ключа нет, то производится полное решение и в отображение добавляется пара ключ/полученное значение. Если ключ имеется, то результатом подзадачи будет значение по ключу.

Выбор способа реализации отображения остаётся за нами. Мы можем использовать и деревья поиска, и хеш-таблицы.

7.9 Этапы решения задачи методом ДП

Сформулируем общий протокол действий при подозрении на то, что задача может быть решена методом ДП.

1. Определяется необходимость именно в динамическом программировании. При быстром уменьшении подзадач задача решается методом «разделяй и властвуй».
2. Определяется максимальный уровень рекурсии в главной задаче. Например, в задаче на покрытие прямоугольника максимальный уровень равен 15.
3. Для нетривиальных задач всегда вначале разрабатывается рекурсивный вариант решения задачи.

4. Разрабатывается метод отображения аргументов задачи в результат.
5. Если выбирается нисходящий вариант, то реализуется процесс мемоизации.
6. Если максимальный уровень слишком большой, то нисходящий метод неприменим, но по результатам исследования реализуется восходящий метод.

7.10 Многомерные варианты

7.10.1 Расстояние редактирования

Перефразируя Льва Толстого, если состояние одинаковости строк — одно, то разными они могут быть многими способами.

Если мы сделали описку в слове, исправить её можно несколькими способами — например, заменив неверную букву на верную, удалив лишнюю букву или добавив новую букву. Чем больше таких операций мы делаем, тем меньше слова (с нашей точки зрения — строки) похожи друг на друга.

Количество операций для превращения одного слова в другое называется расстоянием редактирования или расстоянием Левенштейна (между прочим, это — выпускник мехмата, доктор наук, всю жизнь проработавший в ИПМ им. Келдыша). Это расстояние — мера различия между двумя строками. Именно этой мерой измеряют сходство или различие двух генных последовательностей.

Задача. Определить минимальное количество операций для преобразования одного слова в другое, при следующих допустимых операциях:

- замена одной буквы на другую;
- вставка одной буквы;
- замена одной буквы.

Например, сколько нужно операций, чтобы превратить слово `СЛОН` в слово `ОГОНЬ`? Например, это можно сделать следующей последовательностью:
`СЛОН` → `СГОН` → `СГОНЬ` → `ОГОНЬ`

Решение задачи. Так как мы расположили эту задачу в разделе динамического программирования, то в её решении должно присутствовать уравнение Беллмана. В данном случае пока неясно, что является рекурсивной функцией, решающей данную задачу, — а точнее, что есть аргументы функции?

Давайте зафиксируем входные строки — исходную строку и строку назначения как `s` и `d`. В процессе изменения строка `s` должна превратиться в `d`. Если это — задача динамического программирования, то как найти более простые подзадачи и что является мерой простоты? Похоже на то, что если бы мы знали решения подзадач для более коротких строк, из них можно было бы вывести и решение полной задачи. Например, в приведённом выше примере, если было бы известно решение задач, как перевести `СЛОН` в `ОГОН`, то, добавив одну букву, мы бы получили решение и основной задачи. Пусть `i` и `j` — номера последних символов строк-префиксов `s` и `d` соответственно. Если оперировать только последними символами строк, то мы получаем три варианта:

- заменить последний символ строки `s` на последний символ строки `d`;
- добавить символ к концу строки `s`;
- удалить последний символ строки `s`.

Последовательность переходов, которую мы предложили ранее, не обладает нужными свойствами (последовательным увеличением сложности задачи). А есть ли нужная нам последовательность, в которой каждая подзадача решается на подстроке всё более возрастающей длины? Да, имеется.

1. В префиксах строк длины 1 `С` и `О` меняем букву `С` на `О`. `СЛОН` → `ОЛОН`.
2. В префиксах длины 2 меняем `Л` на `Г`. `ОЛОН` → `ОГОН`.
3. В префиксах длины 4 добавляем букву `Ь`. `ОГОН` → `ОГОНЬ`.

Теперь осталось это всё формализовать.

Введём двумерную матрицу `D`, элемент `D[i][j]` в которой есть минимальное из всех количеств значений различий между всеми `s` длиной от 1 до `i` и префиксом строки `d` длиной `j`.

$$D_{i,j} = \min \begin{pmatrix} D_{i-1,j-1}, & \text{если } s_i = t_i \text{ или } D_{i-1,j-1} + 1 \text{ если } s_i \neq t_i \\ D_{i-1,j} + 1 & \\ D_{i,j-1} + 1 & \end{pmatrix}.$$

Если последние символы префиксов совпадают, тогда нам ничего не надо делать — иначе требуется замена, что даст нам штраф в единицу.

Вторая строка показывает, что мы должны вставить символ для соответствия со строкой `d`.

Третье число — случай удаления лишнего символа.

Чисто рекурсивный вариант решения этой задачи, очевидно, будет слишком медленным. К счастью для нас, плотность значений аргументов и их дискретность позволяют использовать динамическое программирова-

ние, сохраняя результаты решённых задач. Более того, эту задачу можно решить без рекурсии, просто заполняя таблицу по строкам.

В качестве примера рассмотрим задачу преобразования строки `ARROGANT` в строку `SURROGATE`.

A	R	R	O	G	A	N	T

S	U	R	R	O	G	A	T	E

Рис. 7.158. Расстояние редактирования: исходные строки

Для удобства добавим один лишний левый столбец и одну лишнюю верхнюю строку к таблице результатов. Заполним их последовательно возрастающими числами. Эти числа означают, что пустая строка может превратиться в образец за число операций, равное длине образца.

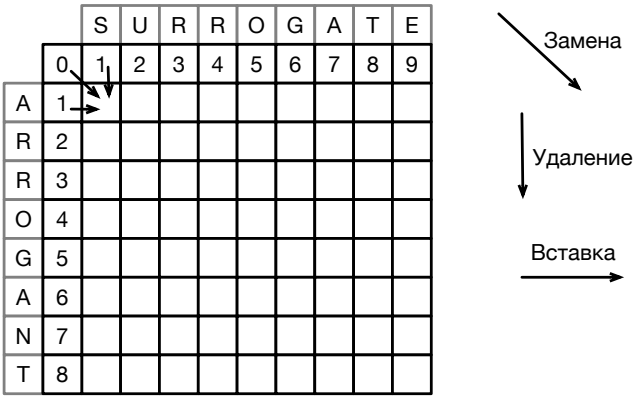


Рис. 7.159. Расстояние редактирования: первоначальное заполнение таблицы

Заполнение таблицы ведём по строкам. Значение в первой свободной ячейке зависит от трёх элементов таблицы:

		S	U	R	R	O	G	A	T	E
	0	1	2	3	4	5	6	7	8	9
A	1	1								
R	2									
R	3									
O	4									
G	5									
A	6									
N	7									
T	8									

Рис. 7.160. Расстояние редактирования: вычисление значения очередной ячейки

Так как буквы, `S` столбца и `A` строки, не совпадают, значение, полученное по диагональной стрелке, увеличивается на 1. По горизонтальной и вертикальной стрелке в ячейку приходят увеличенные на 1 числа из тех ячеек.

		S	U	R	R	O	G	A	T	E
	0	1	2	3	4	5	6	7	8	9
A	1	1	2	3	4	5	6	6		
R	2									
R	3									
O	4									
G	5									
A	6									
N	7									
T	8									

Рис. 7.161. Расстояние редактирование: совпадение букв

А вот при совпадении букв штраф за замену отсутствует — и в клеточку записывается число 6, значение, полученное по диагональной стрелке.

Итогом решения задачи является правый нижний элемент таблицы.

		S	U	R	R	O	G	A	T	E
	0	1	2	3	4	5	6	7	8	9
A	1	1	2	3	4	5	6	6	7	8
R	2	2	2	2	3	4	5	6	7	8
R	3	3	3	2	2	3	4	5	6	7
O	4	4	4	3	3	2	3	4	5	6
G	5	5	5	4	4	3	2	2	3	5
A	6	6	6	5	5	4	3	2	3	4
N	7	7	7	6	6	5	4	3	3	4
T	8	8	8	7	7	6	5	4	3	4

Рис. 7.162. Расстояние редактирования: итог

Нетрудно убедиться, что сложность алгоритма есть $|s| \times |d|$ по времени, а хранения всей таблицы не требуется, достаточно двух строк. Для больших строк, длиной в миллионы и десятки миллионов символов (сравнение генных последовательностей), алгоритм становится трудноразрешимым, и его модификации — одна из задач вычислительной биологии.

7.10.2 Задача о счастливых билетах

Задача. Билет состоит из N цифр от 0 до 9 и является счастливым, если сумма первой половины его цифр равна сумме второй половины. N — чётное число. Найти количество счастливых билетов.

Решение задачи. Казалось бы, причём здесь ДП? Это ведь комбинаторная задача. Как мы уже замечали, комбинаторные задачи часто решаются методом ДП. Эта задача — не исключение.

Сведём задачу к другой. Пусть, например, $N = 6$.

3	3	5	6	4	1
3	3	5	3	5	8

Заменим во второй половине числа все цифры на их дополнение до девяти. Количество таких чисел в точности равно количеству счастливых, так как отображение биективное. Исходный инвариант: $x_1+x_2+x_3 = x_4+x_5+x_6$. Инвариант после отображения: $x_1+x_2+x_3+(9-x_1)+(9-x_2)+(9-x_3) = 3\cdot 9$. Требуется найти количество N-значных чисел, сумма которых равна $9 \cdot \frac{N}{2}$

Привычное решение задачи наталкивается на проблему: нам нужно найти не просто количество любых чисел от 0 до 9, сумма которых 27, нужно, чтобы таких чисел было именно 6. Количество таких чисел есть сумма количеств чисел:

- первая цифра которых 0 и сумма пяти остальных равна 27;
- первая цифра которых равна 1 и сумма пяти остальных равна 26;
- ...
- первая цифра которых равна 9 и сумма остальных пяти равна 18

Вот и необходимая декомпозиция!

Нам удалось разбить задачу подзадачи меньшего ранга. Обозначим за $f(n, left)$ количество чисел, имеющих n знаков, сумма которых $left$. Тогда

$$f(6, 27) = f(5, 27) + f(5, 26) + \cdots + f(5, 19) + f(5, 18).$$

Доопределим функцию $f(n, left)$ таким образом, что при $n > 0$ и $left < 0$ она возвращает 0 и $f(1, left) = 1$, если $0 \leqslant left \leqslant 9$ и $f(1, left) = 0$ в противном случае.

Тогда $f(n, left) = \sum\limits_{i=0}^{9} f(n-1, left-i)$.

Мы свели задачу к задаче динамического программирования, но в двухмерном варианте. Если задача двухмерная, то и таблица решений — двухмерная. Первый размер определяется размерностью задачи n. Второй размер определяется максимальным значением $left = 9 \cdot \frac{n}{2}$. Нерешённые подзадачи в таблице помечаются числом -1, так как ни одна из подзадач не может возвратить отрицательное число.

Значения в таблице решений занимают последовательные ячейки, она не разрежена, следовательно, задача допускает восходящее решение. Таблица заполняется, начиная от значения $n = 1$ и всех возможных $left$ от 0 до 9. Максимальный уровень рекурсии равен n, это немного, поэтому вполне возможно решить задачу и нисходящим способом.

7.10.3 Задача о вторичной структуре РНК

Ещё одна задача из вычислительной биологии.

Задача. Имеется последовательность *оснований*

$$B = b_1 b_2 \dots b_n, b_i \in \{A, C, G, U\}.$$

Каждое основание может образовывать пару не более, чем с одним другим основанием.

$$\begin{aligned} A &\leftrightarrow U \\ C &\leftrightarrow G \end{aligned}$$

Образуется `вторичная структура`.

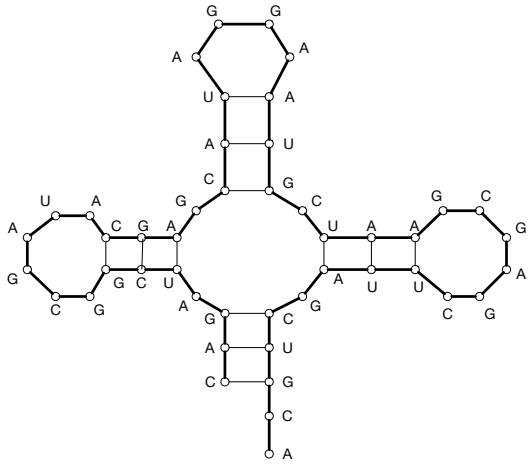


Рис. 7.163. Вторичная структура РНК: пример

Условия, накладываемые на вторичную структуру:

1. **Отсутствие резких поворотов.** Не существует пар (b_i, b_j), для которых $|i - j| \leqslant 4$.
2. **Состав пар.** Возможны только пары (A, U) и (C, G).
3. **Вхождение оснований.** Каждое основание входит не более, чем в одну пару.
4. **Отсутствие пересечений.** Для двух произвольных пар (b_i, b_j) и (b_k, b_l) невозможно условие $i < k < j < l$.

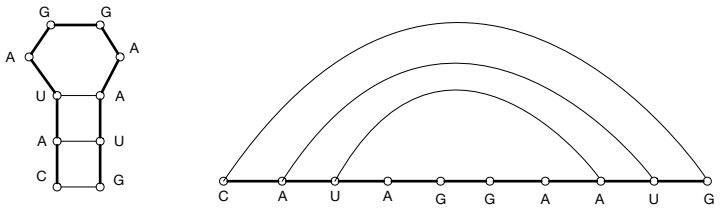


Рис. 7.164. Вторичная структура РНК: условия

Решение задачи. Задача ли это динамического программирования? Если да, то что есть «подзадача» и что есть «консолидация»?
Пусть $F(k)$ — максимальное количество пар для вторичной структуры $b_1, b_2, \ldots, b_k$.
Тогда $F(1) = F(2) = F(3) = F(4) = F(5) = 0$.
Предположим, что задачи $F(1), F(2), \ldots, F(t-1)$, уже решены. Как решить задачу для $F(t)$?
Возможны два варианта:
- во вторичной структуре $b_1, b_2, \ldots, b_t$ t не участвует в паре.
- t образует пару с каким-то элементов u, $u < t - 4$.

- Первый случай порождает подзадачу $F(t-1)$.
- А что во втором случае?

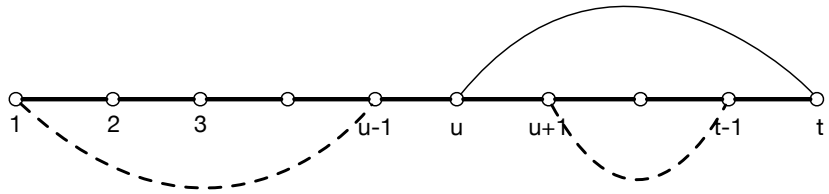


Рис. 7.165. Вторичная структура РНК: неудачная попытка декомпозиции

Имеется запрет на пересечения, следовательно, нет пар, левый конец которых меньше u, а правый — больше u.

Возникает две подзадачи, первую из которых мы решить можем, а вто-

рую — нет.

Необходимость решения задач второго рода подсказывает: за целую задачу взять $F(k, l)$, то есть два параметра.

$$F(k,l) = \begin{cases} 0, \text{ если } k+4 \leqslant l \\ \max(F(k,l-1), 1 + \max_u(F(k,u-1) + F(u+1,l-1)) & \text{иначе} \end{cases}$$

Уравнение Беллмана написано, бóльшая часть задачи решена. Для реализации восходящего решения необходимо сначала решать подзадачи, образующие более короткие подпоследовательности.

7.11 Домашние задания

Задача 31. Игра в фишки

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	16 мегабайт

На столе лежит куча из $1 \leqslant N \leqslant 10^6$ фишек. Игроки `First` и `Second` ходят строго по очереди, первый ход за игроком `First`. Каждым ходом игрок может взять из кучи любое количество фишек, не превосходящее целой части квадратного корня из оставшегося на столе количества фишек. Например при 28 фишках на столе он может взять от одной до пяти фишек. Игра заканчивается, когда на столе не остаётся ни одной фишки, и победителем объявляется тот, кто совершил последний ход.

Требуется вывести имя игрока, который побеждает при обоюдно лучшей игре.

Формат входных данных

`N`

Формат выходных данных

`First`
или
`Second`

Примеры

стандартный ввод	стандартный вывод
5	Second
10	First

Замечание

Если на доске 5 фишек, то первый игрок может взять или одну, или две фишки. Если он возьмёт одну фишку, то второй — две и наоборот. Теперь перед первым игроком лежит куча в 2 фишки, и на его единственный возможный ход — взятие одной фишки — второй забирает последнюю и выигрывает.

Задача 32. Путешествие продавца

Имя входного файла:	стандартный ввод
Имя выходного файла:	стандартный вывод
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Как обычно, кому-то надо продать что-то во многих городах. Имеются города, представленные как M множеств (столбцов) по N городов (строк) в каждом.

Продавец должен посетить ровно по одному городу из каждого множества, затратив на это как можно меньшую сумму денег. Он должен посетить сначала город из первого множества, затем из второго и так далее, строго по порядку. Он может выбирать начало своего путешествия. Число, которое находится в i-й строке и j-м столбце, означает стоимость перемещения из предыдущего места в этот город. Однако имеется ограничение на перемещения: продавец может перемещаться из города в i-й строке только в города следующего столбца, находящиеся в одной из строк $i-1, i, i+1$, если такие строки существуют.

Иногда, чтобы заставить продавца посетить какой-то город, ему доплачивают, то есть, стоимость перемещения может быть отрицательной.

Требуется определить наименьшую стоимость маршрута и сам маршрут.

Формат входных данных

```
N M
C11 C12 ... C1M
C21 C22 ... C2M
```

```
...   ....   .... ....
CN1 CN2 ... CNM
```

$3 \leqslant N \leqslant 150$
$3 \leqslant M \leqslant 1000$
$-1000 \leqslant C_{ij} \leqslant 1000$

Формат выходных данных

Первая строка — список через пробел номеров строк (начиная с 1) из M посещённых городов.

Вторая строка — общая стоимость поездки.

Если имеется несколько маршрутов с одной стоимостью, требуется вывести маршрут, наименьший в лексикографическом порядке.

Начинать и заканчивать маршрут можно в любой строке.

Примеры

стандартный ввод	стандартный вывод
5 4 1 7 4 3 5 1 6 7 4 1 9 2 7 3 7 5 8 2 4 1	1 2 1 1 9
5 6 3 4 6 2 8 6 6 1 8 2 7 4 5 9 3 9 9 5 8 4 1 3 9 6 3 7 2 8 6 4	1 2 3 2 2 2 20

Задача 33. Наибольшая общая подстрока

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Во входном файле находятся две строки длиной до 30000 символов, состоящих из цифр и прописных и строчных букв латинского алфавита, каждая в отдельной строке файла.

Необходимо найти общую подстроку наибольшей длины. Если таких подстрок несколько, то следует вывести ту из них, которая лексикографически меньше.

Обратите внимание, что в приведённом примере имеется две подстроки длины 4 — `rash` и `abra`. Несмотря на то, что первая встречается раньше, ответом будет вторая, так как она лексикографически меньше.

Пример

стандартный ввод	стандартный вывод
ubrashvabracadabra calamburashabratha	abra

Задача 34. Счастливые билеты

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Билет состоит из чётного числа N цифр в M-ричной системе счисления. Счастливым билетом называется билет, сумма первой половины цифр которого равна сумме второй половины цифр.

Найти количество счастливых билетов. Учтите: их число может быть велико.

Формат входных данных

`N M`

$2 \leqslant N \leqslant 150$
$N \mod 2 = 0$
$2 \leqslant M \leqslant 26$

Формат выходных данных

Количество счастливых билетов

Примеры

стандартный ввод	стандартный вывод
6 10	55252
28 12	358061060774375014229298133 20

Задача 35. Самая тяжёлая подтаблица

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	3 секунды
Ограничение по памяти:	256 мегабайт

В каждой ячейке прямоугольной таблицы размером $N \times M$ записано число от -10^9 до 10^9.

Назовём подтаблицей любую часть таблицы, включая целую, образующую прямоугольник, а её весом — сумму всех её чисел.

Найти вес самой тяжёлой из всех возможных подтаблиц, которые можно построить на основе оригинальной.

Формат входных данных

```
N M
C11 C12 ... C1M
C21 C22 ... C2M
... ... ... ...
CN1 CN2 ... CNM
```

$5 \leqslant N, M \leqslant 500$

Формат выходных данных

`MaximalPossibleWeight`

Примеры

стандартный ввод	стандартный вывод
5 5 1 0 7 -8 2 2 7 -5 3 1 6 -8 4 2 1 -7 3 1 -2 1 2 7 4 0 -50	24
5 5 10 -1 -1 7 -3 -6 -6 5 7 -6 8 -2 1 5 6 -1 -2 -3 -8 1 -9 -9 5 6 -1	27

Лекция 8

8.1 Графы. Представление графов

Мы с самого начала книги сталкиваемся с графами. Настало время рассказать о них поподробнее.

- Географические карты. Какой маршрут из Москвы в Лондон требует наименьших расходов? Какой требует наименьшего времени? Требуется информация о связях между городами и о стоимости этих связей.
- Микросхемы. Транзисторы, резисторы и конденсаторы связаны между собой проводниками. Есть ли короткие замыкания в системе? Можно ли так переставить компоненты, чтобы не было пересечения проводников?
- Расписания задач. Одна задача не может быть начата без решения других, следовательно, имеются связи между задачами. Как составить график решения задач так, чтобы весь процесс завершился за наименьшее время?
- Компьютерные сети. Узлы — конечные устройства, компьютеры, планшеты, телефоны, коммутаторы, маршрутизаторы... Каждая связь обладает свойствами латентности и пропускной способности. По какому маршруту послать сообщение, чтобы оно было доставлено до адресата за наименьшее время? Есть ли в сети «критические узлы», отказ которых приведёт к разделению сети на несвязные компоненты?
- Структура программы. Узлы — функции в программе. Связи — может ли одна функция вызвать другую (статический анализ) или что она вызовет в процессе исполнения программы (динамический анализ).

Чтобы узнать, какие ресурсы потребуется выделять системе, требуется граф исполнения программы.

8.1.1 Графы: основные термины

Определение 31. ***Ориентированный граф:*** *$G = (V, E)$ есть пара из V — конечного множества и E — подмножества множества $V \times V$. Обозначается как (v_i, v_j),*

Определение 32. ***Вершины графа:*** *элементы множества V (vertex, vertices).*

Определение 33. ***Рёбра графа:*** *элементы множества E (edges).*

Определение 34. ***Неориентированный граф:*** *рёбра есть неупорядоченные пары.*

Определение 35. ***Петля:*** *ребро из вершины v_1 в вершину v_2, где $v_1 = v_2$.*

Определение 36. ***Смежные вершины:*** *Вершина v_j смежна вершине v_i если имеется ребро (v_i, v_j).*

Определение 37. ***Множество смежных вершин:*** *обозначаем $Adj[v]$.*

Определение 38. ***Степень вершины:*** *величина $|Adj[v]|$.*

Определение 39. ***Путь из v_0 в v_n:*** *последовательность рёбер, таких, что $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2) \dots e_n = (v_{n-1}, v_n)$.*

Определение 40. ***Простой путь:*** *путь, в котором все вершины попарно различны.*

Определение 41. ***Длина пути:*** *количество n рёбер в пути.*

Определение 42. ***Цикл:*** *путь, в котором $v_0 = v_n$.*

Определение 43. ***Неориентированный связный граф:*** *для любой пары вершин существует путь между ними.*

Определение 44. ***Связная компонента вершины*** *v: множество вершин неориентированного графа, до которых существует путь из v.*

Определение 45. ***Расстояние между*** *$\delta(v_i, v_j)$: длина кратчайшего пути из v_i в v_j.*

$$\delta(u, v) = 0 \Leftrightarrow u = v,$$
$$\delta(u, v) \leqslant \delta(u, v') + \delta(v', v).$$

Определение 46. ***Дерево****: связный граф без циклов.*

Определение 47. ***Граф со взвешенными рёбрами****: каждому ребру приписан вес* $c(u, v)$.

Пример ориентированного графа[15]:

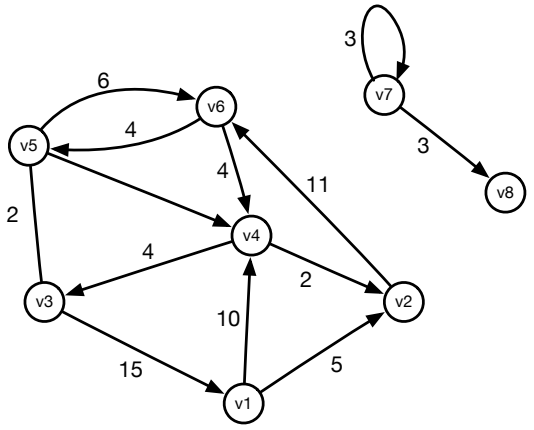


Рис. 8.166. Ориентированный граф с двумя компонентами связности

Вот несколько типичных задач, связанных с графами:
- Дан граф. Является ли он связным?
- Является ли граф деревом?
- Найти кратчайший путь из узла u в узел v.
- Найти цикл, проходящий по всем рёбрам ровно один раз (цикл Эйлера).
- Найти цикл, проходящий по всем вершинам ровно один раз (цикл Гамильтона).
- Проверка на планарность — определить, можно ли нарисовать граф на плоскости без самопересечений.

[15]Заметим, что в олимпиадной математике, в отличие от информатики, кратные рёбра и петли по умолчанию запрещены.

Очевидно, что список задач на графы воистину обширен — и у нас не будет возможности изучить все существующие алгоритмы. Но ряд алгоритмов мы всё же изучим. Перед этим рассмотрим варианты представления графов как структур данных.

8.1.2 Представление графа в памяти

Имеется несколько популярных способов представления графов в памяти — в виде матрицы смежности, в виде множеств смежности и в виде множества рёбер.

Представление графа в памяти в виде матрицы смежности

Квадратная матрица Adj размером $|V| \times |V|$, где элемент Adj_{ij} есть вес ребра от узла V_i до V_j. Если граф — невзвешенный, то обычно наличие связи обозначается единицей (или логической истиной), отсутствие — нулём или логической ложью. Это представление не позволяет описывать графы, содержащие кратные рёбра (мультирёбра).

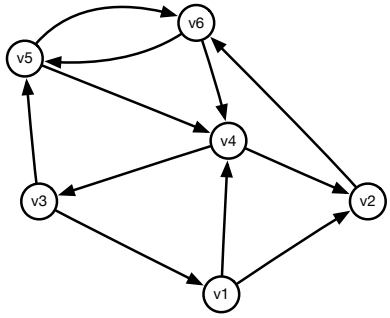


Рис. 8.167. Ориентированный невзвешенный граф

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	0	1
3	1	0	0	0	1	0
4	0	1	1	0	0	0
5	0	0	1	1	0	1
6	0	0	0	1	1	0

Пример матрицы смежности взвешенного графа.

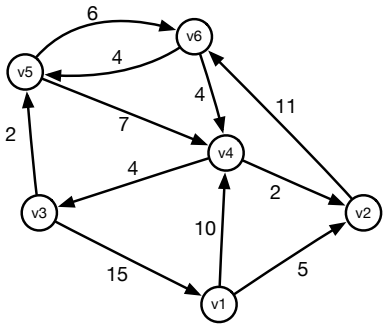


Рис. 8.168. Ориентированный взвешенный граф

	1	2	3	4	5	6
1	0	5	0	10	0	0
2	0	0	0	0	0	11
3	15	0	0	0	2	0
4	0	2	4	0	0	0
5	0	0	0	7	0	6
6	0	0	0	4	4	0

Представление графа в памяти в виде множеств смежности

Для каждого узла имеется множество смежных с ним узлов или *соседей*.
$v_1 : \{2, 4\}$
$v_2 : \{6\}$
$v_3 : \{1, 5\}$

$v_4 : \{2, 3\}$
$v_5 : \{3, 4, 6\}$
$v_6 : \{4, 5\}$

Для взвешенного графа элементы множеств смежности есть пары, один элемент которых есть номер смежного узла, а другой — вес связи. Мультирёбра в этом представлении, хотя и возможны, но ограничивают гибкость представления.

$v_1 : \{(2, 5), (4, 10)\}$
$v_2 : \{(6, 11)\}$
$v_3 : \{(1, 15), (5, 2)\}$
$v_4 : \{(2, 2), (3, 4)\}$
$v_5 : \{(4, 7), (6, 6)\}$
$v_6 : \{(4, 4), (5, 4)\}$

Представление взвешенного графа в памяти в виде списка рёбер

Для некоторых алгоритмов оказывается достаточным, если граф представлен массивом троек: {откуда, куда, стоимость}. Мультирёбра в этом представлении вполне возможны.

$$\{\{1, 2, 5\}, \{1, 4, 10\}, \{2, 6, 11\}, \{3, 1, 15\}, \{3, 5, 2\},$$
$$\{4, 2, 2\}, \{5, 4, 7\}, \{5, 6, 6\}, \{6, 4, 4\}, \{6, 5, 4\}\}$$

В таблице представлены преимущества и недостатки методов представления.

Представление	Матрица смежности	Множества смежности	Список рёбер
Занимаемая память	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$	$O(\|E\|)$
Особенности	Простой доступ	Требует мало памяти для ряда графов	Можно иметь мультирёбра

8.2 Обход графа. `BFS`. `DFS`

Давайте введём ещё один термин:

Определение 48. ***Предшественник*** $\pi(u)$ ***на пути от*** s*: предпоследняя вершина в кратчайшем пути из* s *в* u*.*

8.2.1 Обход графа: поиск в ширину, BFS

Поиск в ширину от вершины s — просмотр вершин графа в порядке возрастания расстояния от s. Это позволяет, например, для невзвешенных графов решить задачу определения кратчайшего расстояния от одной вершины до других. Побочным эффектом данного алгоритма является установление достижимости одной вершины из другой.

В качестве вспомогательной структуры данных используется абстракция `Queue`, очередь вершин, которые ещё не посещены алгоритмом. Сам алгоритм очень прост по сути. Для каждой вершины `u` определим её цвет `c[u]`: белый будет обозначать, что вершина ещё не обработана, серый — что она обрабатывается, а чёрный — что обработка завершена. Второй атрибут вершины `d[u]` — её расстояние до начальной. Алгоритм начинается выкрашиванием начальной вершины в серый цвет и установлением всех расстояний (кроме начальной вершины) в бесконечность. Начальная вершина отправляется в очередь — и начинается главный цикл. На каждой итерации из очереди забирается очередная серая вершина — и все её необработанные соседи помещаются в очередь, предварительно будучи выкрашены в серый цвет. После того, как все смежные вершины просмотрены и часть из них отправлена в очередь, текущая вершина выкрашивается в чёрный цвет.

В этой лекции мы отступаем от традиции приводить код исключительно на `C++`. В некоторых алгоритмах вас ждёт псевдокод, похожий на тот, который применяется в книге [4]. Почему? Потому, что имеются различные представления структуры графа, а также различные способы реализации вспомогательных абстракций и универсальный код здесь невозможен.

```
1:  procedure BFS(G : Graph, s : Vertex)
2:      for all u ∈ V[G] \ {s} do
3:          d[u] ← ∞;      c[u] ← white;      π[u] ← nil
4:      end for
5:      d[s] ← 0
6:      c[s] ← grey
7:      Q.enqueue(s)
8:      while Q ≠ ∅ do
9:          u ← Q.dequeue()
10:         for all v ∈ Adj[u] do
11:             if c[v] = white then
12:                 d[v] ← d[u] + 1
13:                 π[v] = u
14:                 Q.enqueue(v)
```

```
15:                 c[v] = grey
16:             end if
17:         end for
18:         c[u] ← black
19:     end while
20: end procedure
```

Попробуем проиллюстрировать алгоритм на простом графе (рис. 8.169).

В начале алгоритма все вершины, кроме одной, белые — и очередь содержит начальную вершину.

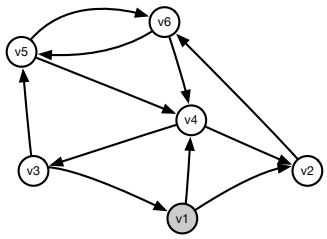


Рис. 8.169. BFS: нераскрашенный граф

$$d = \{0, \infty, \infty, \infty, \infty, \infty\} \qquad \pi = \{nil, nil, nil, nil, nil, nil\} \qquad Q = \{v_1\}.$$

После первой итерации цикла `While` в очередь попали все смежные с v_1 вершины, так как они были белыми (рис. 8.170).

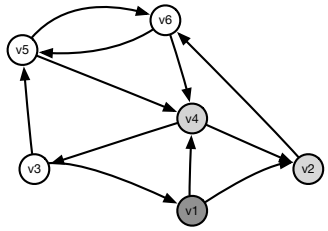


Рис. 8.170. BFS: начат обход с вершины v_1

$d = \{0, 1, \infty, 1, \infty, \infty\}$ $\pi = \{nil, v_1, nil, v_1, nil, nil\}$ $Q = \{v_4, v_2\}$.

При втором прохождении цикла `While` из очереди извлечена вершина v_4 — и туда добавлена вершина v_3. Соседняя с v_4 вершина уже имеет серый цвет, и поэтому она игнорируется.

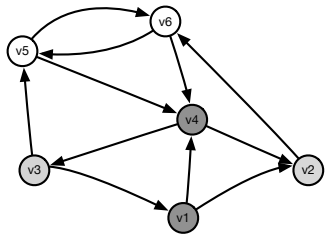


Рис. 8.171. BFS: полностью обработаны вершины v_1 и v_4

$d = \{0, 1, 2, 1, \infty, \infty\}$ $\pi = \{nil, v_1, v_4, v_1, nil, nil\}$ $Q = \{v_2, v_3\}$.

Третье прохождение цикла `While` уберёт из очереди v_2 и добавит в конец очереди v_6.

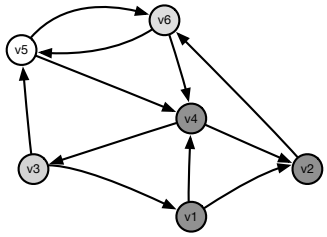


Рис. 8.172. BFS: после третьей итерации

$d = \{0, 1, 2, 1, \infty, 2\}$ $\pi = \{nil, v_1, v_4, v_1, nil, v_2\}$ $Q = \{v_3, v_6\}$.

После четвёртого прохождения цикла `While` очередь покинет вершина v_3 и в неё добавится вершина v_5. После этого все вершины выкрашены в цвета, отличные от белого, и в очередь больше ничего добавляться не будет.

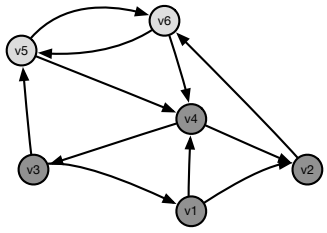


Рис. 8.173. BFS: после четвёртой итерации

$d = \{0, 1, 2, 1, 3, 2\}$ $\pi = \{nil, v_1, v_4, v_1, v_3, v_2\}$ $Q = \{v_6, v_5\}$.

Завершение алгоритма даёт нам заполненные массивы расстояний от начальной вершины и предшественников.

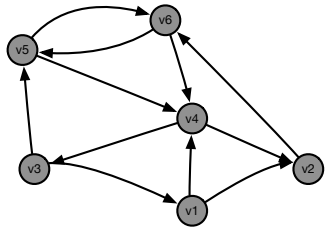


Рис. 8.174. BFS: итог

$d = \{0, 1, 2, 1, 3, 2\}$ $\pi = \{nil, v_1, v_4, v_1, v_3, v_2\}$ $Q = \{\}$.

Представление в виде множеств смежности для этого алгоритма наиболее удобно, так как позволяет легко узнать все смежные вершины. Сложность алгоритма посчитать несложно.

Фаза инициализации имеет сложность $O(|V|)$. Каждая вершина попадает в очередь не более одного раза. Для каждой попавшей в очередь вершины проверяются все смежные вершины, что имеет сложность

$$\sum_{v \in V} |Adj(v)| = O(|E|).$$

Итоговая сложность алгоритма — $T = O(|V| + |E|)$.

Все окрашенные по завершении обхода вершины составляют компоненту связности для начальной вершины, то есть множество достижимых из начальной вершин. К сожалению, является ли эта компонента связности *сильной*, когда все вершины компоненты достижимы друг из друга, или *слабой*, этот алгоритм ответа не даёт. А вот применение следующего обхода может сочетаться с алгоритмом выделения компонент сильной связности, и это — обход в глубину, `DFS`.

8.2.2 Поиск в глубину: алгоритм `DFS`

Этот алгоритм пытается идти вглубь, пока это возможно. Обнаружив вершину, алгоритм не возвращается, пока не обработает всех её потомков. Цвета, в которые окрашиваются вершины, те же самые: белый для непросмотренных вершин, серый для обрабатываемых вершин и чёрный для обработанных вершин.

Используются переменные:

- $time$ — глобальные часы;
- $d[u]$ — время начала обработки вершины u;
- $f[u]$ — время окончания обработки вершины u;
- $\pi[u]$ — предшественник вершины u.

Сам алгоритм часто используется для получения компонент связности всего графа, поэтому его делят на две части — нерекурсивную и рекурсивную. Нерекурсивная часть просто инициализирует переменные и запускает рекурсивный обход `DFS-vizit` для каждой из необработанных вершин.

```
1:  procedure DFS(G : Graph)
2:      for all u ∈ V[G] do
3:          c[u] ← white;        π[u] ← nil
4:      end for
5:      time ← 0
6:      for all u ∈ V[G] do
7:          if c[v] = white then
8:              DFS-vizit(v)
9:          end if
10:     end for
11: end procedure
```

Рекурсивная часть на время выкрашивает вершину в серый цвет, устанавливает время входа `d[u]` и рекурсивно вызывает себя для всех необработанных смежных вершин. После завершения обработки фиксируется время выхода `f[u]`, и вершина выкрашивается в чёрный цвет.

```
1:  procedure DFS-VIZIT(u : Vertex)
2:      c[u] ← grey
3:      time ← time + 1
4:      d[u] ← time
5:      for all v ∈ Adj[u] do
6:          if c[v] = white then
7:              π[v] ← u
8:              DFS-vizit(v)
9:          end if
10:     end for
11:     c[u] ← black
12:     time ← time + 1
13:     f[u] ← time
14: end procedure
```

Прогон алгоритма DFS: начинается обход с вершины v_1.

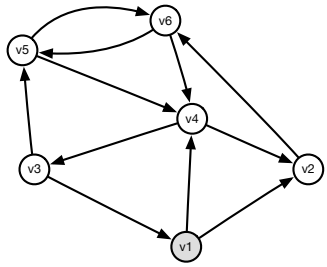


Рис. 8.175. DFS: запуск алгоритма с вершины v_1

$d = \{1, -1, -1, -1, -1, -1\}$
$f = \{-1, -1, -1, -1, -1, -1\}$
$\pi = \{nil, nil, nil, nil, nil, nil\}$
Первый рекурсивный вызов DFS-vizit(v_2).

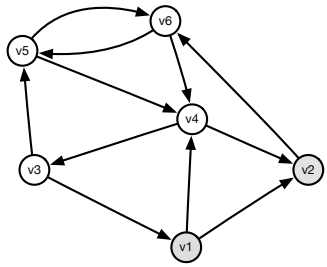


Рис. 8.176. DFS: первый рекурсивный вызов входа в вершину v_2

$d = \{1, 2, -1, -1, -1, -1\}$
$f = \{-1, -1, -1, -1, -1, -1\}$

$\pi = \{nil, v_1, nil, nil, nil, nil\}$

Второй рекурсивный вызов `DFS-vizit`(v_6). Далее рекурсия идёт по вершинам v_4, v_3 и v_5, после чего оказывается, что у вершины v_5 нет необработанных соседей, следовательно, из рекурсии на v_5 нужно выйти.

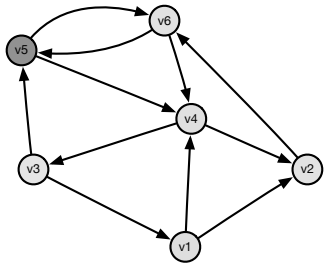


Рис. 8.177. DFS: первый выход из рекурсии на вершине v_5

$d = \{1, 2, 5, 4, 6, 3\}$
$f = \{-1, -1, -1, -1, 7, -1\}$
$\pi = \{nil, v_1, v_4, v_6, v_3, v_2\}$
Для всех остальных вершин рекурсия тоже завершается.

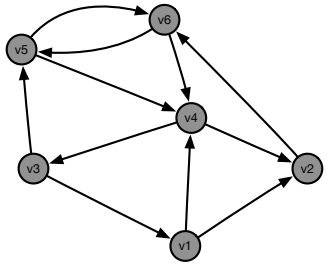


Рис. 8.178. DFS: итог

$d = \{1, 2, 5, 4, 6, 3\}$
$f = \{12, 11, 8, 9, 7, 10\}$
$\pi = \{nil, v_1, v_4, v_6, v_3, v_2\}$

Чтобы определить сложность алгоритма будем полагать, что граф представлен множествами смежности. Тогда инициализация потребует $O(|V|)$, посещение каждой вершины ровно по одному разу — $O(|V|)$ и проверка каждого ребра — $O(|E|)$, итого — $O(|V| + |E|)$.

Не пытайтесь найти кратчайшие маршруты этим алгоритмом, он для этого не предназначен. А для чего он? Например, для нахождения компонент связности (об этом чуть позже) или для топологической сортировки.

8.2.3 Топологическая сортировка

Задача. Имеется ориентированный граф $G = (V, E)$ без циклов.

Требуется указать такой порядок вершин на множестве V, что любое ребро ведёт из меньшей вершины к большей.

Нам потребуется небольшая модификация алгоритма `DFS` и структура данных `Queue`. Времена входов и выходов наряду с предшественниками не понадобятся.

```
 1: procedure TopoSort(G : Graph)
 2:     L ← ∅
 3:     for all u ∈ V[G] do
 4:         c[u] ← white;
 5:     end for
 6:     for all u ∈ V[G] do
 7:         if c[v] = white then
 8:             DFS-vizit(u)
 9:         end if
10:     end for
11: end procedure
```

```
 1: procedure DFS-vizit(u : Vertex)
 2:     c[u] ← grey
 3:     for all v ∈ Adj[u] do
 4:         if c[v] = white then
 5:             DFS-vizit(u)
 6:         end if
 7:     end for
 8:     c[u] ← black
```

```
 9:     L.insert(u)
10: end procedure
```

Прогон алгоритма топологической сортировки

Пусть обход следующего графа начнётся с вершины v_1:

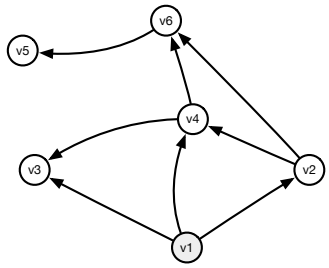


Рис. 8.179. Топологическая сортировка

В результате обхода около каждой вершины написан её порядковый номер.

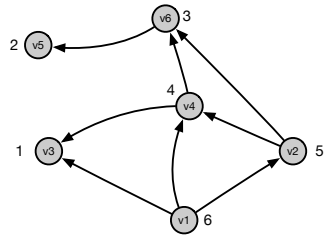


Рис. 8.180. Топологическая сортировка: результат

Порядок вершин (добавляем **в начало** по номерам): $V_1, V_2, V_4, V_6, V_5, V_3$.

А теперь вспомним метод динамического программирования. Мы использовали термин «порядок на подзадачах». Теперь всё это можно сказать более строго: подзадачи должны образовывать ациклический направленный граф. Процедура топологической сортировки на этом графе и даст нам порядок решения этих подзадач, так как будет гарантировано, что все подзадачи решаются раньше самой задачи.

8.3 Поиск компонент связности

Для неориентированных графов найти компоненты связности можно, например, запустив поиск `BFS` или `DFS`. Все выкрашенные по завершении поиска вершины образуют компоненту связности. После этого выбирается произвольным образом необработанная вершина и алгоритм повторяется, формируя другую компоненту связности. Алгоритм заканчивается, когда не остаётся необработанных вершин.

Если требуется найти компоненты сильной связности, то для ориентированных графов эта процедура не приведёт к правильному решению. Требуется другой алгоритм, например, прост в реализации *алгоритм Косарайю* (Kosaraju).

Попробуем найти все компоненты сильной связности для следующего графа:

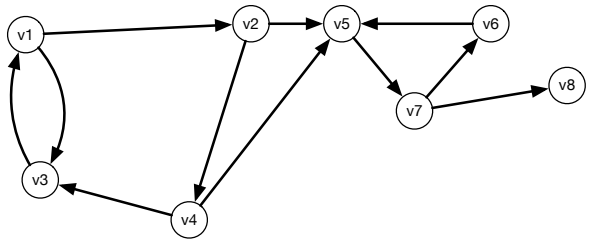


Рис. 8.181. Поиск компонентов сильной связности: пример

Для данного графа проведём полный DFS поиск. Так как в алгоритме полного DFS не специфицировано, с какой вершины начинается поиск, можно выбрать произвольную. Для наблюдения за алгоритмом давайте около каждой вершины писать два числа: время входа в вершину и через знак / — время выхода из вершины.

Начав обход с вершины v_2, мы можем получить такой результат:

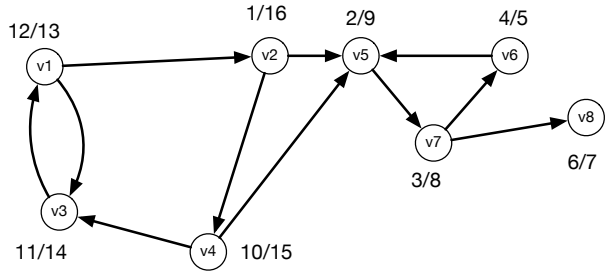


Рис. 8.182. Поиск компонент сильной связности: прямая нумерация

До сих пор время входа/выхода из вершины нам не требовалось. Составим таблицу этого времени для всех вершин:

Номер вершины	1	2	3	4	5	6	7	8
Время входа/выхода	12/13	1/16	11/14	10/15	2/9	4/5	3/8	6/7

Следующая операция — заменить направления всех связей (перевернуть все стрелки). Если из вершины `u` была связь с вершиной `v`, то после переворота вершина `v` получит связь с вершиной `u`. Вот что получится для нашего графа:

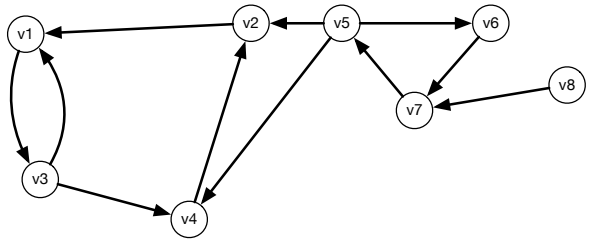


Рис. 8.183. Поиск компонент сильной связности: обращённый граф

Этот граф нужно обойти ещё раз. Но теперь порядок обхода уже не произволен. Мы должны выбрать в качестве начальной вершины ту из необработанных, у которой наибольшее значение времени выхода. В нашем случае это вершина 2. Обход покрасил вершины v_1, v_2, v_3 и v_4. После завершения обхода остались непокрашенные вершины v_5, v_6, v_7 и v_8. Из них снова выбираем вершину с наибольшим временем выхода — и так до тех пор, пока останутся непокрашенные вершины.

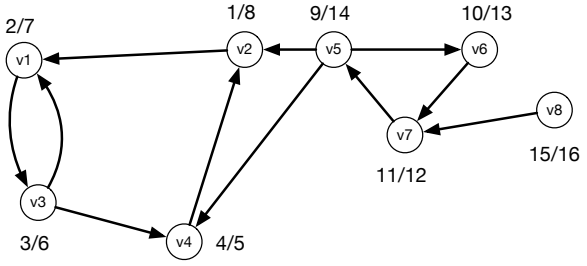


Рис. 8.184. Поиск компонент сильной связности: обход обращённого графа

Каждый «малый» проход алгоритма DFS даст нам вершины, которые принадлежат одной компоненте сильной связности.

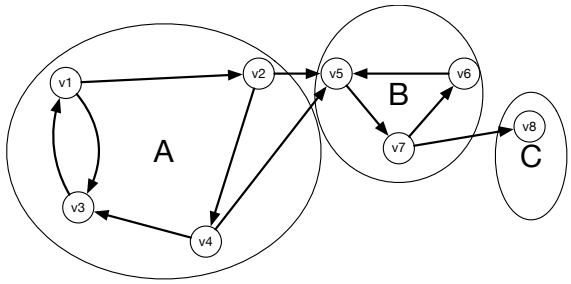


Рис. 8.185. Поиск компонент сильной связности: выделение компонентов

Рассматривая компоненту сильной связности как единую мета-вершину, мы получаем новый граф, который называется *конденсацией* исходного графа или *конденсированным графом*.

8.4 Поиск специальных элементов графа

Алгоритм поиска в глубину служит основой для большого количества других алгоритмов на графах. Мы уже упоминали про задачу определения планарности графа. В ней требуется установить, можно ли таким образом нарисовать граф на плоскости, что все рёбра будут изображены непересекающимися линиями. Один из самых эффективных алгоритмов определения планарности — гамма-алгоритм, требует нахождения специальных вершин графа, *точек сочленения* и специальных рёбер графа, *мостов*. Здесь и далее мы будем рассматривать только неориентированные графы $G = (V, E)$.

Введём ещё несколько терминов.

Определение 49. ***Точка сочленения (junction point)*** *— такая вершина графа, удаление которой вместе с исходящими из неё рёбрами, приводит к увеличению числа компонент связности графа.*

В литературе существует много вариантов названия данного понятия — *точка раздела*, *точка артикуляции*, *разделяющая вершина*. На английском

языке, наряду с термином *junction point* применяются также термины *cut vertex*, *articulation vertex*

Определение 50. ***Блок*** — *связный непустой граф, не содержащий точек сочленения. Другое название —* ***компонент двусвязности****.*

Определение 51. ***Мост (bridge)*** — *такое ребро графа, удаление которого приводит к увеличению числа компонент связности графа.*

Достаточно очевидно, что для любого моста (u, v) вершины u и v являются точками сочленения.

Для обоснования корректности приводимых далее алгоритмов необходимо ввести ещё одно понятие.

Определение 52. *Пара различных рёбер e_1 и e_2 удовлетворяют* ***отношению R на множестве рёбер E*** *если существует простой цикл, который содержит эти рёбра.*

Если мы добавим к этому определению ещё одно: **любое ребро e находится в отношении R с самим собой**, то математики узнают в отношении R *отношение эквивалентности*. Мы не будем доказывать этот факт.

Важно, что после установления такого отношения R, множество всех рёбер графа E можно разбить на непересекающиеся множества таким образом, что каждое ребро попадает ровно в одно из них. Математики скажут, что множество E разбито на *классы эквивалентности* относительно R.

8.4.1 Поиск точек сочленения

Свойства точек сочленения можно вывести из понятия отношения R. Давайте рассмотрим граф на рис. 8.186.

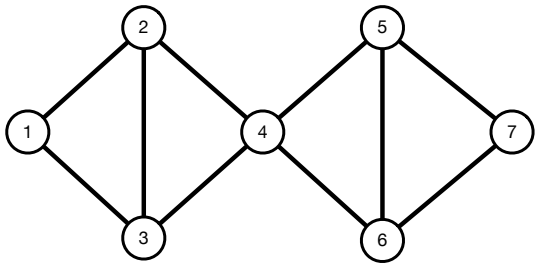


Рис. 8.186. Точки сочленения

Попытаемся разбить множество рёбер на классы эквивалентности по отношению R. Для каждого из рёбер попробуем определить те, которые входят в его класс эквивалентности. Для простоты будем выкрашивать рёбра в один и тот же цвет, если они принадлежат одному классу, и в разный — если они принадлежат разным классам.

Рассмотрим, например, ребро (1,2), пусть его цвет будет красным. Рёбра (1,2), (1,3) и (2,3) образуют простой цикл, значит они — тоже красные. Красным цветом выкрасим рёбра (2,4) и (3,4). А вот с ребром (4,5) простых циклов, содержащих ребро (1,2) (как, впрочем, и любых других красных рёбер) не существует. Выкрашиваем это ребро в зелёный цвет и соответственно ему в этот же цвет выкрашиваем все рёбра, с которыми оно образует простые циклы.

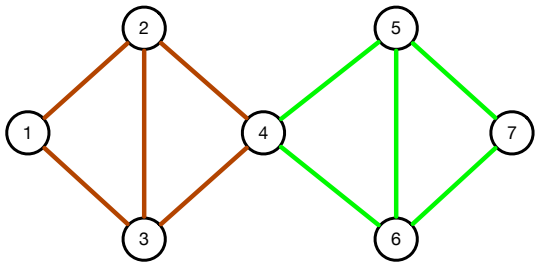


Рис. 8.187. Точки сочленения, раскраска графа

Обратите внимание, что вершина 4 — единственная, к которой подходят

рёбра одного цвета. Удаление этой вершины и рёбер, относящихся к ней, приведёт к графу, который уже имеет две компоненты связности.

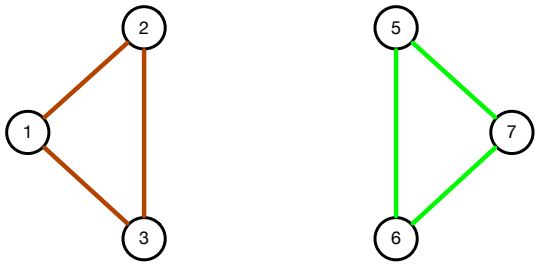


Рис. 8.188. Точки сочленения, распавшийся граф

Искать точку сочленения, удаляя вершину и проверяя число компонентов связности получившегося графа, конечно же можно. Но сложность такого алгоритма слишком велика — $O(|V| \cdot (|V| + |E|))$. К счастью, есть и более эффективные алгоритмы. Используем тот факт, что каждая точка сочленения принадлежит минимум двум классам эквивалентности.

Основным инструментом поиска таких вершин будет алгоритм `DFS`. Для простоты будем полагать, что рассматриваемые далее графы — связные. Это значит, что процедура поиска в глубину, с какого бы узла она не была бы запущена, даст нам дерево поиска. Для рассматриваемого графа оно может быть таким (рис. 8.189):

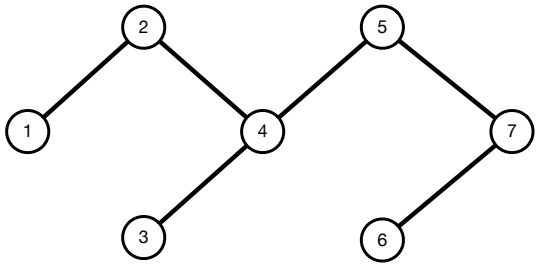


Рис. 8.189. Дерево поиска в графе

Рёбра из дерева поиска назовём *прямыми*, а оставшиеся — *обратными*.

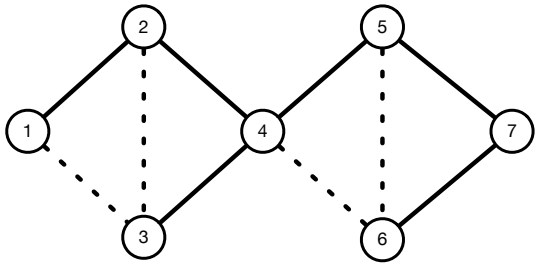


Рис. 8.190. Дерево поиска в графе — прямые и обратные рёбра

Если при обходе алгоритмом `DFS` мы попытаемся попасть в уже выкрашенную в серый или чёрный цвет вершину, то мы обнаружили цикл. Наряду с уже известными нам массивами d (время захода в вершину) и f (время выхода из вершины) добавим массив l, который в вершине u будет принимать минимальное значение из всех $d[v]$, где v пробегает по концам всех обратных рёбер, начинающихся в поддеревьях с корнем в u. Возможно, это звучит сложно, но для реализации нам не потребуется добавочной работы и рекурсия всё сделает за нас.

Вызов функции `DFS-junction` с первым аргументом, равным номеру корневой вершины обхода и вторым аргументом — любым числом, меньшим нуля, приведёт к построению дерева обхода и регистрации всех точек сочленения. В переменной `children` фиксируется количество прямых рёбер, в которые запускается обход дерева в данной вершине, назовём их *дочерними*.

Алгоритм опирается на два факта:

1. Если в корневой вершине имеется более одного дочернего ребра, то она является точкой сочленения.

2. Если вершина u — не корневая в данном обходе и если хотя бы для одной из её дочерних вершин v $l[v] \geqslant d[u]$, то u — точка сочленения.

Отсюда и вытекает алгоритм `DFS-junction`.

```
 1: procedure DFS-JUNCTION(u : Vertex, p : Vertex)
 2:     c[u] ← grey
 3:     time ← time + 1
 4:     d[u] ← time
 5:     l[u] ← time
 6:     children ← 0
 7:     for all v ∈ Adj[u] do
 8:         if v ≠ p then
 9:             if c[v] = white then
10:                 DFS-junction(v,u)
11:                 children ← children + 1
12:                 l[u] ← min(l[u], l[v])
13:                 if p ⩾ 0 & l[v] ⩾ d[u] then
14:                     Register u as junction point
15:                 end if
16:             else
17:                 l[u] ← min(l[u], d[v])
18:             end if
19:         end if
20:     end for
21:     if p < 0 & children ⩾ 2 then
22:         Register u as junction point
23:     end if
24:     c[u] ← black
25:     time ← time + 1
26:     f[u] ← time
27: end procedure
```

8.4.2 Поиск мостов

Мост по нашему определению по-совместительству является и компонентом двусвязности. Дадим несколько утверждений без доказательства:

- Ребро тогда и только тогда является мостом, когда оно не принадлежит ни одному простому циклу.
- Каждый класс эквивалентности по отношению R либо компонента двусвязности, либо мост.
- Обратные рёбра мостами являться не могут.

Это утверждения дают нам возможность придумать алгоритм поиска всех мостов в графе. Допустим, что прямое ребро $e = (u, v)$ является мостом в связном графе G, причём v — дочерняя вершина u. Тогда при его удалении граф распадётся на две компоненты связности, в одной из которых останется вершина u, а другая будет определяться поддеревом дерева обхода, начинающимся в вершине v. Это означает, что из v не имеется обратных рёбер в первую компоненту связности. Вспомним, что массив l в обходе `DFS-junction` в алгоритме нахождения точек связности уже содержит нужную нам информацию.

```
 1: procedure DFS-BRIDGE(u : Vertex)
 2:     c[u] ← grey
 3:     time ← time + 1
 4:     d[u] ← time
 5:     l[u] ← time
 6:     for all v ∈ Adj[u] do
 7:         if c[v] = white then
 8:             DFS-bridge(v)
 9:             l[u] ← min(l[u], l[v])
10:             if l[v] > d[u]  &  l[v] ⩾ d[v] then
11:                 Register (u, v) as bridge
12:             end if
13:         else
14:             l[u] ← min(l[u], d[v])
15:         end if
16:     end for
17:     c[u] ← black
18:     time ← time + 1
19:     f[u] ← time
20: end procedure
```

8.5 Остовные деревья

Давайте введём ещё несколько терминов.

С точки зрения теории графов **дерево** есть ациклический связный граф.

Множество деревьев называется **лесом (forest)** или **бором**.

Остовное дерево связного графа — подграф, который содержит все вершины графа и представляет собой дерево.

Остовный лес графа — лес, содержащий все вершины графа.

Построение остовных деревьев — одна из основных задач в компьютерных сетях. Решение таких задач позволит оптимально спланировать маршрут от одного узла сети до других. Проблема состоит в том, что для некоторого типа узлов в передаче сообщений недопустимо иметь несколько возможных маршрутов. Например, если компьютер соединён с маршрутизатором по `Wi-Fi` и `Ethernet` одновременно, то в некоторых операционных системах сообщения от компьютера до маршрутизатора не будут доходить из-за наличия цикла. Построение остовного дерева — избавление от циклов в графе.

Остовных деревьев для одного графа может быть много. Например, для графа

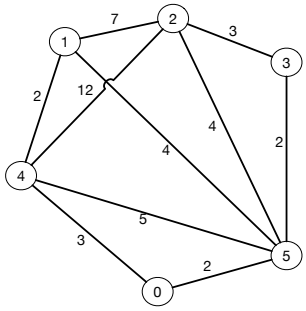


Рис. 8.191. Остовные деревья: граф

остовным деревом может быть и такое:

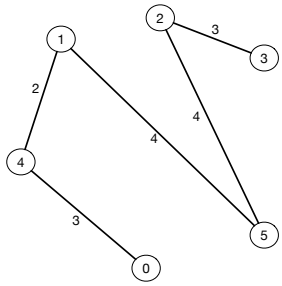


Рис. 8.192. Остовные деревья: дерево 1

и такое:

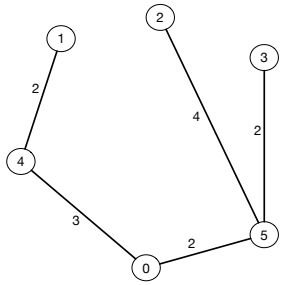


Рис. 8.193. Остовные деревья: дерево 2

Чаще всего задача формулируется как нахождение минимального остовного дерева.

Определение 53. *MST (Minimal Spanning Tree) — минимальное остовное дерево взвешенного графа — есть остовное дерево, вес которого (сумма его всех рёбер) не превосходит вес любого другого остовного дерева.*

Именно минимальные остовные деревья больше всего интересуют проектировщиков сетей.

Определение 54. ***Сечение графа*** *— разбиение множества вершин графа на два непересекающихся подмножества.*

Определение 55. ***Перекрёстное ребро*** *— ребро, соединяющее вершину одного множества с вершиной другого множества.*

8.5.1 Свойства MST

Лемма. Если T — произвольное остовное дерево, то добавление любого ребра e между двумя вершинами u и v создаёт цикл, содержащий вершины u, v и ребро e.

Лемма. При любом сечении графа каждое минимальное перекрёстное ребро принадлежит некоторому MST-дереву и каждое MST-дерево содержит перекрёстное ребро.

Доказательство от противного. Пусть e — минимальное перекрёстное ребро, не принадлежащее ни одному MST, и пусть T — MST дерево, не содержащее e. Добавим e в T. В этом графе есть цикл, содержащий e, и он содержит ребро e', с весом, не меньшим e. Если удалить e', то получится остовное дерево не большего веса, что противоречит условию минимальности T или предположению, что e не содержится в T.

Следствие. Каждое ребро дерева MST есть минимальное перекрёстное ребро, определяемое вершинами поддеревьев, соединённых этим ребром.

Это следствие служит основой одного из популярных алгоритмов нахождения MST — алгоритма Прима.

8.5.2 Алгоритм Прима

Алгоритм Прима строит MST, используя в каждый момент времени сечение графа на два подграфа. Один подграф, вначале состоящий из одной корневой вершины, содержит те вершины, которые входят в уже построенный фрагмент MST. Другой подграф содержит все оставшиеся вершины. Алгоритм шаг за шагом добавляет по одной вершине в первое множество, убирая её из второго. Как только новой подходящей вершины найти не удаётся, алгоритм завершается.

1. Выбираем произвольную вершину. Это — `MST` дерево, состоящее из одной древесной вершины.
2. Выбираем минимальное перекрёстное ребро между `MST` любой из вершин, входящих в древесное множество, и любой из вершин, входящих в недревесное множество.
3. Повторяем операцию до тех пор, пока все вершины не окажутся в дереве.

Применим алгоритм к исходному графу, изображённому на рис. 8.194.

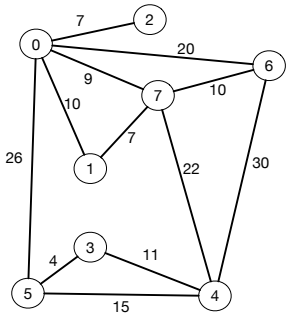


Рис. 8.194. Алгоритм Прима: подопытный граф

Вершина `0` — корневая. Переводим её в `MST`. Проверяем все веса рёбер, ведущих из `MST` в не-`MST`.

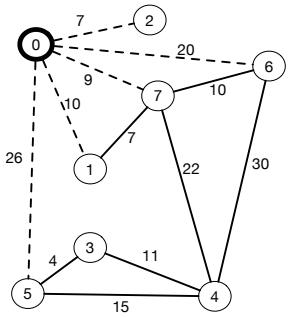


Рис. 8.195. Алгоритм Прима: отмечены все рёбра из `MST` в не-`MST`

`(0-2)` — самое лёгкое ребро. Переводим вершину `2` и ребро `(0-2)` в `MST`.

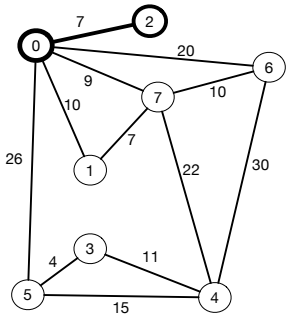


Рис. 8.196. Алгоритм Прима: вершина 2 переведена в `MST`

Отмечаем все рёбра из `MST` в не-`MST`.

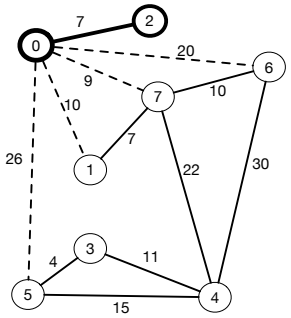


Рис. 8.197. Алгоритм Прима: отмечены все рёбра из `MST` в не `MST`

Переносим вершину `7` и ребро `(0-7)` в `MST`.

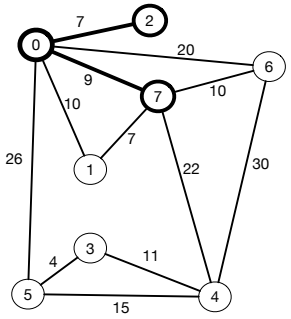


Рис. 8.198. Алгоритм Прима: вершина 7 перенесена в `MST`

Отмечаем все рёбра из `MST` в не-`MST`.

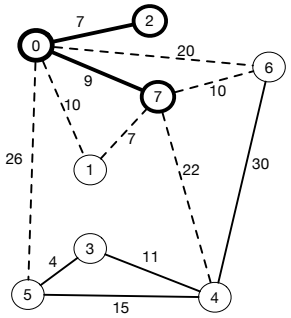


Рис. 8.199. Алгоритм Прима: отмечены все рёбра из `MST` в не `MST`

Переносим вершину `1` и ребро `(1-7)` в `MST`.

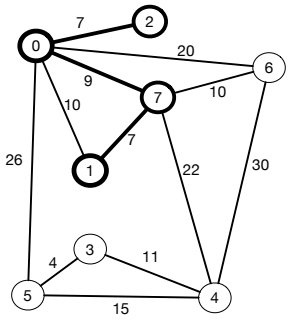


Рис. 8.200. Алгоритм Прима: вершина 1 перенесена в `MST`

Отмечаем все рёбра из `MST` в не-`MST`.

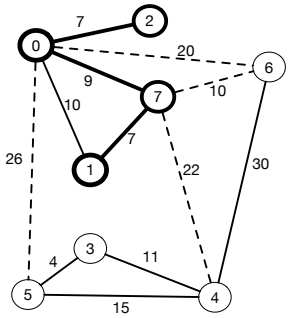


Рис. 8.201. Алгоритм Прима: отмечены все рёбра из `MST` в не `MST`

Переносим вершину `6` и ребро `(7-6)` в `MST`.

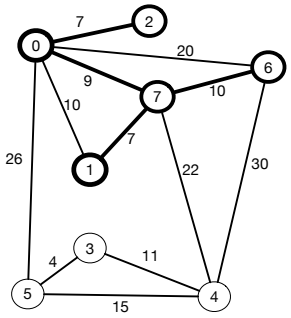


Рис. 8.202. Алгоритм Прима: вершина 6 перенесена в `MST`

Продолжая подобным образом, приходим к заключительной ситуации, когда все вершины оказываются в `MST` — алгоритм завершён.

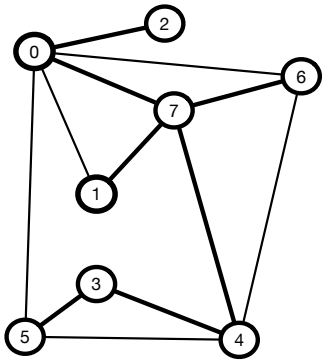


Рис. 8.203. Алгоритм Прима: итог

Сложность алгоритма $O(|V|^2)$.

Если попытаться реализовать данный алгоритм именно в том виде, как мы его разбирали, то окажется, что он не очень эффективен. При поиске очередного ребра из `MST` в не-`MST` мы забываем про те рёбра, которые уже проверяли. Задача упростится, если мы введём понятие **накопитель**.

Накопитель — структура данных, которая содержит множество рёбер-кандидатов.

Более эффективна будет следующая реализация алгоритма Прима, использующая накопитель.

1. Выбираем произвольную вершину. Это — `MST` дерево, состоящее из одной вершины. Делаем вершину текущей.
2. Помещаем в накопитель все рёбра, которые ведут из этой вершины в не-`MST` узлы. Если в какой-либо из узлов уже ведёт ребро с большей длиной, заменяем его ребром с меньшей длиной.
3. Выбираем ребро с минимальным весом из накопителя.
4. Повторяем операцию до тех пор, пока все вершины не окажутся в дереве.

Алгоритм Прима — обобщение поиска на графе. Если представить накопитель как приоритетную очередь с операциями «добавить элемент», «извлечь минимальное» и «увеличить приоритет», то сложность алгоритма составит $O(|E| \log |V|)$. Действительно, операции с приоритетной очередью исполняются за $O(\log N)$, а в нашем случае N не превосходит $|V|$.

Такое обобщение поиска на графе носит название `PFS` (Priority First Search) — поиск по приоритету.

8.5.3 Алгоритм Краскала

Один из самых старых алгоритмов на графах — это алгоритм Краскала, который известен с 1956 года, но затем был забыт как непрактичный до момента изобретения подходящей структуры данных и связанного с ней вспомогательного алгоритма.

Сам алгоритм — один из наиболее изящных в теории графов. Предварительным условием алгоритма является связность графа. Заключается он в следующем.

1. Создаётся число непересекающихся множеств по количеству вершин — и каждая вершина составляет своё множество.
2. Множество `MST` вначале пусто.
3. Среди всех рёбер, не принадлежащих `MST`, выбирается самое короткое из всех рёбер, которые не образуют цикл. Это значит, что вершины ребра должны принадлежать различным множествам.
4. Выбранное ребро добавляется к множеству `MST`.
5. Множества, которым принадлежат вершины выбранного ребра, сливаются в единое.
6. Если размер множества `MST` стал равен $|V| - 1$, то алгоритм завершён, иначе отправляемся к пункту 3.

Проиллюстрируем алгоритм на том же графе, который применялся в алгоритме Прима.

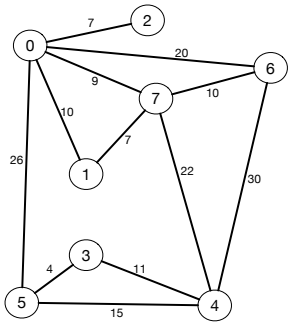


Рис. 8.204. Алгоритм Краскала: подопытный граф

Создадим список рёбер, упорядоченный по возрастанию,

i	3	0	1	0	0	6	3	4	0	0	4
j	5	2	7	7	1	7	4	5	6	5	6
W_{ij}	4	7	7	9	10	10	11	15	22	26	30

а также таблицу, показывающую, какому множеству принадлежит вершина:

V_i	0	1	2	3	4	5	6	7
p	0	1	2	3	4	5	6	7

На первой итерации самое короткое ребро — $(3, 5)$ с длиной 4. Так как вершины 3 и 5 принадлежат разным множествам, отправляем ребро в множество `MST` и объединяем множества. Для простоты будем полагать пока, что при объединении множеств они получают меньший из номеров.

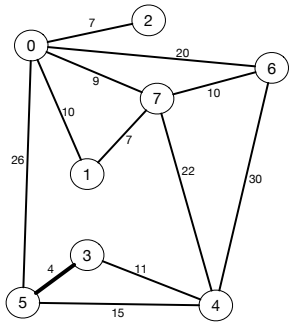


Рис. 8.205. Алгоритм Краскала: добавлено первое ребро

V_i	0	1	2	3	4	5	6	7
p	0	1	2	3	4	**3**	6	7

На второй итерации среди оставшихся рёбер имеются два подходящих с одинаковым весом:

i	0	1	0	0	6	3	4	0	0	4
j	2	7	7	1	7	4	5	6	5	6
W_{ij}	7	7	9	10	10	11	15	22	26	30

Мы можем из них выбрать произвольное. В самом деле, что произойдёт после использования первого из них? Если добавление первого ребра не помешает добавлению второго, то, очевидно, всё в порядке. Если же оно помешает, то есть после добавления первого ребра добавление второго создаст цикл, то очевидно, что удаление любого из рёбер решит проблему, и общий вес дерева останется неизменным.

Выберем произвольным образом ребро (0,2) и поместим вершину 2 в множество номер 0.

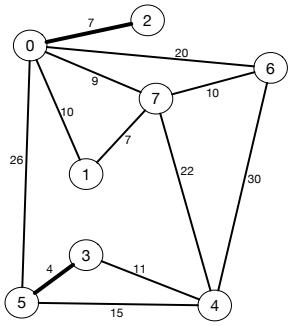


Рис. 8.206. Алгоритм Краскала: добавлено второе ребро

V_i	0	1	2	3	4	5	6	7
p	0	1	**0**	3	4	3	6	7

Ребро (1,7) привело к слиянию множеств 1 и 7.

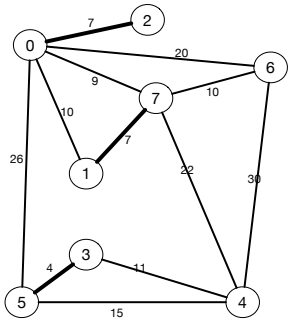


Рис. 8.207. Алгоритм Краскала: добавлено третье ребро

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	**1**

Следующий этап — более интересный. Самым коротким ребром из оставшихся оказалось ребро (0,7).

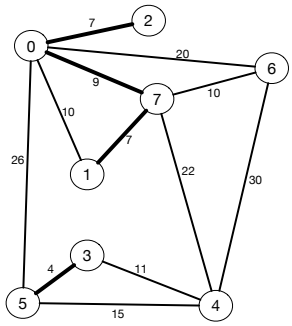


Рис. 8.208. Алгоритм Краскала: добавлено четвёртое ребро

Нам нужно слить два множества — одно, содержащее $\{0, 2\}$, и другое — содержащее $\{1, 7\}$. Предположим, что новое множество получит номер 0. Что теперь делать с массивом p, в котором записаны номера множеств для вершин? Прямолинейная реализация алгоритма говорит нам, что нужно найти в массиве p все единицы (номер второго множества) и заменить их на нули (номер того множества, куда переходят элементы первого). К счастью, для эффективного решения этой задачи существует подходящая структура данных: *система непересекающихся множеств*. В англоязычной литературе она носит названия `Union-Find` или `Disjoint Set Union`, `DSU`. Именно изобретение этой структуры данных и вдохнуло новую жизнь в алгоритм Краскала. Немного отступим от разбора примера, чтобы её пояснить.

8.5.4 Система непересекающихся множеств

Абстракция `DSU` реализует три операции:

- `create(n)` — создать набор множеств из n элементов. Это мы и сделали, когда создали массив p при решении задачи.
- `find_root(x)`. Вместо помещения номера множества в каждую из вершин множества (что, как мы видели, требует поиска по всему масси-

ву и достаточно неэффективно), давайте введём понятие *представителя* множества. В нашем случае массив p пока имеет такой вид:

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	1

Для вершины 7, например, массив p хранит число 1 — номер множества, он же и *представитель* множества 1. Если мы, не глядя, для слияния множеств `0,2` и `1,7` поместим в `p[7]` число 0, то в дальнейшем получим проблемы: для вершины 1 представителем останется 1, что, очевидно, неверно, так как после слияния вершина 1 должна принадлежать множеству 0. Сейчас `p[7]=1` — и это означает, что и у седьмой, и у первой вершины представители одинаковые. Если номер вершины совпадает с номером представителя, то, очевидно, что для этой вершины в массив `p` ничего не было записано в процессе прохождения алгоритма, то есть эта вершина есть корень дерева. Таким образом, после слияния нам нужно просто заменить всех родителей вершины (поэтому, кстати, мы и назвали массив `p` — parents) на нового представителя. Это делается изящным рекурсивным алгоритмом:

```
int find_root(int r) {
   if (p[r] == r) return r; // Тривиальный случай
   return p[r] = find_root(p[r]); Рекурсивный случай
}
```

Убедитесь сами, что этот алгоритм действительно работает.

- `merge(l,r)` — сливает два множества. Для сохранения корректности алгоритма вполне достаточно любого из присвоений: `p[l] = r` или `p[r] = l`. Всю дальнейшую корректировку родителей в дальнейшем сделает метод `find_root`.

  На практике для повышения эффективности используются несколько приёмов. Одним из них является использование ещё одного массива, хранящего длины деревьев: слияние производится к более короткому дереву. Вторым — случайный выбор дерева-приёмника.

```
void merge(int l, int r) {
   l = find_root(l);
   r = find_root(r);
   if (rand() % 1) {
      p[l] = r;
   } else {
      p[r] = l;
```

```
    }
  }
```

Обратите внимание на то, что внутри операции слияния имеется операция поиска, которая заменяет аргументы значениями корней их деревьев!

Вернёмся к нашей задаче. Для определения, каким деревьям принадлежат концы ребра `(0,7)` мы дважды вызовем `find_root`. `find_root(0)` проверит, что в элементе `p[0]` находится 0, и вернёт его, как номер множества.

А вот `find_root(7)` сначала убедится, что в `p[7]` лежит 1, и вызовет `find_root(1)`, после чего, возможно, заменит `p[7]` на 1.

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	**0**

Теперь мы убедились, что концы ребра `0` принадлежат разным множествам, и сливаем множества, вызвав `merge(0,7)`. Первое, что сделает операция `merge` — заменит свои аргументы, 0 и 7, корнями деревьев, которым принадлежат 0 и 7, то есть, 0 и 1 соответственно. После этого в `p[1]` помещается 0 — и деревья слиты. Обратите внимание на то, что в `p[7]` всё ещё находится 1.

V_i	0	1	2	3	4	5	6	7
p	0	`0`	0	3	4	3	6	1

Следующее ребро — `6,7` (рис. 8.209).

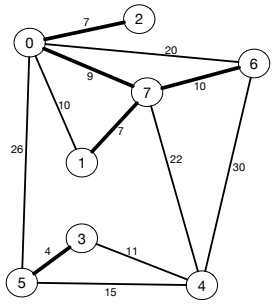


Рис. 8.209. Алгоритм Краскала: добавлено пятое ребро

Поиск корня дерева для седьмой вершины приведёт к тому, что для вершины 7 будет установлен корень 0 — и поэтому `p[7]` станет равным 0. Это же значение будет присвоено и `p[6]`. Как мы видим, алгоритм не спешит присваивать правильные значения корней вершинам, откладывая это действие «на потом». И действительно, для некоторых вершин не рекурсивного поиска в дальнейшем не потребуется, так что система непересекающихся множеств относится не к жадным алгоритмам, а к *ленивым* (хорошо сформировавшийся лентяй все дела откладывает «на потом»).

V_i	0	1	2	3	4	5	6	7
p	0	0	0	3	4	3	0	0

На следующем этапе ребро `(3,4)` окажется самым коротким.

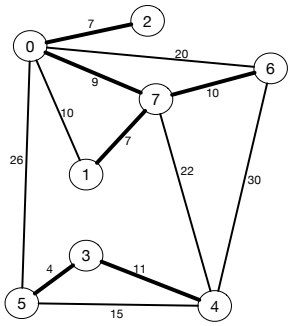


Рис. 8.210. Алгоритм Краскала: добавлено шестое ребро

V_i	0	1	2	3	4	5	6	7
p	0	0	0	3	3	3	0	0

Поиск множеств, которым принадлежат концы следующих рёбер `(4,5)` и `(0,6)`, покажет, что эти рёбра принадлежат одному множеству, а вот ребро `(4,7)` подходит:

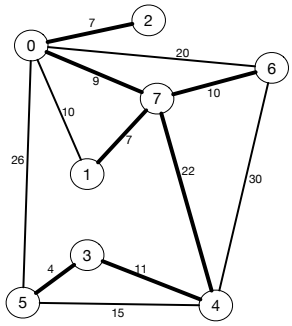


Рис. 8.211. Алгоритм Краскала: добавлено седьмое, последнее ребро

V_i	0	1	2	3	4	5	6	7
p	0	0	0	3	0	3	0	0

На этом алгоритм завершается, так как количество рёбер в множестве `MST` достигло $7 = N - 1$.

Как оценить сложность алгоритма? Первая часть алгоритма — сортировка рёбер. Сложность этой операции $O(|E| \log |E|)$. А как оценить сложность операции поиска и слияния? В 1984 году Tarjan (в русскоязычной литературе он именуется и Тарджаном, и Тарьяном) доказал, используя функцию Аккермана, что операция поиска в `DSU` имеет амортизированную сложность `O(1)`. Таким образом, сложность всего алгоритма Краскала и есть $O(|E| \log |E|)$. Для достаточно разреженных графов он обычно быстрее алгоритма Прима, для заполненных — наоборот.

8.6 Алгоритм Дейкстры

8.6.1 Дерево кратчайших путей

Мы уже умеем находить кратчайшие маршруты от выбранной вершины в графе до других поиском `BFS`, но только для отдельной категории графов, невзвешенных. К сожалению, на взвешенных графах этот поиск даёт осечку — и приходится изобретать другие алгоритмы.

Определение 56. ***Дерево кратчайших путей*** *(SPT) для s — подграф, содержащий s и все вершины, достижимые из s, образующий направленное поддерево с корнем в s, где каждый путь от вершины s до вершины u является кратчайшим из всех возможных путей.*

Задача. Пусть заданы граф G и вершина s. Построить SPT для s.

Решение задачи. Алгоритм Дейкстры строит SPT, определяя длины кратчайших путей от заданной вершины до остальных. Веса рёбер могут принимать произвольные неотрицательные значения. Если в графе есть отрицательные рёбра, но нет отрицательных циклов, то можно применить алгоритм Форда-Беллмана, о нём — далее.

Алгоритм Дейкстры — ещё один вариант класса жадных алгоритмов PFS (как мы помним, к этому классу алгоритмов относится и алгоритм Прима).

1. В SPT заносится корневой узел (исток).
2. На каждом шаге в SPT добавляется одно ребро, которое формирует кратчайший путь из истока в не-SPT.
3. Вершины заносятся в SPT в порядке их расстояния по SPT от начальной вершины.

Обозначим за U множество вершин, принадлежащих уже найденному оптимальному множеству. На каждом шаге мы будем исследовать вершины, не принадлежащие U, и одну из них добавлять в U. Жадная стратегия заключается в том, что после того, как вершина попала в оптимальное множество, она его не покидает.

В начале алгоритма оптимальное множество U состоит из одного элемента — вершины s.

Длины кратчайших путей до вершин множества обозначим, как $d(s, v), v \in U$.

Основной шаг — нахождение среди вершин, смежных с U, такой вершины $u, u \notin U$, что достигается минимум

$$\min_{v \in U, u \notin U} d(s, v) + w(v, u).$$

После нахождения этой вершины она добавляется в множество $U : U \leftarrow U \cup \{u\}$. Эта операция повторяется до тех пор, пока множество U может пополняться.

В реализации алгоритма используются переменные:

- $d[u]$ — длина кратчайшего пути из вершины s до вершины u;
- $\pi[u]$ — предшественник u в кратчайшем пути от s;
- $w(u, v)$ — вес пути из u в v (длина ребра, вес ребра, метрика пути);

- Q — приоритетная по значению d очередь узлов на обработку;
- U — множество вершин с уже известным финальным расстоянием.

Сам алгоритм формализуется следующим образом:

```
 1: procedure DIJKSTRA(G : Graph; w : weights; s : Vertex)
 2:     for all v ∈ V do
 3:         d[v] ← ∞
 4:         π[v] ← nil
 5:     end for
 6:     d[s] ← 0
 7:     U ← ∅
 8:     Q ← V
 9:     while Q ≠ ∅ do
10:         u ← Q.extractMin()
11:         U ← U ∪ {u}
12:         for all v ∈ Adj[u], v ∉ U do
13:             Relax(u,v)
14:         end for
15:     end while
16: end procedure
```

```
 1: procedure RELAX(u, v : Vertex)
 2:     if d[v] > d[u] + w(u, v) then
 3:         d[v] = d[u] + w(u, v)
 4:         π[v] ← u
 5:     end if
 6: end procedure
```

Здесь мы сталкиваемся с новым понятием — *релаксацией*. В данном алгоритме релаксация — корректировка кратчайшего пути между двумя вершинами, если находится путь через какую-либо промежуточную вершину.

Попробуем посмотреть, как изменяются данные при прохождении алгоритма для какого-то исходного графа (рис. 8.212).

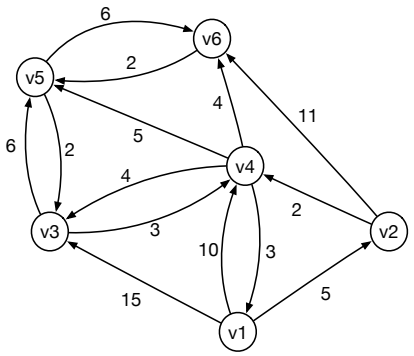


Рис. 8.212. Алгоритм Дейкстры: подопытный граф

В начале алгоритма `SPT` содержит одну вершину, v_1.

Просмотр смежных вершин добавляет в накопитель три вершины — v_2, v_3 и v_4 (рис. 8.213).

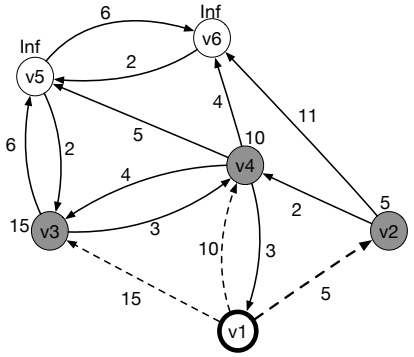


Рис. 8.213. Алгоритм Дейкстры: в накопитель добавлены вершины v_2, v_3 и v_4

Соответственно, элементы массива d_2, d_3 и d_4 становятся равны $5, 15$ и 10. Обратите внимание, как это происходит, например, для d_2: так как d_2 пока равно ∞, и $d_1 + w_{12} < d_2$, то происходит релаксация пути — и d_2 становится равно $d_1 + w_{12}$.

Далее из накопителя выбирается элемент с наименьшим значением d, это — второй элемент. Для всех исходящих связей проверяется возможность релаксации. Так как $d_2 + w_{24} < d_4$, то d_4 становится равным $d_2 + w_{24}$, а это значит, что старый наилучший маршрут $(1 \to 4)$ заменён на $(1 \to 2 \to 4)$.

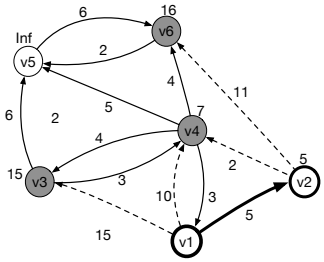


Рис. 8.214. Алгоритм Дейкстры: в накопитель добавлена вершина v_6, и произошла релаксация маршрута до v_4

Выбран узел v_3. Все смежные вершины корректируют значения d.

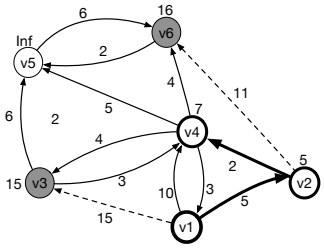


Рис. 8.215. Алгоритм Дейкстры: проведены все релаксации после добавления v_6

В накопитель отправляется v_5. Релаксация: $(1 \to 2 \to 6)$ заменено на $(1 \to 2 \to 4 \to 6)$, $(1 \to 3)$ на $(1 \to 2 \to 4 \to 3)$.

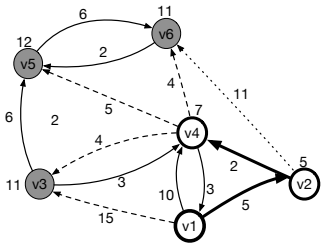


Рис. 8.216. Алгоритм Дейкстры: вершина v_5 отправляется в накопитель и проводятся все релаксации

Заключительное состояние:

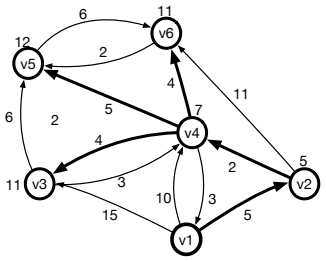


Рис. 8.217. Алгоритм Дейкстры: итог

8.6.2 Алгоритм Дейкстры: сложность

Для того, чтобы поместить во множество U все вершины, требуется сделать $|V| - 1$ шаг. На каждом шаге происходит корректировка расстояния до смежных вершин $O(|E_i|)$ и выбор минимального элемента из накопителя

$O(\log |V|)$. Общая сложность алгоритма сильно зависит от структур данных, применяемых при работе с накопителем. Для изученных нами структур данных она составляет

$$T = \sum_{i=1,|V|} |E_i| + |E| \cdot \log |V| = O(|E| \cdot \log |V|).$$

С помощью алгоритма Дейкстры можно найти расстояния от конкретной вершины до всех достижимых. Применив его для всех вершин, можно составить таблицу кратчайших расстояний между каждой парой вершин. Для насыщенного графа классический вариант алгоритма Дейкстры, который мы рассматривали, будет иметь сложность $O(|V|^2 \log |V|)$. Соответственно, построение таблицы всех кратчайших расстояний между парами будет иметь сложность $O(|V|^3 \log |V|.)$ Много ли это? Оказывается, можно и быстрее, существует более быстрый алгоритм, имеющий сложность $O(|V|^3)$.

8.7 Алгоритм Флойда-Уоршалла

Этот алгоритм наряду с таблицей кратчайших расстояний между любыми парами вершин позволяет обнаруживать для каждой пары (u, v) ту вершину p, смежную с u, через которую проходит кратчайший путь из u в v.

Сам алгоритм известен с 1962 года. Для его исполнения наиболее удобно использовать представление графа матрицей смежности, в которой число, находящееся в i-й строке и j-м столбце, есть вес связи между вершинами i и j. Для корректной работы алгоритма модифицируем матрицу C таким образом, что если вершина v не является соседней для u, то $C_{uv} = +\infty$. После исполнения алгоритма матрица D будет содержать в элементе C_{uv} вес кратчайшего пути из u в v. Хорошим свойством алгоритма является то, что допустимы пути с отрицательным весом, а вот циклы с отрицательным весом недопустимы, но могут быть обнаружены.

Сам алгоритм может быть описан в рекуррентной форме как

$$D_{ij}^{(k)} = \begin{cases} C_{ij}, & \text{если } k = 0, \\ \min\left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right), & \text{если } k \geqslant 1 \end{cases}$$

Это — задача динамического программирования, решаемая восходящим методом.

Код этого алгоритма исключительно прост.

```
void FloydWarshall(long long **d, int n) {
for (int i = 0; i < n; i++) {
  for (int s = 0; s < n; s++) {
    for(int t = 0; t < n; t++) {
      if (s != t) { // Пропустим петлю
        if (d[s][t] > d[s][i] + d[i][t]) {
          d[s][t] = d[s][i] + d[i][t];
        }
      }
    }
  }
}
}
```

Нулевая итерация состоит в том, что заключительная матрица D заполняется элементами матрицы C. Инвариант алгоритма: после итерации n каждый элемент D_{ij} матрицы D содержит значение кратчайшего маршрута из i в j с промежуточными маршрутами, проходящими через вершину n. Это утверждение верно и после нулевой итерации, так как нумерация вершин в алгоритме начинается с единицы[16].

Для того, чтобы посмотреть, как работает данный алгоритм, воспользуемся уже знакомым графом с рис. 8.212.

Матрица D после нулевой итерации имеет вид:

$D^{(0)} =$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	10	∞	∞
V_2	∞	0	∞	2	∞	11
V_3	∞	∞	0	3	6	∞
V_4	3	∞	4	0	5	4
V_5	∞	∞	2	∞	0	6
V_6	∞	∞	∞	∞	5	0

Начальная матрица $D^{(0)}$ содержит метрики всех наилучших маршрутов единичной длины. Каждая следующая итерация алгоритма добавляет в матрицу $B^{(i)}$ элементы, связанные с маршрутами, проходящими через вершину i.

[16]Заметьте, что в приведённом коде нумерация вершин ведётся от нуля.

$D^{(1)} =$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	10	∞	∞
V_2	∞	0	∞	2	∞	11
V_3	∞	∞	0	3	6	∞
V_4	3	**8**	4	0	5	4
V_5	∞	∞	2	∞	0	6
V_6	∞	∞	∞	∞	5	0

После первой итерации произошла одна релаксация: элемент D_{42} получил значение 8. Жирным шрифтом помечены изменившиеся элементы таблицы.

$D^{(2)} =$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	**7**	∞	**16**
V_2	∞	0	∞	2	∞	11
V_3	∞	∞	0	3	6	∞
V_4	3	**8**	4	0	5	4
V_5	∞	∞	2	∞	0	6
V_6	∞	∞	∞	∞	5	0

Вторая итерация пробует провести операцию релаксации с использованием вершины V_2 в качестве промежуточной. После неё изменился элемент D_{14}. Он стал равен 7. И действительно, произошла релаксация: $(1 \to 4)$ заменено на $(1 \to 2 \to 4)$. Появился маршрут $(1 \to 2 \to 6)$.

$D^{(3)} =$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	7	**21**	16
V_2	∞	0	∞	2	∞	11
V_3	∞	∞	0	3	6	∞
V_4	3	8	4	0	5	4
V_5	∞	∞	2	**5**	0	6
V_6	∞	∞	∞	∞	5	0

Третья итерация добавила маршруты $(1 \to 3 \to 5)$, вес которого равен 21 и маршрут $(5 \to 3 \to 4)$ с весом 5.

$D^{(4)} =$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	7	12	16
V_2	**5**	0	**6**	2	**7**	11
V_3	**6**	**11**	0	3	6	**7**
V_4	3	8	4	0	5	4
V_5	**8**	**13**	2	5	0	6
V_6	∞	∞	∞	∞	5	0

Четвёртая итерация добавила много новых маршрутов, так как она проводила релаксацию через оживлённый перекрёсток v_4.

$D^{(4)} =$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	7	12	16
V_2	5	0	6	2	7	11
V_3	6	11	0	3	6	7
V_4	3	8	4	0	5	4
V_5	8	13	2	5	0	6
V_6	**13**	**18**	**7**	**10**	5	0

Пятая, последняя, итерация, завершает алгоритм. Заключительная матрица D содержит кратчайшие расстояния между вершинами. Если элемент D_{ij} матрицы равен бесконечности, то вершина j недостижима из вершины i.

Для определения того, имеет ли граф отрицательный цикл, достаточно повторить алгоритм, используя в качестве начальной матрицы D матрицу, полученную после $|V| - 1$ итерации. Если в какой-то момент операция релаксации завершится успехом, то это даст нам два факта: граф содержит отрицательный цикл — и вершины графа, участвующие в успешной операции релаксации, содержатся в этом цикле.

8.8 Домашние задания

Задача 36. Зелье

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	256 мегабайт

Злой маг Крокобобр варит зелье. У него есть большая колба, которую можно ставить на огонь и две колбы поменьше, которые огня не выдержат. В большой колбе налита компонента зелья, которую нужно подогреть на огне, маленькие колбы пусты. Ёмкость большой колбы магу Крокобобру известна — N миллилитров, маленьких — M и K миллилитров. Смесь можно переливать из любой колбы в любую, если выполняется одно из условий: либо после переливания одна из колб становится пустой, либо одна из колб становится полной, частичные переливания недопустимы.

Требуется ровно L миллилитров смеси в большой колбе. Помогите Крокобобру определить, сколько переливаний он должен сделать для этого.

Формат входных данных

`N M K L`, $1 \leqslant N, M, K, L \leqslant 2000$

Формат выходных данных

Одно число, равное количеству необходимых переливаний, или слово OOPS, если это невозможно.

Пример

стандартный ввод	стандартный вывод
10 6 5 8	7

Задача 37. Скутер

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	1 секунда
Ограничение по памяти:	256 мегабайт

У Еремея есть электросамокат — и он хочет доехать от дома до института, затратив как можно меньше энергии. Весь город расположен на холмистой местности и разделён на квадраты. Для каждого перекрёстка известна его высота в метрах над уровнем моря. Если ехать от перекрёстка с большей высотой до смежного с ним перекрёстка с меньшей высотой, то электроэнергию можно не тратить, а если наоборот, то расход энергии равен разнице высот между перекрёстками.

Помогите Еремею спланировать маршрут, чтобы он затратил наименьшее возможное количество энергии от дома до института и вывести это количество. Дом находится в левом верхнем углу, институт — в правом нижнем.

Формат входных данных

```
N M
H11 H12 ... H1M
H21 H22 ... H2M
... ... ... ...
HN1 HN2 ... HNM
```

$1 \leqslant N, M \leqslant 1000$
$0 \leqslant H_{ij} \leqslant 1000, H \in \mathbb{Z}$

Формат выходных данных

`MinConsumingEnergy`

Пример

стандартный ввод	стандартный вывод
3 5 3 1 4 1 5 9 2 6 5 3 5 9 7 9 3	5

Задача 38. Коврики

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Имеется прямоугольная область, состоящая из $6 \leqslant N \leqslant 1000$ на $6 \leqslant M \leqslant 1000$ одинаковых квадратных клеток. Часть клеток свободна, часть закрашена. Ковриком называется максимальное множество закрашенных клеток, имеющих общую границу. Коврики размером 1×1 тоже допустимы.

Требуется по заданной раскраске области определить количество находящихся на ней ковриков.

Формат входных данных

В первой строке содержатся N и M — размеры области. В каждой из следующих N строк находится ровно M символов, которые могут быть точкой, если поле свободно, или плюсом, если поле закрашено.

Формат выходных данных

Общее количество ковриков на поле.

Примеры

стандартный ввод	стандартный вывод
6 6 .++++. .....+ +..+.. ..++.. .+.... .+...+	6
10 10 .++..++.++ +.+....+++ ....++ +....++... +++.+++.++ .....+.+.. +.+.++++++ +.++.++..+ ++++...++. +...+..+..	10
5 5 ..+.. .+++. .+..+ +.+.+ +++++	2

Задача 39. Передающие станции

Имя входного файла:	стандартный ввод
Имя выходного файла:	стандартный вывод
Ограничение по времени:	1 секунда
Ограничение по памяти:	256 мегабайт

Компания сотовой связи получила лицензии на установку вышек сотовой связи в одном из новых районов города. На каждую вышку нужно установить приёмо-передатчик, который позволит связаться с другими вышками. Для снижения общей стоимости проекта приёмо-передатчики решено закупить оптом с большой скидкой, поэтому на всех вышках они будут

однотипные. Выпускается много типов передатчиков, каждый из которых имеет различную мощность, причём чем больше мощность, тем передатчик дороже и тем больше охватываемой территории. Наша задача — подобрать модель передатчика с наименьшей возможной мощностью, исходя из расположения вышек, для того, чтобы можно было передавать сообщения между любыми вышками, возможно, ретранслируя их.

Формат входных данных

Первая строка содержит количество вышек $2 \leqslant N \leqslant 2000$.

В остальных N строках — пары координат вышек в Декартовой системе $-10000 \leqslant X_i, Y_i \leqslant 10000$.

Формат выходных данных

Одно число — наименьший требуемый радиус, достаточный для функционирования всей системы с точностью до 4-х знаков после запятой.

Пример

стандартный ввод	стандартный вывод
5 0 0 1 0 0 1 1 1 3 3	2.8284

Задача 40. Кратчайшие пути

Имя входного файла:	`стандартный ввод`
Имя выходного файла:	`стандартный вывод`
Ограничение по времени:	2 секунды
Ограничение по памяти:	64 мегабайта

Взвешенный граф с `N` вершинами задан своими `M` рёбрами E_i, возможно, отрицательного веса. Требуется найти все кратчайшие пути от вершины `S` до остальных вершин. Если граф содержит отрицательные циклы, вывести слово `IMPOSSIBLE`. Если от вершины `S` до какой-либо из вершин нет маршрута, то в качестве длины маршрута вывести слово `UNREACHABLE`.

Вершины графа нумеруются, начиная с нуля.

$$3 \leqslant N \leqslant 800$$

$$1 \leqslant M \leqslant 30000$$

$$-1000 \leqslant W_i \leqslant 1000$$

В исходном файле, начиная со второй строки, описываются рёбра графа. Первое число — начальная вершина, второе число — конечная вершина, третье число — вес ребра. Кратных рёбер в графе нет, вершины нумеруются, начиная с нуля.

Формат входных данных

```
N M S
S1 E1 W1
S2 E2 W2
...
```

Формат выходных данных

```
IMPOSSIBLE

или
D1 D2 D3 ... DN
где Di может быть UNREACHABLE
```

Примеры

стандартный ввод	стандартный вывод
4 5 0 0 1 100 1 2 100 2 0 -100 0 2 1000 3 1 15	0 100 200 UNREACHABLE
3 3 0 0 1 5 1 2 8 2 0 -20	IMPOSSIBLE

Литература

[1] Кнут Д. *Искусство программирования, том 1. Основные алгоритмы., 3-е изд.* Издательский Дом «Вильямс», М., 2018.

[2] Кнут Д. *Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.* Издательский Дом «Вильямс», М., 2018.

[3] Кнут Д. *Искусство программирования, том 3. Сортировка и поиск, 3-е изд.* Издательский Дом «Вильямс», М., 2018.

[4] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. *Алгоритмы. Построение и анализ, 2-е издание. : Пер. с англ.* Издательский дом «Вильямс», Москва, Санкт-Петербург, Киев, 2013.

[5] Седжвик Р. *Алгоритмы на C++, Пер. с англ.* Издательский Дом «Вильямс», М., 2011.