Конспект курса алгоритмов

СПбГУ, второй семестр, 2020 год

Алексей Гордеев

Оглавление

I Жадные алгоритмы

1 Поиск минимального остовного дерева . . . 7
1.1 Лемма о разрезе 7
1.2 Алгоритм Прима 7
1.3 Алгоритм Краскала 8
1.4 Система непересекающихся множеств 8

II Динамическое программирование

2 Базовые понятия . . . 15
2.1 Поиск кратчайшего пути в ациклическом ориентированном графе 15
2.2 Метод динамического программирования 15
2.3 Способ вычисления переходов 16
2.4 Восстановление ответа 17
2.5 Ленивое динамическое программирование 17

3 Примеры задач . . . 18
3.1 Наибольшая общая подпоследовательность 18
3.2 Расстояние Левенштейна 19
3.3 Алгоритм Хиршберга 20
3.4 Наибольшая возрастающая подпоследовательность 21
3.5 Задача о рюкзаке 23
3.6 Произведение матриц 24
3.7 Независимое множество максимального веса в дереве 25
3.8 Динамическое программирование по подмножествам 26
3.9 Генерация номера по объекту и объекта по номеру 29
3.10 Решение рекуррентных соотношений возведением матрицы в степень 32

III Введение в теорию сложности вычислений

4 Классы сложности P и NP . . . 35
4.1 Задачи поиска 35

4.2 Decision-задачи 35
4.3 Класс P 35
4.4 Класс NP 36

5 NP-полные задачи . . . 37
5.1 Сведение по Карпу 37
5.2 NP-трудные и NP-полные задачи 37
5.3 SAT $\to$ 3-SAT 38
5.4 3-SAT $\to$ Задача о независимом множестве 39
5.5 Задача о независимом множестве $\to$ Задача о вершинном покрытии 39
5.6 Задача о вершинном покрытии $\to$ Задача о покрытии множества 40
5.7 Задача о независимом множестве $\to$ Задача о клике 40
5.8 3-SAT $\to$ Задача о трёхдольном сочетании 40
5.9 Задача о трёхдольном сочетании $\to$ Уравнение в нулях и единицах 42
5.10 Уравнение в нулях и единицах $\to$ Целочисленное линейное программирование 42
5.11 Уравнение в нулях и единицах $\to$ Задача о сумме подмножества 43
5.12 Задача о сумме подмножества $\to$ Задача о рюкзаке 43
5.13 Уравнение в нулях и единицах $\to$ Задача о гамильтоновом цикле 43
5.14 Задача о гамильтоновом цикле $\to$ Задача о гамильтоновом пути 45
5.15 Задача о гамильтоновом цикле $\to$ Задача коммивояжёра 45
5.16 Любая задача из NP $\to$ SAT 45

IV Приближённые алгоритмы и алгоритмы кэширования

6 Приближённые алгоритмы . . . 49
6.1 Задача о покрытии множества 49
6.2 Задача о вершинном покрытии 50
6.3 Кластеризация 50
6.4 Задача коммивояжёра 51
6.5 Задача о рюкзаке 53
6.6 MAX-E3-SAT 54

7 Алгоритмы кэширования . . . 55
7.1 LRU и другие алгоритмы маркировки 55
7.2 Рандомизированный алгоритм маркировки 56

V Продвинутые структуры данных

8 Дерево отрезков . . . 58
8.1 Построение 58
8.2 Реализация операций “снизу” 59

8.3 Реализация операций “сверху” 60
8.4 Область применения дерева отрезков 61
8.5 Изменение элементов на подотрезке 61
8.6 Динамическое дерево отрезков 63
8.7 Персистентное дерево отрезков 64
8.8 Две модельных задачи 66
8.9 Дерево отрезков сортированных массивов 67
8.10 Fractional cascading 67
8.11 Многомерное дерево отрезков 68
8.12 Метод сканирующей прямой 70
8.13 k-я порядковая статистика на отрезке 71

9 AVL-дерево 74
9.1 Двоичное дерево поиска 74
9.2 Базовые операции 75
9.3 AVL-дерево 76
9.4 Балансировка 76
9.5 Реализация 78

10 Декартово дерево 80
10.1 Декартово дерево 80
10.2 Операции Split и Merge 80
10.3 Остальные операции 82
10.4 Построение 83
10.5 Случайное дерево поиска 83
10.6 Дополнительные операции 85

11 Краткое отступление про B-деревья 88
11.1 B-дерево 88
11.2 2-3-дерево 88
11.3 2-3-4-дерево и RB-дерево 88

12 Splay-Дерево 90
12.1 Операция Splay 90
12.2 Остальные операции 92
12.3 Реализация 92
12.4 Оценка времени работы 93

13 Skip-list 96
13.1 Основная идея 96
13.2 Реализация 97
13.3 Оценка времени работы 99

14 RMQ и LCA 100
14.1 Разреженная таблица 100
14.2 Блочная оптимизация 101
14.3 Конструкция Фишера-Хойна 102
14.4 LCA 103
14.5 Алгоритм Тарьяна 104

I Жадные алгоритмы

1 Поиск минимального остовного дерева 7
1.1 Лемма о разрезе
1.2 Алгоритм Прима
1.3 Алгоритм Краскала
1.4 Система непересекающихся множеств

1. Поиск минимального остовного дерева

Остовное дерево (spanning tree), или *остов*, связного неориентированного графа — это остовный подграф этого графа, являющийся деревом, или, другими словами, связный остовный подграф без циклов. *Вес* $w(T)$ остова T взвешенного графа — это сумма весов его рёбер.

Пусть дан связный неориентированный граф с неотрицательными весами рёбер, и мы хотим выбрать подмножество рёбер как можно меньшего суммарного веса, с помощью которого можно было бы добраться из любой вершины в любую. Если в подмножестве рёбер есть цикл, из него всегда можно выкинуть одно из рёбер, не увеличив суммарный вес и не нарушив связности. Значит, мы хотим найти остовное дерево минимально возможного веса, или *минимальное остовное дерево (minimum spanning tree)*.

1.1 Лемма о разрезе

Поставленную задачу мы научимся решать жадно сразу двумя способами. В основе обоих алгоритмов лежит одна и та же лемма.

Разрез (cut) (A, B) графа $G = (V, E)$ — это произвольное разбиение множества вершин графа на два непересекающихся множества: $A \cup B = V$, $A \cap B = \emptyset$. Будем говорить, что ребро e *проходит через разрез* (A, B), если оно соединяет вершину из A с вершиной из B.

Лемма 1.1.1 — Лемма о разрезе.
Пусть множество рёбер $F \subset E$ входит в некоторый минимальный остов графа $G = (V, E)$. Пусть (A, B) — разрез G, причём ни одно ребро из F не проходит через разрез (A, B). Пусть e — ребро с наименьшим весом из всех, проходящих через разрез (A, B). Тогда $F \cup \{e\}$ входит в некоторый минимальный остов графа G.

Доказательство. Пусть $T = (V, E')$ — минимальный остов, содержащий F. Если e тоже лежит в T, то доказывать нечего. Иначе рассмотрим граф $T' = (V, E' \cup \{e\})$. В T' найдётся цикл, проходящий через ребро e, на этом цикле найдётся ещё хотя бы одно проходящее через разрез (A, B) ребро e'. При этом $w_e \leqslant w_{e'}$. Тогда $T'' = (V, E' \cup \{e\} \setminus \{e'\})$ — остов, при этом $w(T'') \leqslant w(T)$, T'' содержит $F \cup \{e\}$. ■

1.2 Алгоритм Прима

Алгоритм Прима (Jarnik, 30; Prim, 57; Dijkstra, 59) постепенно расширяет множество вершин A и множество рёбер F, образующих дерево на A. Вначале A состоит из одной любой вершины, F пусто. На каждом шаге алгоритм находит ребро $e = (u, v)$ минимального веса, ведущее из A в $V \setminus A$. Конец этого ребра $v \in V \setminus A$ алгоритм добавляет в A, а само ребро e — к множеству F. При этом $F \cup \{e\}$ образует дерево на $A \cup \{v\}$.

По лемме о разрезе для множества рёбер F и разреза $(A, V \setminus A)$ если F входило в какой-то минимальный остов, то и $F \cup \{e\}$ тоже входит в какой-то минимальный остов. Когда $A = V$, рёбра F образуют остовное дерево всего графа G, при этом F входит в какое-то минимальное остовное дерево. Значит, минимальное остовное дерево состоит ровно из рёбер множества F.

Для того, чтобы быстро находить вершину v и вес ребра w_e, для каждой вершины $u \in V$ алгоритм поддерживает вес минимального ребра, ведущего из A в u:

$$d_u = \min\{w_f \ : \ f : a \to u, a \in A\}.$$

Тогда на очередном шаге v — это вершина из $V \setminus A$ с минимальным значением d_v, а само d_v — вес ребра e. Для перехода к следующему шагу нужно обновить значения d_u весами рёбер, исходящих из v.

Получается алгоритм, практически идентичный алгоритму Дейкстры (меняются лишь приоритеты вершин). Соответственно, в зависимости от реализации, можно получить оценки времени работы $O(V^2 + E)$, $O((V + E)\log V)$ ($O(V \log V + E)$ при использовании фибоначчиевой кучи).

```
1  vector<int> d(n) # в графе n вершин
2  fill(d, inf)
3  d[0] = 0
4  for v = 0..(n - 1):
5    q <-- (v, d[v]) # кладём в очередь вершину v с приоритетом d[v]
6  int W = 0 # сюда запишем вес минимального остова
7  for i = 0..(n - 1):
8    v <-- q # достаём из очереди вершину с минимальным приоритетом
9    W += d[v]
10   for u, w in es[v]:
11     if d[u] > w:
12       d[u] = w
13       q.decreaseKey(u, d[u]) # уменьшаем значение приоритета u до d[u]
```

Если нужно восстановить не только вес минимального остова, но и множество входящих в него рёбер, нужно для каждой вершины u дополнительно запоминать, весу какого ребра равняется d_u.

1.3 Алгоритм Краскала

Алгоритм Краскала (Kruskal, 56) действует тоже жадно, но по-другому: он начинает с пустого множества F, рассматривает рёбра в порядке возрастания веса и добавляет очередное ребро e в F, если при этом в F не появится цикла.

В любой момент времени рёбра F образуют лес, и, поскольку граф связен, в конце алгоритма F будет образовывать остовное дерево. Из леммы о разрезе снова следует, что в любой момент времени F входит в какой-то минимальный остов (на этот раз нужно в качестве разреза взять $(A, V \setminus A)$, где A — компонента связности одного из концов ребра e в графе, образованном рёбрами F).

Для реализации алгоритма Краскала нам не хватает ещё одной детали: нужно научиться быстро понимать, образуется ли цикл при добавлении ребра e к множеству F. Цикл образуется тогда и только тогда, когда концы ребра e находятся в одной компоненте связности графа, образованного рёбрами F. Таким образом, наша ближайшая цель — научиться быстро проверять, находятся ли две вершины в одной компоненте связности (а также быстро объединять две компоненты связности в одну).

1.4 Система непересекающихся множеств

Система непересекающихся множеств (disjoint set union, DSU) — структура данных, позволяющая поддерживать разбиение n элементов на непересекающиеся множества, а именно:

- выдавать по элементу идентификатор множества, в котором лежит этот элемент;
- объединять два множества в одно.

Будем считать, что элементы пронумерованы числами от 0 до $n-1$. Необходимо реализовать две функции: `get(a)` — функцию, возвращающую идентификатор множества, где лежит a (в частности, с помощью этой функции можно проверять, лежат ли два элемента в одном множестве); а также `join(a, b)` — функцию, объединяющую множества, в которых лежат элементы a и b. Пусть в случае, когда a и b лежат в одном множестве, функция `join(a,b)` ничего не делает.

Также будем считать, что после инициализации структуры каждое множество состоит из одного элемента и `get(i) = i` (если в решаемой задаче это не так, можно после инициализации объединить элементы в нужные множества с помощью функции `join`).

Реализация связными списками

Будем поддерживать $id[i]$ — номер множества, в котором лежит элемент i, а для множества с номером i будем поддерживать список его элементов $elems[i]$. Единственная хитрость: при объединении множеств сохраним для объединённого множеств идентификатор большего множества из объединяемых, тогда нужно поменять значения id лишь у элементов меньшего множества. Объединение списков же осуществляется за $O(1)$ (достаточно соединить ссылками конец одного списка и начало другого).

```
init(n):
  for i = 0..(n - 1):
    id[i] = i
    elems[i] = {i}

get(i):
  return id[i]

join(a, b):
  a = id[a], b = id[b]
  if a == b:
    return
  if len(elems[a]) < len(elems[b]):
    swap(a, b)
  for x in elems[b]:
    id[x] = a
  elems[a] = concatenate(elems[a], elems[b])
```

При такой реализации функция `get` работает за $O(1)$.

Предложение 1.4.1 Суммарное время работы m вызовов `join` есть $O(m + n\log n)$.

Доказательство. Время работы одного вызова `join` пропорционально $O(1+x)$, где x — число элементов, у которых поменялось значение id. Заметим, что когда у элемента меняется значение id, размер множества, в котором он находится, увеличивается хотя бы два раза. Значит значение id каждого элемента поменяется не более $\log_2 n$ раз. ■

Заметим, что такая реализация DSU уже позволяет реализовать алгоритм Краскала за время $O(E\log E + E + E\log V) = O(E\log V)$: первое слагаемое — сортировка рёбер по весу — мажорирует время работы последующих шагов.

R $O(E\log E) = O(E\log V)$, так как $E \leqslant V^2$, если в графе нет кратных рёбер (а если они есть, от них можно предварительно избавиться, оставив лишь самое лёгкое ребро между каждой парой вершин).

Тем не менее, мы изучим ещё одну реализацию DSU, которая даёт лучшую амортизированную оценку времени работы операций, чем реализация связными списками, и ничуть не сложнее с точки зрения технической реализации.

Реализация деревьями

Каждое множество будем хранить в виде подвешенного дерева, вершины которого — это элементы множества. Для каждого элемента v будем хранить ссылку $p[v]$ на родителя в его дереве-множестве; условимся, что для корня дерева ссылка на родителя ведёт на сам корень: $p[v] = v$. В качестве идентификатора множества удобно использовать корень дерева.

Теперь функция `get(v)` — это просто подъём в корень дерева, где лежит v, а функцию `join(a, b)` можно реализовать, например, подвесив корень одного дерева ребёнком к корню другого.

```
init(n):
  for i = 0..(n - 1):
    p[i] = i

get(v):
  if p[v] == v:
    return v
  return get(p[v])

join(a, b):
  a = get(a), b = get(b)
  if a == b:
    return
  p[a] = b
```

Время работы `join` — это $O(1)$ плюс время работы двух запусков `get`. К сожалению, время работы `get` пока в худшем случае оценивается как $\Theta(n)$, так как высота дерева может равняться его размеру. Для улучшения оценки на время работы `get` применяются следующие две эвристики.

Ранговая эвристика

При объединении деревьев логично подвешивать более низкое дерево к более высокому, тогда высота будет расти медленнее. Для каждой вершины v будем дополнительно хранить ещё одно значение: ранг $rk[v]$, равный высоте поддерева вершины v. Вначале $rk[v] = 0$ для всех вершин, а при объединении деревьев будем подвешивать корень с меньшим рангом ребёнком к корню с большим рангом. При равенстве рангов неважно, какой корень к какому подвешивать, но надо увеличить ранг корня получившегося дерева на единицу.

```
init(n):
  for i = 0..(n - 1):
    p[i] = i, rk[i] = 0

join(a, b):
  a = get(a), b = get(b)
  if a == b:
    return
  if rk[a] > rk[b]:
    swap(a, b)
  if rk[a] == rk[b]:
    rk[b] += 1
  p[a] = b
```

Предложение 1.4.2 Для рангов вершин выполняются следующие свойства:

1. если $v \neq p[v]$, то $rk[v] < rk[p[v]]$;
2. вершина ранга k имеет поддерево размера хотя бы 2^k;
3. количество вершин ранга k не превосходит $n/2^k$.

Доказательство. Первое свойство следует из того, что ранг — это высота поддерева вершины.

Заметим, что вершина ранга k может появиться только при объединении двух деревьев с корнями ранга $k-1$. Отсюда по индукции следует второе свойство.

Третье свойство следует из того, что разные вершины ранга k не могут иметь общих потомков. ■

Из третьего свойства, в частности, следует, что ранг любой вершины не превосходит $\log_2 n$. Значит, высота любого дерева не превосходит $\log_2 n$, и время работы функции get теперь оценивается как $O(\log n)$.

Эвристика сжатия путей

Другая идея: когда функция get(v) проходит путь к корню дерева v и находит его, можно переподвесить v напрямую к корню, чтобы при следующем запуске не проходить этот путь заново; структура дерева изменится, но само множество, соответствующее дереву, при этом останется прежним, поэтому такая операция корректна.

```
1 get(v):
2   if p[v] == v:
3     return v
4   p[v] = get(p[v])
5   return p[v]
```

Если использовать эвристику сжатия путей без ранговой эвристики, то тоже можно получить оценку времени работы get $O(\log n)$, но уже амортизированную.

Предложение 1.4.3 При использовании эвристики сжатия путей без ранговой эвристики m запросов get имеют суммарное время работы $O(m + (m+n)\log n)$.

Доказательство. Пусть $sz(v)$ — размер поддерева v. Назовём ребро из v в $p[v]$ ***лёгким***, если $sz(v) \leqslant sz(p[v])/2$, и ***тяжёлым*** иначе.

Заметим, что при выполнении get проход по каждому лёгкому ребру хотя бы удваивает размер текущего поддерева. Значит, каждый запрос get пройдёт не более чем по $\log_2 n$ лёгким рёбрам, а суммарно проходов по лёгким рёбрам будет не более $m \log_2 n$.

Теперь заметим, что если ребро из v в $w = p[v]$ тяжёлое, и при этом w не корень, то после прохода по такому ребру в запросе get размер поддерева w уменьшится хотя бы в два раза. При этом у вершины, не являющейся корнем, размер поддерева может только уменьшаться. Значит, за всё время работы мы суммарно пройдём по тяжёлым рёбрам, не ведущим в корень, не более $n \log_2 n$ раз — не более чем $\log_2 n$ раз для каждой вершины w.

Осталось учесть тяжёлые рёбра, ведущие в корень; на каждый запрос get приходится не более одного такого ребра, суммарно их не более m. ■

Совместное использование эвристик

Что будет, если использовать обе эвристики одновременно? Заметим, что в этом случае ранг уже не всегда равен высоте поддерева вершины. Тем не менее, большая часть свойств остаётся верной.

Предложение 1.4.4 Для рангов вершин выполняются следующие свойства:

1. если $v \neq p[v]$, то $rk[v] < rk[p[v]]$;
2. **корневая** вершина ранга k имеет поддерево размера хотя бы 2^k;
3. количество вершин ранга k не превосходит $n/2^k$.

Доказательство. Первое свойство выполняется сразу после `join`, в котором v перестала быть корнем, а при переподвешивании v в `get` разность между рангами только увеличивается.

Как и раньше, вершина ранга k может появиться только при объединении двух деревьев с корнями ранга $k-1$. Отсюда по индукции следует второе свойство (но только для корневых вершин: важно, что пока вершина остаётся корнем своего дерева, размер её поддерева не уменьшается).

Третье свойство теперь тоже нужно доказывать аккуратнее: любая вершина ранга k была корневой, когда получила этот ранг. В этот момент в её поддереве было хотя бы 2^k вершин, при этом ни до, ни после этого эти вершины не могли оказаться в поддереве другой вершины ранга k: до этого они находились в деревьях ранга $k-1$, а после этого любой новый корень дерева, где они находятся, будет иметь ранг строго больше k. Значит, каждой вершине ранга k соответствует множество из 2^k вершин, и эти множества не пересекаются для разных вершин ранга k. ■

Определение 1.4.1 *Итерированный логарифм* n по основанию a

$$\log_a^* n = \begin{cases} 0, & \text{если } n \leqslant 1, \\ 1 + \log_a^*(\log_a n), & \text{если } n > 1. \end{cases}$$

R На практике можно считать, что если n — число элементов, то $\log_2^* n \leqslant 5$, так как даже $\log_2^*(2^{65536}) = 5$.

Теорема 1.4.5 При использовании обеих эвристик суммарное время работы m запросов `get` есть $O(m + n\log_2^* n)$.

Доказательство. Ранг любой вершины в любой момент лежит в отрезке $[0, \log_2 n]$. Покроем этот отрезок непересекающимися отрезками вида $[k+1, 2^k]$ (а также отрезком $[0,0]$):

$$\{0\},\ \{1\},\ \{2\},\ \{3,4\},\ \{5,6,\ldots,2^4=16\},\ \{17,18,\ldots,2^{16}=65536\},\ldots$$

Всего таких отрезков будет не более $\log_2^* n + 1$. Пронумеруем эти отрезки.

Назовём *крутостью* c_v ребра из v в $p[v]$ разность номеров отрезков, в которые попадают $rk[p[v]]$ и $rk[v]$. Так, крутость ребра из вершины ранга 5 в вершину ранга 13 равна нулю; крутость ребра из вершины ранга 4 в вершину ранга 65538 равна трём. Крутость любого ребра не превосходит $\log_2^* n$. Поскольку разность рангов родителя вершины и самой вершины в ходе алгоритма может только возрастать, крутость ребра из вершины в её родителя тоже может только возрастать.

Время работы i-го запроса `get` пропорционально числу пройденных по пути к корню рёбер. Поделим эти рёбра на несколько типов: отдельно посчитаем ребро, ведущее в корень дерева, и ближайшее к корню ребро положительной крутости (возможно, эти два ребра совпадают). Пусть помимо этих рёбер на пути было ещё a_i рёбер положительной крутости (назовём их a-рёбрами) и b_i рёбер нулевой крутости (назовём их b-рёбрами). Тогда при выполнении i-го запроса `get` было посещено не более $2 + a_i + b_i$ рёбер.

Пусть ребро из v в $p[v]$ — одно из a-рёбер. Поскольку на пути в корень было ещё хотя бы одно ребро положительной крутости, крутость c_v после переподвешивания вершины v строго возрастёт. Значит, из каждой вершины v мы могли подняться по a-ребру за всё время работы не более $\log_2^* n$ раз, то есть $\sum_i a_i \leqslant n\log_2^* n$.

Теперь пусть ребро из v в $p[v]$ — одно из b-рёбер. После переподвешивания v разность рангов $rk[p[v]] - rk[v]$ возросла хотя бы на единицу. Пусть $rk[v]$ находится в отрезке $[k+1, 2^k]$, тогда подъёмов из вершины v по b-ребру было не более 2^k за всё время работы (после 2^k подъёмов $rk[p[v]]$ уже точно будет находиться в следующем отрезке, то есть ребро в родителя будет иметь положительную крутость). Заметим, что вершин с рангом в отрезке $[k+1, 2^k]$ всего не более

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \dots \leqslant \frac{n}{2^k},$$

значит, проходов по b-рёбрам из вершин с рангом в отрезке $[k+1, 2^k]$ за всё время было не более $2^k \cdot n/2^k = n$. Тогда всего проходов по b-рёбрам было не более $n \log_2^* n$.

Итак, суммарное время работы m запросов get есть

$$O\left(\sum_{i=1}^{m} 2 + a_i + b_i\right) = O(2m + n\log_2^* n + n\log_2^* n) = O(m + n\log_2^* n).$$

■

R Можно показать, что суммарное время работы m запросов get не превосходит $O(m \cdot \alpha(n))$, где $\alpha(n)$ — обратная функция Аккермана, растущая ещё медленнее, чем итерированный логарифм.

Возвращаясь к алгоритму Краскала

```
sort(es) # сортируем рёбра по возрастанию веса
init(n) # инициализируем DSU
for e in es: # ребро из e.a в e.b веса e.w
  if get(e.a) != get(e.b):
    join(e.a, e.b)
    mst.push_back(e)# добавляем ребро к ответу
```

При использовании обеих эвристик получаем время работы $O(E \log V + E + V \log_2^* V)$ (или $O(E \log V + E \cdot \alpha(V))$). В случае, когда рёбра уже даны в порядке возрастания веса, либо когда из-за особенностей задачи рёбра можно отсортировать быстрее, чем за $O(E \log V)$, реализация DSU деревьями позволяет улучшить эффективность алгоритма.

R Помимо изученных нами алгоритмов, известен рандомизированный алгоритм поиска минимального остова с линейным математическим ожиданием временем работы (Karger, Klein, Tarjan, 1995), а также алгоритм с временем работы $O(E \cdot \alpha(E, V))$ (Chazelle, 2000), где α — двупараметрическая версия обратной функции Аккермана (которая тоже растёт очень медленно).

II Динамическое программирование

2 Базовые понятия 15
2.1 Поиск кратчайшего пути в ациклическом ориентированном графе
2.2 Метод динамического программирования
2.3 Способ вычисления переходов
2.4 Восстановление ответа
2.5 Ленивое динамическое программирование

3 Примеры задач 18
3.1 Наибольшая общая подпоследовательность
3.2 Расстояние Левенштейна
3.3 Алгоритм Хиршберга
3.4 Наибольшая возрастающая подпоследовательность
3.5 Задача о рюкзаке
3.6 Произведение матриц
3.7 Независимое множество максимального веса в дереве
3.8 Динамическое программирование по подмножествам
3.9 Генерация номера по объекту и объекта по номеру
3.10 Решение рекуррентных соотношений возведением матрицы в степень

2. Базовые понятия

2.1 Поиск кратчайшего пути в ациклическом ориентированном графе

Пусть перед нами стоит задача поиска кратчайших путей от вершины s до всех остальных вершин в ациклическом ориентированном графе. Конечно, можно воспользоваться одним из уже изученных алгоритмов. Есть, однако, и более простое решение с линейным временем работы.

Для любой вершины $v \neq s$ верно следующее равенство:

$$dist[v] = \min\{dist[u] + w_e : \ e = (u, v) \in E\}.$$

Будем перебирать вершины в порядке топологической сортировки. Тогда все значения $dist$, встречающиеся в выражении справа, к моменту рассмотрения вершины v уже будут найдены.

```
1 fill(dist, inf)
2 dist[s] = 0
3 for v in topOrder: # рассматриваем вершины в порядке топологической сортировки
4   if v == s:
5     continue
6   for u, w in inEs[v]: # inEs[v] - список входящих в v рёбер
7     dist[v] = min(dist[v], dist[u] + w)
```

Алгоритм рассматривает подзадачи вычисления $dist[v]$ для конкретной вершины v в порядке увеличения сложности; каждую текущую подзадачу он сводит к нескольким подзадачам попроще, которые уже решены.

Заметим, что ровно таким же способом можно найти и самый длинный путь до каждой вершины; или, например, путь с максимальным произведением длин рёбер.

2.2 Метод динамического программирования

Метод динамического программирования так и устроен: задача, которую нужно решить, сводится к подзадачам попроще, те — к подзадачам ещё попроще, и так далее. После этого все подзадачи решаются в правильном порядке. Какой порядок правильный? Построим на подзадачах ориентированный граф: вершины — это подзадачи, а ребро из a в b есть, если для решения подзадачи b требуется сначала решить a. Тогда правильный порядок — это топологическая сортировка этого графа (разумеется, в графе на подзадачах не должно быть циклов, иначе никакой порядок не подойдёт, и решить задачу таким методом не удастся). Этот граф, как правило, не строится явно; правильный порядок часто виден сразу и не требует запуска алгоритма топологической сортировки.

Подзадачи мы будем называть состояниями, зависимости между состояниями (рёбра построенного выше графа) — переходами. Начальные состояния — это самые простые подзадачи, решение которых очевидно или известно заранее. В задаче выше начальное состояние — $dist[s] = 0$. Может быть полезно держать в голове аналогию с методом математической индукции: начальное состояние — аналог базы индукции; в методе индукции утверждение для всех остальных состояний доказывается с помощью индукционного

перехода, а в методе динамического программирования значения остальных состояний вычисляются с помощью переходов.

На самом деле мы уже встречались с методом динамического программирования: полиномиальный алгоритм вычисления чисел Фибоначчи, алгоритмы Флойда и Форда-Беллмана, решето Эратосфена — все эти алгоритмы можно интерпретировать как динамическое программирование.

Проще всего с числами Фибоначчи: каждое число Фибоначчи соответствует одному состоянию; для того, чтобы узнать значение n-го состояния, мы пользуемся значениями $(n-1)$-го и $(n-2)$-го.

В алгоритме Флойда состояние — это тройка (k, i, j): $dist[k, i, j]$ — это длина кратчайшего пути из i в j, промежуточные вершины в котором имеют номера меньше k. Начальные состояния — тройки с $k = 0$, для вычисления $dist[k+1, i, j]$ нужны $dist[k, i, j]$, $dist[k, i, k]$ и $dist[k, k, j]$.

В алгоритме Форда-Беллмана состояние — это пара (k, v): $dist[k, v]$ — это длина кратчайшего пути из s в v, состоящего из не более чем k рёбер (мы изучали версию алгоритма, соптимизированную по памяти и использующую одномерный массив вместо двумерного).

На задачу определения простых чисел на отрезке $[1, n]$ можно смотреть так: есть переходы из числа во все числа, кратные ему; число простое, если в него нет переходов. Решето Эратосфена рассматривает только переходы из простых чисел в их кратные (потому что этого достаточно); линейная версия решета действует ещё аккуратнее, оставляя лишь по одному переходу в каждое составное число.

2.3 Способ вычисления переходов

Во многих задачах переходы можно вычислять двумя способами: “вперёд” и “назад”. Так, при поиске кратчайшего пути в ациклическом ориентированном графе вместо того, чтобы смотреть “назад” на входящие в вершину рёбра и пересчитывать текущее состояние через предыдущие, мы могли бы смотреть “вперёд” на исходящие из вершины рёбра и пересчитывать следующие состояния через текущее.

```
1 fill(dist, inf)
2 dist[s] = 0
3 for v in topOrder: # рассматриваем вершины в порядке топологической сортировки
4   for u, w in es[v]: # es[v] - список исходящих из v рёбер
5     dist[u] = min(dist[u], dist[v] + w)
```

Иногда оказывается, что один из способов оказывается по тем или иным причинам удобнее другого. Рассмотрим для примера такую модельную задачу: пусть $dp[1] = 1$, и мы хотим для каждого x от 2 до n вычислить

$$dp[x] = \sum_{\substack{d<x \\ d|x}} dp[d].$$

Для того, чтобы считать переходы назад, нам придётся найти все делители каждого числа, что требует какой-то дополнительной работы. Считать же переходы вперёд очень просто: для фиксированного d значение $dp[d]$ нужно добавить ко всем значениям состояний, кратных d и не больших n. Их легко перебрать простым циклом. При этом суммарное время работы составит

$$O\left(n + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n}\right) = O(n \log n).$$

2.4 Восстановление ответа

В задаче о кратчайшем пути может требоваться найти не просто длину пути, но и сам путь. Подобное может требоваться и в других задачах: нужно найти не просто число, а какой-то развёрнутый ответ. Как правило, восстановление развёрнутого ответа оказывается эквивалентным восстановлению пути из оптимальных переходов от начального состояния к конечному (мы увидим это на примерах задач). Есть два стандартных способа восстановления ответа. Первым из них мы пользовались в алгоритмах Флойда и Форда-Беллмана: нужно для каждого состояния запомнить, из какого состояния в него был сделан оптимальный переход. Тогда, пользуясь запомненными ссылками, можно откатиться от конечного состояния к начальному и восстановить весь путь.

Второй способ — можно ничего не запоминать, а для того, чтобы найти, откуда был сделан оптимальный переход, просто перебрать все переходы заново (то есть повторить заново всё, что делали при вычислении значения состояния, но запомнить, при рассмотрении какого перехода был найден ответ).

2.5 Ленивое динамическое программирование

Вычисление переходов назад можно делать лениво: напишем рекурсивную функцию, делающую рекурсивные вызовы от состояний, которые нужны для вычисления значения текущего состояния. При этом будем запоминать уже вычисленные значения, чтобы не проводить одни и те же вычисления несколько раз. Пример того, как можно лениво решать задачу поиска кратчайшего пути в ациклическом ориентированном графе:

```
1  fill(dist, -1) # -1 значит, что состояние ещё не рассматривалось
2  
3  findDist(v):
4    if dist[v] != -1:
5      return dist[v]
6    dist[v] = inf
7    for u, w in inEs[v]: # inEs[v] - список входящих в v рёбер
8      dist[v] = min(dist[v], findDist(u) + w)
9  
10 dist[s] = 0
11 for v = 0..(n - 1): # порядок рассмотрения вершин не важен!
12   findDist(v)
```

Как правило, ленивая версия решения работает медленнее неленивой из-за рекурсивных вызовов. Зато иногда она бывает полезна, когда нужно вычислить значение не всех состояний, а какого-либо одного. Ленивая версия не будет вычислять значения состояний, которые не используются при вычислении требуемого. Если таких состояний много, выигрыш в скорости может быть велик. Модельный пример: значения $\gcd$ от всех пар чисел от 0 до n можно вычислить методом динамического программирования, пользуясь соотношением

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

При этом, как мы знаем, вычисление одного конкретного состояния требует вычисления лишь ещё $O(\log n)$ других состояний. Поэтому если нас интересуют лишь несколько состояний, а не все, намного эффективнее вычислить их лениво.

3. Примеры задач

3.1 Наибольшая общая подпоследовательность

Пусть даны две последовательности чисел (символов) $a_1, \ldots, a_n$ и $b_1, \ldots, b_m$. Требуется найти последовательность c как можно большей длины, которая была бы подпоследовательностью и a, и b. Например, для последовательностей $2, 1, 2, 3, 1, 2$ и $2, 3, 2, 2, 1$ возможные ответы — $2, 2, 2$; $2, 3, 2$; $2, 3, 1$ и $2, 2, 1$.

Пусть $l[i, j]$ — длина наибольшей общей последовательности (НОП, longest common subsequence, LCS) последовательностей $a_1, \ldots, a_i$ и $b_1, \ldots, b_j$. Начальные значения: $l[i, 0] = l[0, j] = 0$. Нас интересует $l[n, m]$.

Как вычислить $l[i, j]$? Посмотрим на a_i и b_j: либо один из них не входит в ответ (то есть $l[i, j] = l[i - 1, j]$ или $l[i, j] = l[i, j - 1]$), либо они оба совпадают с последним элементом ответа, тогда $l[i, j] = l[i - 1, j - 1] + 1$. Последний случай возможен лишь при $a_i = b_j$. Поскольку нас интересует наибольшая общая подпоследовательность, нужно взять максимум по всем возможным переходам. Заметим, что если мы будем перебирать состояния в порядке увеличения i, а потом j, то к моменту рассмотрения состояния (i, j) состояния $(i - 1, j)$, $(i, j - 1)$ и $(i - 1, j - 1)$ уже будут рассмотрены. Получаем решение за $O(nm)$.

```
1 fill(l, 0)
2 for i = 1..n:
3   for j = 1..m:
4     l[i, j] = max(l[i - 1, j], l[i, j - 1])
5     if a[i] == b[j]:
6       l[i, j] = max(l[i, j], l[i - 1, j - 1] + 1)
```

			2	3	2	2	1
		0	1	2	3	4	5
	0	0	0	0	0	0	0
2	1	0	1	1	1	1	1
1	2	0	1	1	1	1	2
2	3	0	1	1	2	2	2
3	4	0	1	2	2	2	2
1	5	0	1	2	2	2	3
2	6	0	1	2	3	3	3

Рис. 3.1: Значения l для последовательностей $2, 1, 2, 3, 1, 2$ и $2, 3, 2, 2, 1$

Восстановление ответа

Пусть требуется узнать не только длину НОП, но и найти саму подпоследовательность. Как мы уже обсуждали, это делается любым из двух способов: можно запомнить для каждого состояния, какой из переходов в него был оптимальным, а можно ничего не запоминать, а находить оптимальный переход по мере надобности, перебирая все переходы заново. Так

или иначе, будем откатываться из конечного состояния (n, m), переходя по оптимальным переходам и добавляя символ в ответ, когда переход идёт "по диагонали". Остановимся, когда придём в одно из начальных состояний. Приведём для разнообразия реализацию второго способа.

```
i = n, j = m
while i > 0 and j > 0:
  if a[i] == b[j] and l[i, j] == l[i - 1, j - 1] + 1:
    ans.push_back(a[i])
    i -=1, j -= 1
  else if l[i, j] == l[i - 1, j]:
    i -= 1
  else: # l[i, j] == l[i, j - 1]
    j -= 1
reverse(ans.begin(), ans.end())
```

Оптимизация памяти

Пусть нас интересует только длина НОП, тогда можно обойтись $O(\min(m, n))$ дополнительной памяти. Для этого заметим, что при вычислении i-й строки таблицы состояний мы используем только i-ю и $(i-1)$-ю строки. Значит, все предыдущие строки можно уже не хранить — получаем решение с $O(m)$ памяти. Остаётся заметить, что последовательности всегда можно поменять местами, поэтому хватит и $O(\min(m, n))$ памяти.

```
int l[2][m + 1]
fill(l, 0)
for i = 1..n:
  cur = i % 2, prev = 1 - cur
  for j = 1..m:
    l[cur, j] = max(l[prev, j], l[cur, j - 1])
    if a[i] == b[j]:
      l[cur, j] = max(l[cur, j], l[prev, j - 1] + 1)
```

3.2 Расстояние Левенштейна

Расстояние Левенштейна (редакционное расстояние) между двумя последовательностями — это минимальное количество операций вставки, удаления и замены символа, с помощью которых можно из одной последовательности получить другую.

Расстояние Левенштейна между $a_1, \dots, a_n$ и $b_1, \dots, b_m$ можно вычислить алгоритмом, практически полностью совпадающим с алгоритмом поиска НОП: пусть $d[i, j]$ — расстояние Левенштейна между последовательностями $a_1, \dots, a_i$ и $b_1, \dots, b_j$. Тогда $d[i, 0] = i$, $d[0, j] = j$, а при $i, j > 0$ верно

$$d[i, j] = \min \begin{cases} l[i-1, j] + 1, \\ l[i, j-1] + 1, \\ l[i-1, j-1] + \chi(a_i, b_j), \end{cases}$$

где $\chi(x, y) = 0$ при $x = y$, $\chi(x, y) = 1$ иначе. Первый случай соответствует удалению a_i, второй — вставке в последовательность a символа b_j после a_i, третий — замене a_i на b_j (или бездействию в случае $a_i = b_j$).

Восстановление ответа делается полностью аналогично восстановлению ответа в алгоритме поиска НОП. Если требуется найти лишь расстояние, но не последовательность действий, то снова достаточно $O(\min(m, n))$ дополнительной памяти.

```
for i = 0..n:
  for j = 0..m:
    if i == 0 or j == 0:
      d[i, j] = max(i, j)
      continue
    d[i, j] = min(d[i - 1, j], d[i, j - 1]) + 1
    if a[i] == b[j]:
      d[i, j] = min(d[i, j], d[i - 1, j - 1])
    else:
      d[i, j] = min(d[i, j], d[i - 1, j - 1] + 1)
```

			2	3	2	2	1
		0	1	2	3	4	5
	0	0	1	2	3	4	5
2	1	1	0	1	2	3	4
1	2	2	1	1	2	3	3
2	3	3	2	2	1	2	3
3	4	4	3	2	2	2	3
1	5	5	4	3	3	3	2
2	6	6	5	4	3	3	3

Рис. 3.2: Значения d для последовательностей $2, 1, 2, 3, 1, 2$ и $2, 3, 2, 2, 1$

2	1	2	3	1	2
2	3	2	3	1	2
2	3	2	2	1	2
2	3	2	2	1	

Рис. 3.3: Последовательность операций, соответствующая расстоянию Левенштейна

3.3 Алгоритм Хиршберга

В предыдущих двух алгоритмах мы имели альтернативу: либо мы умеем восстанавливать ответ, либо можем себе позволить использовать $O(\min(m, n))$ дополнительной памяти. Алгоритм Хиршберга позволяет восстанавливать ответ, используя $O(\min(m, n) + \log(\max(m, n)))$ дополнительной памяти. Идея алгоритма применима не только в этих двух, но и во многих других задачах.

Для определённости будем решать задачу о поиске НОП. В очередной раз применим метод "разделяй и властвуй": отдельно решим задачу для последовательностей $a_1, \ldots, a_{n/2}$ и $b_1, \ldots, b_m$, и для последовательностей $a_n, \ldots, a_{n/2+1}$ и $b_m, \ldots, b_1$. И ту, и другую задачу решим, используя $O(m)$ дополнительной памяти.

Пусть в результате $l[j]$ — длина НОП $a_1, \ldots, a_{n/2}$ и $b_1, \ldots, b_j$; $r[j]$ — длина НОП $a_n, \ldots, a_{n/2+1}$ и $b_m, \ldots, b_{m+1-j}$. Тогда длина НОП $a_1, \ldots, a_n$ и $b_1, \ldots, b_m$ — это максимум $l[j] + r[m - j]$ по $0 \leqslant j \leqslant m$.

Пусть максимальное значение достигается на j_{max}. Сделаем рекурсивные запуски алгоритма от $a_1, \ldots, a_{n/2}$ и $b_1, \ldots, b_{j_{max}}$ и от $a_{n/2+1}, \ldots, a_n$ и $b_{j_{max}+1}, \ldots, b_m$, чтобы восстановить половинки ответа.

Глубина рекурсии составит $O(\log n)$, каждый рекурсивный запуск может переиспользовать одни и те же $O(m)$ дополнительной памяти, значит, всего понадобится $O(m + \log n)$ дополнительной памяти.

Осталось понять, почему оценка времени работы по-прежнему равна $O(mn)$. Посмотрим на все рекурсивные вызовы на глубине k. Во всех них первая последовательность имеет длину $O(n/2^k)$, а сумма длин вторых последовательностей по всем вызовам на глубине k равна m. Значит, суммарное время работы рекурсивных вызовов на глубине k равно $O(mn/2^k)$. Тогда суммарное время работы всех вызовов есть

$$\sum_k O(mn/2^k) = O(mn).$$

```
hirschberg(a, b):
  n = len(a), m = len(b)
  if n <= 1 or m <= 1:
    # если одна из последовательностей имеет длину не больше 1,
    # решим задачу обычным алгоритмом
    return naiveLCS(a, b)
  l = getLCS(a[0, n / 2), b)
  r = getLCS(reverse(a[n / 2, n)), reverse(b))
  j = argmax(l[j] + r[m - j], j = 0..m)
  return hirschberg(a[0, n / 2), b[0, j))
       + hirschberg(a[n / 2, n), b[j, m))
```

Альтернативный способ

Другой способ, работающий по тем же причинам: будем действовать, как обычно при восстановлении ответа, но запоминать будем не предыдущее состояние, а состояние на оптимальном пути от начального с $i = n/2$.

```
int l[2][m + 1]
int mid[2][m + 1]

relax(i, j, newL, newMid):
  if l[i, j] < newL:
    l[i, j] = newL, mid[i, j] = newMid

hirschberg(a, b):
  n = len(a), m = len(b)
  if n <= 1 or m <= 1:
    return naiveLCS(a, b)
  fill(l, 0), fill(mid, 0)
  for i = 1..n:
    cur = i % 2, prev = 1 - cur
    for j = 1..m:
      relax(cur, j, l[prev, j], mid[prev, j])
      relax(cur, j, l[cur, j - 1], mid[cur, j - 1])
      if a[i] == b[j]:
        relax(cur, j, l[prev, j - 1] + 1, mid[prev, j - 1])
      if i == n / 2:
        mid[cur, j] = j
  j = mid[n, m]
  return hirschberg(a[0, n / 2), b[0, j))
       + hirschberg(a[n / 2, n), b[j, m))
```

3.4 Наибольшая возрастающая подпоследовательность

Дана последовательность чисел $a_1, \ldots, a_n$. Требуется найти возрастающую подпоследовательность (такую, что каждое следующее число в ней строго больше предыдущего) максимально возможной длины — наибольшую возрастающую подпоследовательность (НВП, longest increasing subsequence).

Решение за $O(n^2)$

Пусть l_i — максимально возможная длина возрастающей подпоследовательности, заканчивающейся в a_i. Тогда

$$l_i = 1 + \max_{\substack{0 \leqslant j < i, \\ a_j < a_i}} l_j$$

(максимум по пустому множеству считаем равным нулю).

Ответ на задачу — максимальное значение l_i по всем i. Восстановление ответа делается стандартными методами. Получаем решение за $O(n^2)$.

```
fill(l, 1)
for i = 1..(n - 1):
  for j = 0..(i - 1):
    if a[j] < a[i]:
      l[i] = max(l[i], l[j] + 1)
```

Решение за $O(n \log n)$

Будем поддерживать дополнительный массив: пусть

$$x[i,k] = \min_{\substack{j < i, \\ l_j = k}} a_j$$

(здесь минимум по пустому множеству — бесконечность). Тогда

$$l_i = 1 + \max_{k\colon\ x[i,k] < a_i} k.$$

Как это помогает решить задачу быстрее? Заметим, что элементы одномерного массива $x[i]$ идут в порядке возрастания: для любого $k > 1$ выполняется $x[i, k-1] < x[i,k]$ (если, конечно, оба элемента не равны бесконечности). Почему это так? Если $x[i,k] \neq \infty$, то $x[i,k] = a_j$ для некоторого j такого, что $l_j = k$. Существует $q < j$: $l_q = l_j - 1 = k - 1$, $a_q < a_j$. Тогда $x[i, k-1] \leqslant a_q < a_j = x[i,k]$.

Поскольку элементы массива $x[i]$ идут в порядке возрастания, l_i теперь можно найти двоичным поиском по массиву $x[i]$ за $O(\log n)$. Последнее замечание: массивы $x[i+1]$ и $x[i]$ отличаются не более, чем в одной ячейке: $x[i+1, l_i] = a_i \leqslant x[i, l_i]$. В любой момент мы используем только один из массивов $x[i]$, поэтому будем использовать одномерный массив вместо двумерного; на каждом шаге у него нужно изменить не более одной ячейки.

```
fill(l, 1)
fill(x, inf)
x[1] = a[0]
for i = 1..(n - 1):
  l[i] = lower_bound(x + 1, x + n, a[i]) - x
  x[l[i]] = a[i]
```

Восстановление ответа

Если нужно восстановить ответ, вместе с массивом x будем поддерживать массив списков pos: вначале все списки пустые; в момент, когда мы делаем присвоение $x[l_i] = a_i$, допишем i в конец списка $pos[l_i]$. Теперь для того, чтобы найти предыдущий элемент в НВП, заканчивающейся в i-й позиции, нужно найти максимальный элемент в $pos[l_i]$, меньший i. Поскольку суммарный размер списков pos не превышает n, это можно делать простым проходом по списку.

3.5 Задача о рюкзаке

Есть рюкзак (knapsack) вместимости W, а также n предметов веса $w_1, \ldots, w_n$ и стоимости $p_1, \ldots, p_n$. Необходимо поместить в рюкзак множество предметов как можно большей суммарной стоимости. Более формально, нужно найти такое подмножество предметов $A \subset \{1, \ldots, n\}$, что $\sum_{i \in A} w_i \leqslant W$, а значение $\sum_{i \in A} p_i$ максимально возможно.

Существует много различных вариаций этой задачи. Можно, например, считать, что есть лишь одна копия каждого предмета (версия "без повторений"), а можно — что копий каждого предмета бесконечно много (версия "с повторениями").

Версия "с повторениями"

Пусть $k[j]$ — максимальная возможная суммарная стоимость, если рюкзак имеет вместимость j. Начальное состояние: $k[0] = 0$. Переход заключается в выборе последнего взятого предмета:

$$k[j] = \max_{w_i \leqslant j}(k[j - w_i] + p_i)$$

(максимум по пустому множеству считаем равным нулю). Ответ на задачу — $\max_{0 \leqslant j \leqslant W} k[j]$. Получаем решение за $O(nW)$ с $O(W)$ дополнительной памяти. Восстановление ответа делается стандартными методами.

```
fill(k, 0)
for j = 0..W:
  for i = 1..n:
    if w[i] <= j:
      k[j] = max(k[j], k[j - w[i]] + p[i])
```

Версия "без повторений"

Для того, чтобы не взять один предмет несколько раз, введём дополнительный параметр. Пусть $k[i, j]$ — максимальная возможная суммарная стоимость, если разрешается брать только предметы с номерами не больше i, а рюкзак имеет вместимость j. Начальные состояния: $k[0, j] = k[i, 0] = 0$. Переход: либо i-й предмет входит в ответ (то есть $k[i, j] = k[i - 1, j - w_i]$; этот вариант возможен только при $j \geqslant w_i$), либо не входит (то есть $k[i, j] = k[i - 1, j]$). Получаем решение за $O(nW)$ с такой же оценкой используемой дополнительной памяти. .

```
fill(k, 0)
for i = 1..n:
  for j = 0..W:
    k[i, j] = k[i - 1, j]
    if w[i] <= j:
      k[i, j] = max(k[i, j], k[i - 1, j - w[i]] + p[i])
```

Оптимизация памяти

Заметим, что при вычислении $k[i, j]$ алгоритму могут понадобиться только значения $k[i - 1, q]$ с $q \leqslant j$. Сделаем следующий трюк: будем вычислять значения $k[i, j]$ при фиксированном i в порядке убывания j (а не возрастания, как раньше). Тогда можно обойтись одномерным массивом: при обработке состояния (i, j) в $k[0..j]$ будем хранить значения $k[i - 1, 0], \ldots, k[i - 1, j]$, а в $k[j + 1..W]$ — значения $k[i, j + 1], \ldots, k[i, W]$. Получается, достаточно $O(W)$ дополнительной памяти. При этом получившийся код отличается от решения версии задачи "с повторениями" только порядком перебора значений j.

```
fill(k, 0)
for i = 1..n:
  for j = W..0:
    if w[i] <= j:
      k[j] = max(k[j], k[j - w[i]] + p[i])
```

Восстановление ответа

В версии с двумерным массивом восстановление ответа делается стандартными методами. В версии с одномерным массивом ответ просто так восстановить не удастся; зато можно применить идею алгоритма Хиршберга и восстановить ответ, используя $O(W + \log n)$ дополнительной памяти, и не ухудшив оценку времени работы.

3.6 Произведение матриц

Пусть мы хотим вычислить произведение n матриц $A_1 \times \cdots \times A_n$; матрица A_i имеет размер $m_{i-1} \times m_i$. Сами умножения мы будем производить, пользуясь определением произведения матриц, то есть на вычисление произведения матриц размеров $m \times n$ и $n \times p$ мы потратим $O(mnp)$ действий. Однако мы вольны выбирать порядок, в котором будем производить умножения (то есть расставлять скобки в выражении выше). Хочется минимизировать суммарное количество действий, которые придётся сделать, чтобы вычислить произведение всех матриц.

Пусть, например, мы перемножаем матрицы размеров 100×10, 10×20 и 20×2. Если производить вычисления в порядке $(A_1 \times A_2) \times A_3$, то понадобится совершить $100 \cdot 10 \cdot 20 + 100 \cdot 20 \cdot 2 = 24000$ действий; если же использовать порядок $A_1 \times (A_2 \times A_3)$, понадобится всего $10 \cdot 20 \cdot 2 + 100 \cdot 10 \cdot 2 = 2400$ действий.

Динамическое программирование по подотрезкам

Как решить эту задачу методом динамического программирования? Попробуем свести задачу к подзадачам попроще. Посмотрим на последнюю операцию умножения, которую мы проведём: мы перемножим матрицы $A_1 \times \cdots \times A_j$ и $A_{j+1} \times \cdots \times A_n$ для некоторого j. Операции, которые мы сделали до этого, соответствуют оптимальному вычислению этих двух матриц: получились две независимых подзадачи.

Таким образом, состояния на этот раз соотвествуют не только префиксам (как во многих рассмотренных ранее задачах), но и суффиксам последовательности, соответствующей текущей задаче. Если же мы попытаемся свести задачу для префикса или суффикса к подзадачам попроще, то нам понадобится решать подзадачи уже для *подотрезков* последовательности, соответствующей исходной задаче.

Итак, на этот раз состояния соответствуют подотрезкам исходной задачи. Пусть $c[l, r]$ — минимальное количество операций, необходимое для вычисления $A_l \times \cdots \times A_r$. Начальные состояния: $c[i, i] = 0$. Для того, чтобы выписать переходы, переберём, какую пару матриц мы будем умножать в самом конце:

$$c[l, r] = \min_{l \leqslant i < r} (c[l, i] + c[i + 1, r] + m[l - 1] \cdot m[i] \cdot m[r]).$$

Один из порядков, соответствующий топологической сортировке на графе состояний — рассматривать состояния в порядке увеличения длины подотрезка.

Получилось решение с временем работы $O(n^3)$, использующее $O(n^2)$ дополнительной памяти. Восстановление ответа делается стандартными методами.

R Эту задачу умеют решать за $O(n \log n)$ (Hu, Shing, 1984).

```
fill(c, inf)
for i = 1..n:
  c[i, i] = 0
for s = 1..n:
  for l = 1..(n - s):
    r = l + s # подзадача [l, r]
    for i = l..(r - 1):
      c[l, r] = min(c[l, r],
                    c[l, i] + c[i + 1, r] + m[l - 1] * m[i] * m[r])
```

3.7 Независимое множество максимального веса в дереве

Множество вершин в графе *независимо (independent)*, если никакая пара вершин в нём не соединена ребром.

Дано дерево, каждая вершина v которого имеет вес w_v. Необходимо найти независимое множество максимального суммарного веса.

> **R** Невзвешенная версия этой задачи (все веса равны единице) решается жадным алгоритмом.

Динамическое программирование по поддеревьям

Выберем любую вершину r в качестве корня дерева. Теперь каждой вершине дерева v соответствует подзадача поиска независимого множества максимального веса в поддереве этой вершины; обозначим искомое множество за I_v. Как выразить решение подзадачи для вершины v через подзадачи попроще? Если сама вершина v не входит в I_v, то I_v — объединение I_u по всем u — детям v. Если же v входит в I_v, то никто из детей v в I_v входить не может. В этом случае I_v — объединение I_u по всем u — "внукам" v — детям детей v.

Пусть

$$I[v] = \sum_{u \in I_v} w_u.$$

Удобно также ввести вспомогательную величину

$$J[v] = \sum_{u \text{ — ребёнок } v} I[u].$$

Из рассуждений выше получаем

$$I[v] = \max\left(J[v], w_v + \sum_{u \text{ — ребёнок } v} J[u]\right).$$

Теперь все значения I, J легко вычислить, запустив поиск в глубину из r. Получаем решение за $O(V + E)$.

```
dfs(v, parent):
  I[v] = w[v], J[v] = 0
  for u in to[v]:
    if u == parent:
      continue
    dfs(u, v)
    I[v] += J[u], J[v] += I[u]
  I[v] = max(I[v], J[v])

dfs(0, -1) # r = 0
```

Восстановление ответа

```
# skip = True, если вершина v точно не входит в ответ
restoreSet(v, parent, skip):
  childSkip = False
  if (not skip) and I[v] > J[v]: # v входит в ответ
    res.push_back(v)
    childSkip = True # дети v в ответ не входят
  for u in to[v]:
    if u == parent:
      continue
    restoreSet(u, v, childSkip)

restoreSet(0, -1, False)
```

3.8 Динамическое программирование по подмножествам

Задача коммивояжёра

Коммивояжёр (travelling salesman) хочет посетить каждый из n городов по одному разу и вернуться в свой родной город. Он знает расстояния между всеми парами городов. Какой порядок посещения городов оптимален?

Более формально: дан полный (есть ребро между каждой парой вершин) взвешенный неориентированный граф на n вершинах; необходимо найти гамильтонов цикл (цикл, проходящий по каждой вершине ровно один раз), имеющий минимально возможную длину. Расстояние между вершинами u и v будем обозначать как $d_{u,v}$.

Всего различных маршрутов $(n-1)!$, поэтому перебор их всех возможен лишь при небольших n. Динамическим программированием эту задачу можно решить за $O(2^n \cdot n^2)$, что уже значительно быстрее (хотя оценка времени всё ещё экспоненциальна).

Естественная подзадача в данном случае — задача нахождения "префикса" маршрута (первых нескольких рёбер маршрута, начиная из родного города коммивояжёра; будем считать, что этот город имеет номер 0). Как можно закодировать начало маршрута? Для того, чтобы префикс маршрута можно было попытаться продлить (или, наоборот, рассмотреть префикс без последнего ребра), необходимо знать, какие вершины входят в префикс, а также какая из них является последней. Таким образом, состояние параметризуется парой (A, v), где A — подмножество вершин графа, содержащее вершину 0, $v \in A$ — номер последней вершины в префиксе маршрута.

Начальные состояния: $l[\{0\}, 0] = 0$; $l[A, 0] = \infty$ при $|A| > 1$. Для перехода нужно перебрать последнее ребро префикса маршрута: пусть $v \neq 0$, $v \in A$, тогда

$$l[A, v] = \min_{\substack{u \in A,\\ u \neq v}} (l[A \setminus \{v\}, u] + d_{u,v}).$$

Для получения ответа на задачу нужно перебрать последнюю вершину маршрута: ответ равен

$$\min_{v>0} (l[A, v] + d_{v,0}).$$

Прежде, чем перейти к реализации, нужно научиться использовать множества в качестве индексов массива, а также быстро производить над множествами различные операции (например, удаление элемента из множества).

Работа с множествами

Пронумеруем все подмножества n-элементного множества $U = \{0, \ldots, n-1\}$ n-битными числами: $A \subset U$ сопоставим $mask(A) = \sum_{i \in A} 2^i$. Номер множества удобно использовать в качестве индекса массива; также с помощью битовых операций над числами можно быстро

осуществлять различные операции над множествами с соответствующими номерами. Приведём несколько примеров:

- $mask(A \cup B) = mask(A) \mid mask(B)$, где $\mid$ — битовое “ИЛИ”;
- $mask(A \cap B) = mask(A) \,\&\, mask(B)$, где $\&$ — битовое “И”;
- $mask(A \oplus B) = mask(A) \oplus mask(B)$, где $\oplus$ слева — симметрическая разность множеств, а $\oplus$ справа — битовый “XOR”;
- $mask(A \setminus B) = mask(A) \,\&\, {\sim} mask(B)$, где $\sim$ — битовое отрицание;
- $mask(U) = (1 \ll n) - 1$, где $\ll k$ — битовый сдвиг на k бит влево;
- $mask(\{k\}) = 1 \ll k$;
- $(mask(A) \gg k) \,\&\, 1$ — выражение, равное единице тогда и только тогда, когда $k \in A$ (здесь $\gg k$ — битовый сдвиг на k бит вправо).

В `C++` все битовые операторы, за исключением “XOR”, обозначаются теми же символами, что и выше; `^` — битовый “XOR” в `C++`. Заметим, что битовые операции над стандартными числовыми типами осуществляются за $O(1)$. Мы для простоты будем считать, что размер множества позволяет использовать стандартный тип данных для хранения номеров подмножеств, поэтому все приведённые выше выражения можно вычислить за $O(1)$.

Размер подмножеств

В качестве простого примера вычислим размер каждого подмножества $A \subset U$. Размер множества совпадает с количеством ненулевых бит в его номере. Количество единичных бит в числе x равняется сумме количества единичных бит в $\lfloor \frac{x}{2} \rfloor$ и значения младшего бита x.

```
1 size[0] = 0 # размер пустого множества равен 0
2 for mask = 1..((1 << n) - 1):
3   size[mask] = size[mask >> 1] + (mask & 1)
```

> **R** В компиляторе `C++ GCC` есть встроенная функция `__builtin_popcount(x)`, возвращающая число бит в числе x типа `int` (есть подобные функции и для других стандартных типов); она работает за $O(1)$, но значительно медленнее, чем, например, элементарные битовые операции.

Сумма элементов в подмножестве

Пусть есть n чисел $a_0, \ldots, a_{n-1}$; необходимо для каждого подмножества $A \subset \{0, \ldots, n-1\}$ вычислить $s(A) = \sum_{i \in A} a_i$. Можно вычислить каждую сумму по определению:

```
1 for mask = 0..((1 << n) - 1):
2   s[mask] = 0
3   for i = 0..(n - 1):
4     if ((mask >> i) & 1): # i лежит в множестве с номером mask
5       s[mask] += a[i]
```

Это потребует $O(2^n \cdot n)$ времени. Можно справиться быстрее с помощью простого динамического программирования: пусть $x \in A$ — какой-нибудь элемент A, тогда $s(A) = s(A \setminus \{x\}) + a_x$. При этом, поскольку мы вычисляем значения s в порядке возрастания номеров множеств, а $mask(A \setminus \{x\}) < mask(A)$, $s(A \setminus \{x\})$ уже будет известно к моменту вычисления $s(A)$. В качестве x удобно использовать максимальный элемент A: такой элемент соответствует старшему биту $mask(A)$, номер которого может только возрастать при возрастании номера. Получаем следующее решение за $O(2^n)$:

```
1 x = 0
2 s[0] = 0
3 for mask = 1..((1 << n) - 1):
4   if mask == (1 << (x + 1)): # номер старшего бита увеличился
5     x += 1
6   s[mask] = s[mask ^ (1 << x)] + a[x]
```

Сумма по подмножествам

Пусть теперь для каждого подмножества $A \subset \{0, \ldots, n-1\}$ дано число f_A, и требуется для каждого A вычислить $s(A) = \sum_{B \subset A} f_B$.

Можно для каждого A перебрать все возможные B и проверить, какие из них являются подмножествами A; получится решение за $O(2^n \cdot 2^n) = O(4^n)$.

```
1 for A = 0..((1 << n) - 1):
2   g[A] = 0
3   for B = 0..((1 << n) - 1):
4     if (A & B) == B: # B - подмножество A
5       g[A] += f[B]
```

Можно действовать аккуратнее, и для фиксированного A сразу перебирать только B, являющиеся подмножествами A. Будем перебирать подмножества A в порядке уменьшения их номеров. Пусть $A = \{a_0, \ldots, a_{k-1}\}$, $a_0 < \cdots < a_{k-1}$. Если мы мысленно сдвинем разряды $a_0, \ldots, a_{k-1}$ на позиции $0, \ldots, k-1$, то номера подмножеств A перейдут в числа от 0 до $2^k - 1$. Значит, мы хотим перебрать числа от 0 до $2^k - 1$ в порядке убывания, только со сдвинутыми разрядами: нулевой разряд нужно сдвинуть в a_0-й,. . . ,$(k-1)$-й разряд — в a_{k-1}-й.

Для того, чтобы по числу i получить число $i-1$, нужно занулить младший единичный бит i, а во все биты младше его записать единицу. Значит, для того, чтобы получить по подмножеству $B \subset A$ следующее подмножество A в порядке убывания номеров, нужно удалить из B минимальный элемент $x \in B$ и добавить все элементы из $A \setminus B$, меньшие x. Это соответствует формуле $(mask(B) - 1) \,\&\, mask(A)$.

```
1 for A = 0..((1 << n) - 1):
2   g[A] = f[0] # учли пустое подмножество
3   for (B = A; B > 0; B = ((B - 1) & A)): # все непустые подмножества A
4     g[A] += f[B]
```

Время работы полученного решения пропорционально количеству пар (A, B) таких, что $B \subset A$. Количество таких пар равняется 3^n. Это можно понять следующим способом: множество A размера k имеет 2^k различных подмножеств, количество множеств размера k равно $\binom{n}{k}$, значит, всего пар (A, B)

$$\sum_{k=0}^{n} \binom{n}{k} \cdot 2^k = \sum_{k=0}^{n} \binom{n}{k} \cdot 2^k \cdot 1^{n-k} = (2+1)^n = 3^n.$$

Альтернативный способ: если пара (A, B) зафиксирована, то каждый элемент находится в одном из трёх состояний: лежит в A, лежит в $A \setminus B$ или же не лежит в A. Всего элементов n, значит, всего 3^n вариантов.

Итак, мы получили решение за $O(3^n)$. Можно воспользоваться динамическим программированием и решить задачу ещё быстрее. Пусть

$$g[A, k] = \sum_{\substack{B \subset A, \\ A \setminus B \subset \{0, \ldots, k-1\}}} f_B.$$

Нас интересуют значения $g[A, n] = g(A)$. Начальные значения: $g[A, 0] = f_A$. Переход:

$$g[A, k+1] = \begin{cases} g[A, k], & \text{если } k \notin A, \\ g[A, k] + g[A \setminus \{k\}, k], & \text{если } k \in A. \end{cases}$$

Почему это так? $g[A, k+1]$ — сумма f_B по подмножествам $B \subset A$, которые могут отличаться от A только принадлежностью элементов $0, \ldots, k$. Если $k \notin A$, все такие B отличаются от A только принадлежностью $0, \ldots, k-1$, значит, $g[A, k+1] = g[A, k]$. Если же $k \in A$, то все такие B делятся на две группы: те, которые содержат k, и те, которые не содержат. Сумма по первой группе равна $g[A, k]$. Все множества во второй группе являются подмножествами $A \setminus \{k\}$; сумма по второй группе равна $g[A \setminus \{k\}, k]$.

Заметим, что при вычислении состояния $(A, k+1)$ нас интересуют только состояния (A, k), и, возможно $(A \setminus \{k\}, k)$. Однако мы пользуемся значением $g[A \setminus \{k\}, k]$ только в случае, когда $g[A \setminus \{k\}, k] = g[A \setminus \{k\}, k+1]$. Значит, можно обойтись одномерным массивом вместо двумерного. Получаем решение за $O(2^n \cdot n)$, использующее $O(2^n)$ памяти.

```
for mask = 0..((1 << n) - 1):
  g[mask] = f[mask]
for k = 0..(n - 1):
  for mask = 0..((1 << n) - 1):
    if (mask >> k) & 1:
      g[mask] += g[mask ^ (1 << k)]
```

Реализация решения задачи коммивояжёра

```
fill(l, inf)
l[1, 0] = 0 # 1 - номер множества 0
# номера множеств, содержащих 0 - все нечётные числа
for (mask = 1; mask < (1 << n); mask += 2):
  for v = 1..(n - 1): # перебираем последнюю вершину префикса маршрута
    if ((mask >> v) & 1) == 0: # v не лежит в префиксе
      continue
    for u = 0..(n - 1): # перебираем второй конец последнего ребра
      if u == v or ((mask >> u) & 1) == 0:
        continue
      l[mask, v] = min(l[mask, v], l[mask ^ (1 << v), u] + d[u, v])
res = inf
for v = 1..(n - 1):
  res = min(res, l[(1 << n) - 1, v] + d[v, 0])
```

Восстановление ответа осуществляется стандартными методами.

3.9 Генерация номера по объекту и объекта по номеру

Научимся сопоставлять комбинаторному объекту (например, перестановке) его номер в списке всех возможных объектов (списке всех перестановок фиксированной длины), упорядоченном лексикографически; или, наоборот, генерировать объект по его номеру.

Это бывает полезно, если хочется закодировать объект как можно меньшим количеством бит, а также если хочется использовать объект в качестве индекса массива (например, если при применении метода динамического программирования состояние описывается комбинаторным объектом).

Номер по перестановке

Как вычислить номер перестановки $p_1, \ldots, p_n$ чисел $1, \ldots, n$ в списке всех перестановок длины n, упорядоченных лексикографически?

0	1,2,3
1	1,3,2
2	2,1,3
3	2,3,1
4	3,1,2
5	3,2,1

Рис. 3.4: Упорядоченный лексикографически список перестановок длины 3

Номер перестановки p — это количество перестановок, лексикографически меньших p. Пусть q — одна из таких перестановок. Тогда $q_1 = p_1, \ldots, q_{k-1} = p_{k-1}$, $x = q_k < p_k$ для некоторого k. При этом для фиксированных k, $x < p_k$ существует $(n-k)!$ различных перестановок q, удовлетворяющих условиям выше (так как оставшиеся $n-k$ чисел могут идти в любом порядке).

Получаем следующий алгоритм: переберём k — позицию самого левого несовпадения q и p, переберём $x < p_k$ — значение q_k, и просуммируем количество перестановок q по всем таким k, x.

```
getNum(p, n):
  num = 0
  fill(used, False) # список элементов p[1..k)
  for k = 1..n:
    for x = 1..(p[k] - 1):
      if used[x]:
        continue
      num += factorial[n - k]
    used[p[k]] = True
  return num
```

Тот же метод (перебрать позицию и значение первого несовпадения, просуммировать по ним количество способов "закончить" объект) работает и со многими другими комбинаторными объектами. Разберём ещё один пример — поиск номера правильной скобочной последовательности.

Правильные скобочные последовательности

Скобочная последовательность — это любая последовательность символов '(' (открывающая скобка) и ')' (закрывающая скобка).

Балансом скобочной последовательности будем называть разность количества открывающих и закрывающих скобок в ней.

Правильная скобочная последовательность (ПСП) — это такая скобочная последовательность из $2n$ символов, что баланс любого её префикса неотрицателен, а баланс всей последовательности равен нулю.

0	(((())))	7	(())(())
1	((()()))	8	(())()()
2	((())())	9	()((()))
3	((()))()	10	()(()())
4	(()(()))	11	()(())()
5	(()()())	12	()()(())
6	(()())()	13	()()()()

Рис. 3.5: Упорядоченный лексикографически список ПСП длины 8

Номер по ПСП

Будем искать номер ПСП a длины $2n$ тем же методом: пусть b лексикографически меньше a, причём $b_1 = a_1, \ldots, b_{k-1} = a_{k-1}$; ‘(‘ $= b_k < a_k =$ ‘)‘. Сколько существует таких ПСП b?

Поскольку баланс каждого префикса b неотрицателен, баланс каждого суффикса b неположителен. Тогда, если перевернуть суффикс $b_{k+1}, \ldots, b_{2n}$ и заменить открывающие скобки на закрывающие, а закрывающие — на открывающие, то получится скобочная последовательность длины $2n - k$, баланс любого префикса которой неотрицателен, а баланс её целиком совпадает с балансом последовательности $b_1, \ldots, b_k$. Предподсчитаем количество таких последовательностей для каждой длины и каждого значения баланса динамическим программированием (переход — перебор типа последней скобки).

```
fill(cnt, 0)
for l = 1..(2 * MAXN):
  for b = 0..l:
    cnt[l, b] = cnt[l - 1, b + 1] # закрывающая скобка
    if b > 0:
      cnt[l, b] += cnt[l - 1, b - 1] # открывающая скобка

getNum(a, n):
  num = 0
  b = 0 # баланс
  for k = 1..(2 * n):
    if a[k] == ')':
      num += cnt[2 * n - k, b + 1]
      b -= 1
    else:
      b += 1
  return num
```

Перестановка по номеру

Вернёмся к перестановкам и решим обратную задачу: пусть нужно найти перестановку, имеющую заданный номер num в списке лексикографически упорядоченных перестановок длины n.

Для любого x от 1 до n существует $(n-1)!$ перестановок, начинающихся с x. Тогда первый символ искомой перестановки — минимальное x такое, что $x \cdot (n-1)! > num$. Остаётся найти перестановку с номером $num - x \cdot (n-1)!$ в списке перестановок, начинающихся с x. Для этого будем повторять те же рассуждения, но для следующих символов (второго, третьего, и так далее; надо не забыть, что один символ не может встретиться в перестановке несколько раз).

```
getPermutation(num, n):
  fill(used, False)
  vector<int> p
  for k = 1..n:
    for x = 1..n:
      if used[x]:
        continue
      if num < factorial[n - k]:
        p.push_back(x), used[x] = True
        break
      num -= factorial[n - k]
  return p
```

Тот же метод опять работает и со многими другими комбинаторными объектами. В качестве примера вернёмся к ПСП.

ПСП по номеру

Будем ставить на очередную позицию открывающую скобку, если номер ПСП меньше количества способов закончить ПСП после постановки этой скобки; иначе будем ставить закрывающую скобку и уменьшать номер на количество "пропущенных" ПСП (тех, в которых на этой позиции стоит открывающая скобка).

```
getCBS(num, n):
  b = 0
  vector<char> a
  for k = 1..(2 * n):
    if num < cnt[2 * n - k, b + 1]:
      a.push_back('('), b += 1
    else:
      num -= cnt[2 * n - k, b + 1]
      a.push_back(')'), b -= 1
  return a
```

3.10 Решение рекуррентных соотношений возведением матрицы в степень

Пусть дано рекуррентное соотношение $f_j = a_1 f_{j-1} + \dots + a_k f_{j-k} + b$, а также начальные значения $f_0, \dots, f_{k-1}$; необходимо найти f_n.

Простой способ — применить метод динамического программирования: вычислить все значения f_j по очереди за $O(nk)$. Для того, чтобы получить более быстрое решение, выпишем рекуррентное соотношение в матричном виде:

$$\begin{pmatrix} f_j \\ f_{j-1} \\ f_{j-2} \\ \vdots \\ f_{j-k+1} \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & \cdots & a_{k-1} & a_k & b \\ 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_{j-1} \\ f_{j-2} \\ f_{j-3} \\ \vdots \\ f_{j-k} \\ 1 \end{pmatrix}.$$

Тогда

$$\begin{pmatrix} f_n \\ f_{n-1} \\ f_{n-2} \\ \vdots \\ f_{n-k+1} \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & \cdots & a_{k-1} & a_k & b \\ 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}^{n-k+1} \cdot \begin{pmatrix} f_{k-1} \\ f_{k-2} \\ f_{k-3} \\ \vdots \\ f_0 \\ 1 \end{pmatrix}.$$

Для вычисления матрицы применим ту же идею, с помощью которой мы быстро вычисляли степень числа по модулю: с её помощью $(n-k+1)$-ю степень матрицы $(k+1) \times (k+1)$ можно вычислить за $O(k^3 \log n)$. После этого найти f_n можно за $O(k)$ (здесь мы для простоты считаем, что все операции с числами можно осуществлять за $O(1)$).

Пример: числа Фибоначчи

Применим этот метод к числам Фибоначчи. Поскольку рекуррентное соотношение для чисел Фибоначчи не содержит свободного члена, можно обойтись без последних строки и столбца матрицы.

$$\begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

С учётом того, что f_n имеет длину $\Theta(n)$, получаем рекуррентное соотношение на время работы алгоритма $T(n) = T(n/2) + M(n)$, где $M(n)$ — сложность умножения двух чисел длины n. Если воспользоваться алгоритмом Карацубы, получаем соотношение $T(n) = T(n/2) + O(n^{\log_2 3})$, откуда по основной теореме о рекуррентных соотношениях $T(n) = O(n^{\log_2 3})$.

Введение в теорию сложности вычислений

4 Классы сложности P и NP 35
4.1 Задачи поиска
4.2 Decision-задачи
4.3 Класс P
4.4 Класс NP

5 NP-полные задачи 37
5.1 Сведение по Карпу
5.2 NP-трудные и NP-полные задачи
5.3 SAT → 3-SAT
5.4 3-SAT → Задача о независимом множестве
5.5 Задача о независимом множестве → Задача о вершинном покрытии
5.6 Задача о вершинном покрытии → Задача о покрытии множества
5.7 Задача о независимом множестве → Задача о клике
5.8 3-SAT → Задача о трёхдольном сочетании
5.9 Задача о трёхдольном сочетании → Уравнение в нулях и единицах
5.10 Уравнение в нулях и единицах → Целочисленное линейное программирование
5.11 Уравнение в нулях и единицах → Задача о сумме подмножества
5.12 Задача о сумме подмножества → Задача о рюкзаке
5.13 Уравнение в нулях и единицах → Задача о гамильтоновом цикле
5.14 Задача о гамильтоновом цикле → Задача о гамильтоновом пути
5.15 Задача о гамильтоновом цикле → Задача коммивояжёра
5.16 Любая задача из NP → SAT

4. Классы сложности P и NP

4.1 Задачи поиска

Большая часть задач, которые мы решали до этого, являются *задачами поиска (search problems)* — в них требуется по входу найти какой-то объект: кратчайший путь в данном графе между данной парой вершин, гамильтонов цикл в данном графе, минимум в массиве, расстояние Левенштейна между последовательностями и т.п.

Более формально задача поиска устроена так: существует некоторое бинарное отношение $R(x, y)$, и требуется по входу x найти такое y, что $R(x, y)$ истинно, либо сказать, что таких y не существует. Здесь x и y — битовые строки произвольной длины, то есть $x, y \in \{0, 1\}^*$.

Например, в задаче поиска гамильтонова цикла x — граф, каким-то способом заданный в виде последовательности бит, а y — заданная битами последовательность вершин, и $R(x, y)$ истинно, если y соответствует гамильтонову циклу в графе, заданном x.

4.2 Decision-задачи

Нам будет удобнее изучать *decision-задачи* — в них требуется по входу $x \in \{0, 1\}^*$ выдать бинарный ответ: "да" или "нет". Примеры таких задач: существует ли в данном графе путь длины не более L между данной парой вершин; существует ли в данном графе гамильтонов путь; есть ли в массиве элемент, меньший x; верно ли, что расстояние Левенштейна между последовательностями не превосходит k. Например, задача существования гамильтонова цикла формально определяется так: дан граф, заданный в виде последовательности бит x; требуется вывести "да", если в графе есть гамильтонов цикл, и "нет" иначе.

Каждая decision-задача соответствует подмножеству входов $L \subset \{0, 1\}^*$, ответ на которые положителен. Такие подмножества также называют *формальными языками (formal languages)*; мы будем для краткости называть их просто языками.

Алгоритм, решающий decision-задачу, совпадает с алгоритмом, *распознающим* соответствующий этой задаче язык L, то есть по $x \in \{0, 1\}^*$ определяющий, лежит ли x в L. Дальше мы часто будем отождествлять decision-задачу и соответствующий ей язык.

4.3 Класс P

Класс сложности P — это множество языков L, для которых существует алгоритм, распознающий их за полиномиальное время. Чуть более формально: $L \in \mathrm{P}$, если существует алгоритм A, который распознаёт язык L, и такой многочлен $p(n)$, что для любого $n > 0$ на любом входе $x \in \{0, 1\}^*$, $|x| \leqslant n$, алгоритм A работает за время $O(p(n))$.

Задача проверки существования цикла в графе лежит в P: она решается алгоритмом поиска в глубину, который имеет время работы $O(V + E)$, что пропорционально длине битовой записи представления графа.

Decision-версия задачи о рюкзаке звучит так: необходимо определить, существует ли подмножество предметов, влезающее в рюкзак и имеющее суммарную стоимость хотя бы P. Мы пока умеем решать задачу лишь за $\Theta(nW)$, что может быть экспоненциально

больше длины входных данных (для битовой записи W требуется лишь $\log W$ бит). Про эту задачу неизвестно, лежит ли она в P.

R По задаче поиска, как правило, можно построить decision-задачу, умение решать которую в какой-то степени эквивалентно решению исходной задачи поиска. Рассмотрим, например, задачу о рюкзаке. Если бы мы умели решать за полиномиальное время её decision-версию, то смогли бы решить за полиномиальное время и задачу поиска.

Действительно, пусть мы имеем алгоритм, решающий decision-задачу за полиномиальное время. Сначала мы можем найти P_{opt} — максимально возможную сумму стоимостей предметов, помещающихся в рюкзак: сделаем двоичный поиск по P, внутри которого будем запускать алгоритм, решающий decision-задачу.

Теперь запустим алгоритм, решающий decision-задачу, на множестве предметов без самого первого. Если окажется, что на таком множестве нельзя набрать предметы стоимостью P_{opt}, то первый предмет точно входит в оптимальное решение; если можно, то существует оптимальное решение, в которое он не входит. Остаётся понять, как из множества предметов без первого набрать предметы суммарной стоимостью $P_{opt} - p_1$ в первом случае, или P_{opt} во втором. Сделаем это, применив те же рассуждения ко второму предмету, и так далее.

4.4 Класс NP

Класс сложности NP — это множество задач, решение которых можно *проверить* за полиномиальное время. Более формально: $L \in \mathrm{NP}$, если существуют алгоритм A и такие многочлены $p(n)$, $q(n)$, что для любого $x \in \{0,1\}^*$, $|x| \leqslant n$ и любого $y \in \{0,1\}^*$, $|y| \leqslant q(n)$ алгоритм A работает на входе (x, y) за время $O(p(n))$. При этом

- если $x \in L$, то существует такой $y \in \{0,1\}^*$, $|y| \leqslant q(n)$, что A на входе (x, y) выдаёт “да”;
- если $x \notin L$, то для любого $y \in \{0,1\}^*$, $|y| \leqslant q(n)$, A на входе (x, y) выдаёт “нет”.

Здесь y — своеобразная подсказка, или *сертификат (certificate)*, с помощью которого можно проверить, лежит ли x в L.

Скажем, в задаче о существовании гамильтонова цикла сертификатом может быть сама последовательность вершин, соответствующая гамильтонову циклу: по данной последовательности y легко проверить, соответствует ли она циклу на всех вершинах; с другой стороны, если в графе гамильтонова цикла нет, то и такого сертификата не существует. Значит, задача о существовании гамильтонова цикла лежит в NP.

Decision-версия задачи о рюкзаке также лежит в NP: в данном случае сертификат — это список элементов подмножества, влезающего в рюкзак и имеющего суммарную стоимость хотя бы P.

R P — это сокращение от “polynomial time”. NP же — это сокращение от “nondeterministic polynomial time”. При чём тут недетерминированность? Можно взглянуть на определение класса NP под другим углом: пусть алгоритм A из определения получает на вход x и **произвольную** последовательность бит y. Тогда $x \in L$, если существует хотя бы один вариант развития событий, при котором A вернёт “да”.

Наверное, самый известный открытый вопрос в теории сложности — верно ли, что $\mathrm{P} = \mathrm{NP}$? Другими словами, правда ли, что если решение задачи можно проверить за полиномиальное время, то это решение можно найти за полиномиальное время? В настоящее время большинство учёных считают, что это не так, то есть что существуют задачи в NP, которые нельзя решить за полиномиальное время.

5. NP-полные задачи

5.1 Сведение по Карпу

Язык L_1 *сводится по Карпу* к языку L_2 ($L_1 \leqslant_P L_2$), если существует такая функция $f : \{0,1\}^* \to \{0,1\}^*$, что

- $x \in L_1$ тогда и только тогда, когда $f(x) \in L_2$;
- существует многочлен $p(n)$ и алгоритм A, для любого $x \in \{0,1\}^*$, $|x| \leqslant n$, вычисляющий $f(x)$ за $O(p(n))$ (в частности, $|f(x)| = O(p(n))$).

Свойства сведения по Карпу

- **Транзитивность: если $L_1 \leqslant_P L_2$ и $L_2 \leqslant_P L_3$, то $L_1 \leqslant_P L_3$.**
  Действительно, пусть $f(x)$, $g(x)$ — соответствующие функции сведения, вычислимые за $O(p(|x|))$ и $O(q(|x|))$. Тогда $x \in L_1$ тогда и только тогда, когда $g(f(x)) \in L_3$, при этом для любого $x \in \{0,1\}^*$, $|x| \leqslant n$, значение $g(f(x))$ можно вычислить за $O(q(p(n)))$.
- **Если $L_1 \leqslant_P L_2$ и $L_2 \in \mathrm{P}$, то $L_1 \in \mathrm{P}$.**
  Для того, чтобы проверить, лежит ли x в L_1, вычислим значение функции сведения $f(x)$ за время, полиномиальное от длины x. После этого за полиномиальное от длины $f(x)$ (то есть и от длины x) время проверим, лежит ли $f(x)$ в L_2.
- **Если $L_1 \leqslant_P L_2$ и $L_2 \in \mathrm{NP}$, то $L_1 \in \mathrm{NP}$.**
  Пусть $f(x)$ — функция сведения, вычислимая за $O(p_1(|x|))$; пусть A_2, $p_2(n)$, $q_2(n)$ — алгоритм и многочлены из определения $L_2 \in \mathrm{NP}$. Тогда для любых $x, y \in \{0,1\}^*$, $|x| \leqslant n$, $|y| \leqslant q_2(p_1(n))$, алгоритм, подставляющий на вход алгоритму A пару $(f(x), y)$, будет работать за $O(p_1(n) + p_2(p_1(n)))$. При этом, если $x \in L_1$, то найдётся такой $y \in \{0,1\}^*$, $|y| \leqslant q_2(p_1(n))$, что алгоритм на входе $(f(x), y)$ выдаст "да"; если же $x \notin L_1$, то алгоритм выдаст "нет" на входе $(f(x), y)$ для любого $y \in \{0,1\}^*$, $|y| \leqslant q_2(p_1(n))$.

Таким образом, сведение по Карпу — способ сравнения относительной сложности задач.

5.2 NP-трудные и NP-полные задачи

Задача A NP-*трудная* (NP-*hard*), если любая задача из NP сводится к ней по Карпу: для любой задачи $B \in \mathrm{NP}$ верно $B \leqslant_P A$.

Задача A NP-*полная* (NP-*complete*), если она NP-трудная, и при этом сама лежит в NP. NP-полные задачи — в каком-то смысле, самые сложные задачи в классе NP. Если мы научимся решать хотя бы одну NP-полную задачу за полиномиальное время, то мы научимся решать за полиномиальное время все задачи из NP сразу, то есть докажем, что $\mathrm{P} = \mathrm{NP}$.

Вообще говоря, из определения неочевидно, существует ли хотя бы одна NP-полная задача. Мы покажем (с некоторыми оговорками), что SAT — задача проверки выполнимости конъюнктивной нормальной формы — является NP-полной. Также мы построим

цепочки сведений по Карпу из задачи SAT в ряд других задач, лежащих в классе NP, из чего будет следовать, что все эти задачи также являются NP-полными.

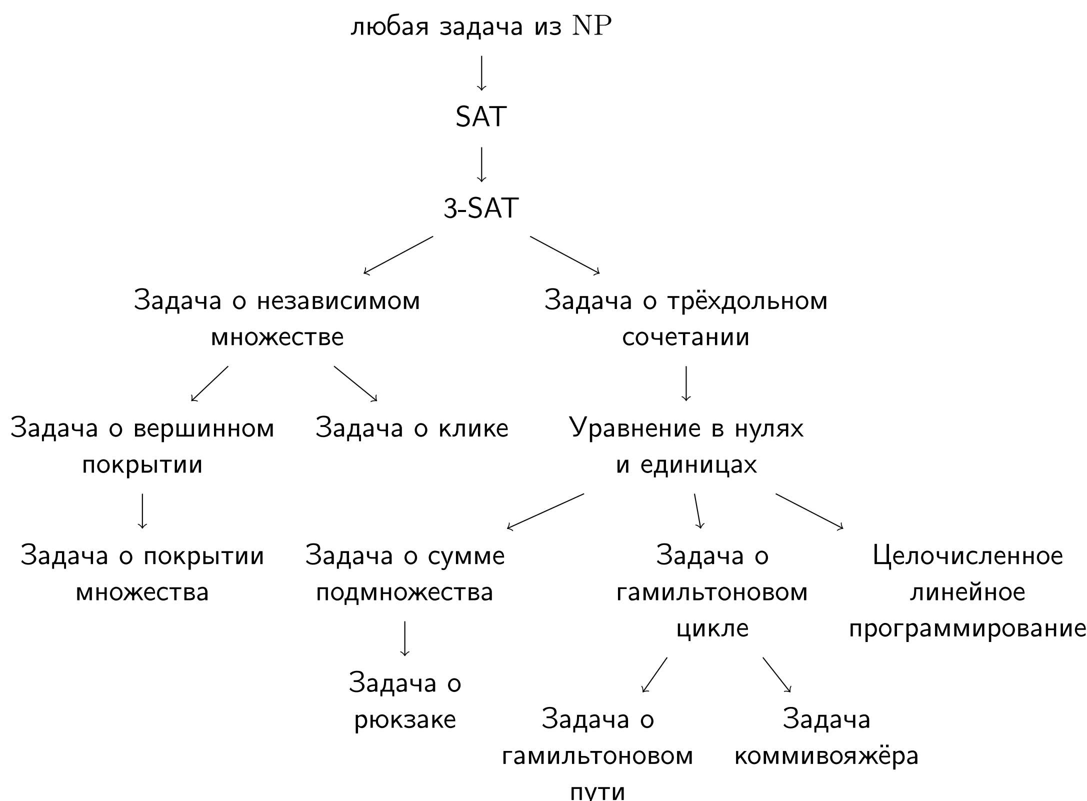


Рис. 5.1: Граф сведений

5.3 SAT $\to$ 3-SAT

В задаче SAT (boolean satisfiability problem) требуется по данной КНФ проверить, является ли она выполнимой. SAT $\in \mathrm{NP}$: сертификат — набор значений переменных, при которых значение формулы истинно.

Задача 3-SAT отличается тем, что каждый дизъюнкт в КНФ состоит не более, чем из трёх литералов (такие КНФ будем называть 3-КНФ). Ясно, что 3-SAT — частный случай SAT. Оказывается, тем не менее, что задача SAT не сложнее задачи 3-SAT: покажем, что SAT $\leqslant_P$ 3-SAT.

Для этого необходимо научиться по любой КНФ A за полиномиальное от её размера время строить 3-КНФ $f(A)$ такую, что A выполнима тогда и только тогда, когда выполнима $f(A)$. Будем делать это следующим итеративным процессом: пусть в A есть хотя бы один дизъюнкт, состоящий из более чем трёх литералов: $(a_1 \vee a_2 \vee \cdots \vee a_k)$, $k \geqslant 4$ (здесь a_i — литералы, а не переменные). Введём новую переменную x и заменим этот дизъюнкт на два: $(a_1 \vee a_2 \vee x) \wedge (\neg x \vee a_3 \vee \cdots \vee a_k)$. При фиксированных значениях $a_1, \ldots, a_k$ значение старого дизъюнкта истинно тогда и только тогда, когда найдётся такое значение x, что оба новых дизъюнкта истинны. Будем повторять эту операцию, пока не получим 3-КНФ.

Количество шагов не превышает размер исходной формулы, каждый шаг можно осуществить за не более чем линейное от размера исходной формулы время, значит, описанная функция f вычислима за полиномиальное от размера A время.

5.4 3-SAT → Задача о независимом множестве

В decision-версии задачи о независимом множестве требуется проверить, существует ли в данном неориентированном графе независимое множество размера хотя бы k. Задача лежит в NP: сертификат — список элементов такого множества. Построим сведение 3-SAT к этой задаче.

Построим по 3-КНФ следующий граф: каждому литералу в каждом дизъюнкте сопоставим свою вершину, помеченную этим литералом; попарно соединим рёбрами вершины, соответствующие литералам каждого дизъюнкта. Кроме того, соединим рёбрами каждую пару вершин, помеченную противоположными литералами (x и $\neg x$).

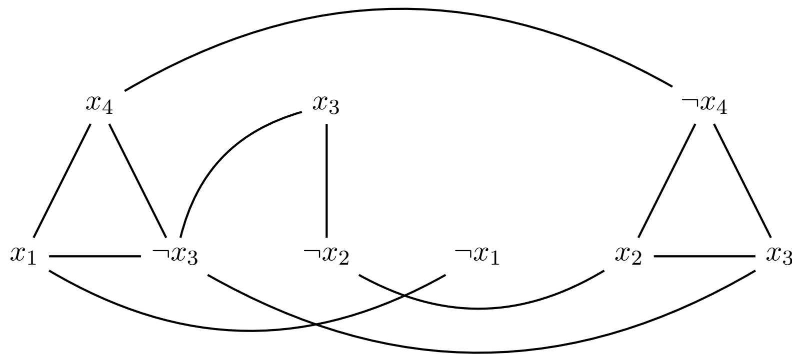


Рис. 5.2: Граф, соответствующий формуле $(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1) \wedge (x_2 \vee x_3 \vee \neg x_4)$

Граф можно построить по формуле за полиномиальное от её размера время. При этом формула выполнима тогда и только тогда, когда в построенном по ней графе есть независимое множество размера хотя бы k, где k — количество дизъюнктов в формуле. Действительно, среди вершин, соответствующим литералам одного дизъюнкта, можно взять в независимое множество не более одной. Значит, если можно одновременно взять в независимое множество k вершин, то, присвоив соответствующим им литералам значение `true` (и присвоив произвольные значения ещё не получившим значение переменным), мы получим выполняющий набор для формулы. Наоборот, по выполняющему набору легко построить независимое множество в графе размера k: нужно выбрать в каждом дизъюнкте любой литерал, имеющий значение `true`, и добавить соответствующую ему вершину в множество.

5.5 Задача о независимом множестве → Задача о вершинном покрытии

Вершинное покрытие (vertex cover) — такое подмножество вершин неориентированного графа U, что любое ребро графа инцидентно хотя бы одной вершине из U.

В задаче о вершинном покрытии требуется проверить, существует ли в данном графе вершинное покрытие размера не более k. Эта задача лежит в NP: сертификат — список вершин такого вершинного покрытия.

Заметим, что множество вершин U является вершинным покрытием в графе $G = (V, E)$ тогда и только тогда, когда $V \setminus U$ является независимым множеством. Таким образом, сведение задачи о независимом множестве к задаче о вершинном покрытии строится очень просто: нужно заменить k на $|V| - k$. Аналогично выглядит сведение в обратную сторону.

5.6 Задача о вершинном покрытии → Задача о покрытии множества

Задача о покрытии множества (set cover problem) формулируется следующим образом: даны конечное множество U; набор его подмножеств $S_1, \ldots, S_m$, объединение которых совпадает с U: $\bigcup_{i=1}^{m} S_i = U$; а также число k. Будем говорить, что подмножество S_i *покрывает* свои элементы. Необходимо проверить, существует ли такой поднабор подмножеств $S_{i_1}, \ldots, S_{i_k}$, что они покрывают все элементы U: $\bigcup_{j=1}^{k} S_{i_j} = U$. Задача лежит в NP: сертификат — список номеров подмножеств из искомого поднабора.

Задача о вершинном покрытии является частным случаем задачи о покрытии множества: если обозначить за U множество рёбер графа, а за S_i — множество рёбер, инцидентных вершине i, то получим задачу о покрытии множества. Сведение заключается в выписывании этих множеств.

5.7 Задача о независимом множестве → Задача о клике

Клика (clique) — подмножество вершин неориентированного графа, любые две из которых смежны. Также кликой называют подграф, индуцированный этим множеством вершин. В задаче о клике требуется проверить, существует ли в данном графе клика размера хотя бы k.

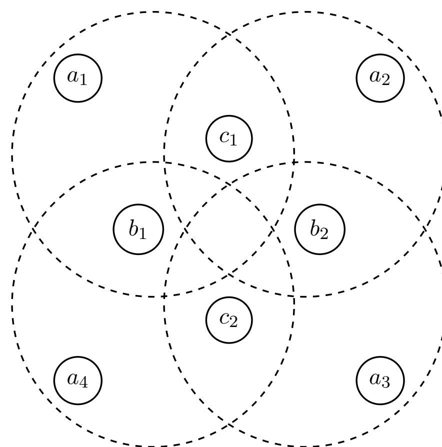
Дополнение (complement) графа G — граф на том же множестве вершин, что и G, в котором пара вершин смежна тогда и только тогда, когда она не смежна в G. Заметим, что множество вершин U является кликой в G тогда и только тогда, когда оно является независимым множеством в дополнении G. Это утверждение даёт сведения в обе стороны между задачей о независимом множестве и задачей о клике.

5.8 3-SAT → Задача о трёхдольном сочетании

Задача о трёхдольном сочетании (3-dimensional matching) формулируется так: даны конечные непересекающиеся множества A, B, C одинакового размера и набор троек вида (a, b, c): $a \in A$, $b \in B$, $c \in C$. Требуется проверить, можно ли покрыть все элементы непересекающимися тройками из набора (*3-сочетанием*). Задача лежит в NP: сертификат — список номеров искомых троек. Построим сведение по Карпу из 3-SAT в задачу о трёхдольном сочетании.

Гаджет для переменной

Рассмотрим такой набор из четырёх троек: (a_1, b_1, c_1), (a_2, b_2, c_1), (a_3, b_2, c_2), (a_4, b_1, c_2).



Если элементы b_1, b_2, c_1, c_2 не входят больше ни в какие тройки, то есть лишь два способа покрыть их 3-сочетанием: взять в него тройки (a_2, b_2, c_1) и (a_4, b_1, c_2), либо тройки (a_1, b_1, c_1) и (a_3, b_2, c_2). Мы сопоставим такую конструкцию (будем называть её *гаджетом (gadget)*) каждой переменной: первый вариант покрытия b_1, b_2, c_1, c_2 будет соответствовать значению `true`, второй — значению `false`.

Итак, для каждой переменной x_i из 3-КНФ заведём свой гаджет; входящие в него элементы обозначим за $a_{1,i}, a_{2,i}, a_{3,i}, a_{4,i}, b_{1,i}, b_{2,i}, c_{1,i}, c_{2,i}$. Теперь каждому набору значений переменных соответствует своё 3-сочетание, покрывающее все элементы вида $b_{j,i}, c_{j,i}$; при этом в i-м гаджете не покрыты элементы $a_{1,i}, a_{3,i}$, если x_i истинно, и не покрыты $a_{2,i}, a_{4,i}$ иначе. Теперь нужно как-то закодировать дизъюнкты.

Кодирование дизъюнктов

Для дизъюнкта $d = q_1 \vee \cdots \vee q_m$, $m \leqslant 3$ (q_i — литералы), заведём элементы b_d, c_d. Для каждого из литералов, входящих в дизъюнкт, добавим в набор одну новую тройку: если $q_j = x_i$, то добавим тройку $(a_{1,i}, b_d, c_d)$ либо тройку $(a_{3,i}, b_d, c_d)$ (какую именно из них, выберем чуть позже). Если же $q_j = \neg x_i$, то добавим тройку $(a_{2,i}, b_d, c_d)$ или тройку $(a_{4,i}, b_d, c_d)$ (опять же, позже выберем, какую именно). Теперь то, что дизъюнкт d выполнен, соответствует тому, что к 3-сочетанию, соответствующему набору значений переменных, можно добавить тройку, покрывающую b_d, c_d.

Теперь хочется, чтобы выполняющий формулу набор значений переменных соответствовал такому 3-сочетанию, к которому можно добавить тройки, покрывающие b_d, c_d для всех дизъюнктов d одновременно. Возникает небольшая проблема: каждый из элементов вида $a_{j,i}$ для этого должен входить не более, чем в одну тройку вида $(a_{j,i}, b_d, c_d)$. Если какой-то литерал встречается в формуле более двух раз, то так сделать не получится.

3-SAT → 3-SAT с ограничением на число вхождений каждого литерала

Мы решим эту проблему так: по данной 3-КНФ построим 3-КНФ, в которую каждый литерал входит не более двух раз (и которая выполнима тогда и только тогда, когда выполнима исходная), а уже по ней будем строить набор троек.

Заметим сначала, что если какая-то переменная x_i входит в дизъюнкты только в виде литерала x_i (или, наоборот, только в виде литерала $\neg x_i$), то в выполняющем наборе всегда можно заменить её значение на `true` (`false`), и он останется выполняющим. Значит, можно сразу присвоить ей значение `true` (`false`) и упростить формулу.

Далее, пусть всё ещё есть литерал, встречающийся в формуле более двух раз. Тогда соответствующая ему переменная x_i встречается в формуле $k > 3$ раз. Заменим первое её вхождение в формулу на новую переменную $x_{i,1}$, второе — на новую переменную $x_{i,2}$, и так далее. Также добавим новые дизъюнкты, обеспечивающие равенство значений новых переменных:

$$(\neg x_{i,1} \vee x_{i,2}) \wedge (\neg x_{i,2} \vee x_{i,3}) \wedge \cdots \wedge (\neg x_{i,k} \vee x_{i,1}).$$

Для любой новой переменной каждый её литерал входит в формулу не более двух раз. Будем повторять эту операцию, пока встречающиеся больше двух раз литералы не закончатся.

Полученная формула имеет размер, полиномиально зависящий от размера исходной формулы. По ней за полиномиальное время можно вышеописанным способом построить набор троек. При этом исходная формула выполнима тогда и только тогда, когда 3-сочетанием можно покрыть все элементы вида $b_{j,i}, c_{j,i}$ и b_d, c_d.

Последний шаг

Если исходная формула невыполнима, то покрыть все элементы 3-сочетанием точно не удастся. Пусть теперь исходная формула выполнима; рассмотрим 3-сочетание, покрывающее все элементы вида $b_{j,i}, c_{j,i}$ и b_d, c_d. Если n — число переменных, а k — число дизъюнктов в модифицированной формуле, то $2n-k$ элементов вида $a_{j,i}$ остаются непокрытыми. Добавим ещё $2n-k$ пар элементов b'_i, c'_i, а также все тройки вида $(a_{j,i}, b'_i, c'_i)$. Теперь в случае, когда формула выполнима, можно покрыть 3-сочетанием все элементы.

5.9 Задача о трёхдольном сочетании → Уравнение в нулях и единицах

Уравнение в нулях и единицах (zero-one equation) задаётся матрицей $\mathbf{A}$ размера $m \times n$, состоящей из нулей и единиц. Требуется проверить, существует ли такой состоящий из нулей и единиц вектор $\mathbf{x}$, что $\mathbf{Ax} = \mathbf{1}$, где $\mathbf{1}$ — вектор, состоящий из m единиц. Задача лежит в NP: сертификат — искомый вектор $\mathbf{x}$. Построим сведение задачи о трёхдольном сочетании к данной задаче.

Заведём по переменной для каждой тройки: пусть $x_i = 1$, если i-я тройка входит 3-сочетание, $x_i = 0$ иначе. Рассмотрим теперь любой элемент из любой доли. Пусть он входит в тройки с номерами $i_1, \ldots, i_l$. То, что данный элемент покрыт ровно одной тройкой из 3-сочетания, равносильно тому, что $x_{i_1} + \cdots + x_{i_l} = 1$.

Для каждого элемента заведём свою строку в матрице: единицы на ней будут стоять в позициях, соответствующих тройкам, в которые этот элемент входит. Для полученной матрицы $\mathbf{A}$ существует такой вектор $\mathbf{x}$, что $\mathbf{Ax} = \mathbf{1}$, тогда и только тогда, когда существует 3-сочетание, покрывающее все элементы.

5.10 Уравнение в нулях и единицах → Целочисленное линейное программирование

Задача целочисленного линейного программирования (integer linear programming, ILP) в каноническом виде формулируется так: дана целочисленная матрица $\mathbf{A}$ размера $m \times n$, целочисленные вектора $\mathbf{b}$ и $\mathbf{c}$ длины m и n соответственно, а также число k; требуется проверить, существует ли такой целочисленный вектор $\mathbf{x}$ длины n, что

$$\mathbf{x} \geqslant \mathbf{0},\ \mathbf{Ax} \leqslant \mathbf{b},\ \mathbf{c}^{\mathsf{T}}\mathbf{x} \geqslant k.$$

Здесь $\mathbf{0}$ — вектор длины n, состоящий из нулей.

Заметим сначала, что можно так расширить матрицу $\mathbf{A}$ и вектор $\mathbf{b}$, чтобы неравенства $\mathbf{x} \geqslant \mathbf{0}$ и $\mathbf{c}^{\mathsf{T}}\mathbf{x} \geqslant k$ оказались частью неравенства $\mathbf{Ax} \leqslant \mathbf{b}$. Поэтому дальше будем считать, что даны целочисленные матрица $\mathbf{A}$ и вектор $\mathbf{b}$, и надо проверить, существует ли такой целочисленный вектор $\mathbf{x}$, что $\mathbf{Ax} \leqslant \mathbf{b}$ (такая задача не является более общей: для того, чтобы вернуться к формулировке с $\mathbf{x} \geqslant \mathbf{0}$, можно заменить каждую переменную x_i на разность двух неотрицательных, удвоив длину вектора). Задача ILP лежит в NP: сертификат — искомый вектор $\mathbf{x}$.

Задача об уравнении в нулях и единицах является частным случаем задачи ILP: вектор $\mathbf{x}$, состоящий из нулей и единиц и такой, что $\mathbf{Ax} = \mathbf{1}$, существует тогда и только тогда, когда существует целочисленный вектор $\mathbf{x}$ такой, что

$$\mathbf{Ax} \geqslant \mathbf{1},\ \mathbf{Ax} \leqslant \mathbf{1},\ \mathbf{x} \geqslant \mathbf{0},\ \mathbf{x} \leqslant \mathbf{1}.$$

Сведение заключается в выписывании матрицы и вектора, соответствующих этим неравенствам.

5.11 Уравнение в нулях и единицах → Задача о сумме подмножества

В задаче о сумме подмножества (subset sum problem) даны n натуральных чисел w_i; требуется проверить, существует ли подмножество этих чисел, сумма элементов которого равна W. Задача лежит в NP: сертификат — список элементов такого подмножества. Построим сведение задачи об уравнении в нулях и единицах к задаче о сумме подмножества.

Сведение устроено так: интерпретируем каждый столбец данной матрицы $\mathbf{A}$ размера $m \times n$ как число w_i, записанное в системе счисления с основанием $n+1$. Тогда искомый вектор $\mathbf{x}$ соответствует такому подмножеству этих чисел, сумма элементов которого равна $W = \sum_{i=0}^{m-1}(n+1)^i$; переносов в разрядах при сложении не будет благодаря правильному выбору основания.

5.12 Задача о сумме подмножества → Задача о рюкзаке

Задача о сумме подмножества является частным случаем decision-версии задачи о рюкзаке: пусть вес каждого предмета равен его стоимости: $p_i = w_i$. Тогда требуется проверить, существует ли влезающее в рюкзак вместимости W подмножество предметов, имеющее стоимость хотя бы W. Сведение снова заключается в переписывании входных данных в другом формате.

5.13 Уравнение в нулях и единицах → Задача о гамильтоновом цикле

В задаче о гамильтоновом цикле *с парными рёбрами* даны неориентированный граф $G = (V, E)$ и множество пар рёбер $C \subset E \times E$; требуется проверить, существует ли гамильтонов цикл в графе G, которой проходит ровно через одно ребро из каждой пары $(e_1, e_2) \in C$.

Мы построим сведение задачи об уравнении в нулях и единицах к задаче о гамильтоновом цикле с парными рёбрами, а также сведение задачи о гамильтоновом цикле с парными рёбрами к обычной задаче о гамильтоновом цикле.

Уравнение в нулях и единицах → Задача о гамильтоновом цикле с парными рёбрами

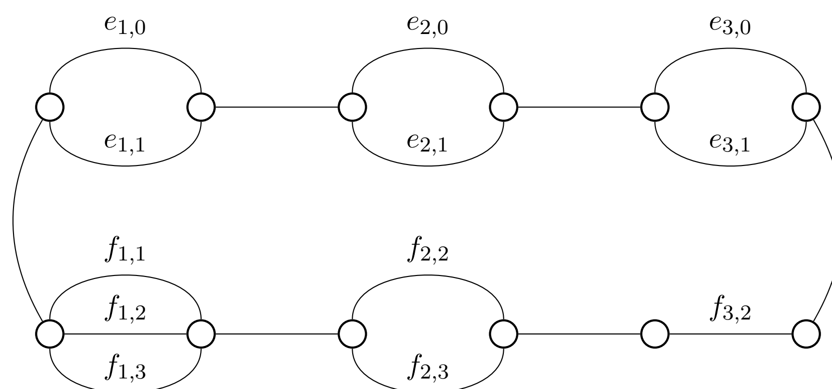


Рис. 5.3: Цикл для уравнения $\begin{pmatrix}1&1&1\\0&1&1\\0&1&0\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} = \begin{pmatrix}1\\1\\1\end{pmatrix}$

По данному уравнению $\mathbf{Ax} = \mathbf{1}$ ($\mathbf{A}$ имеет размер $m \times n$) построим цикл длины $2(m+n)$, в котором будет $m+n$ семейств кратных рёбер, чередующихся с некратными:

каждой из n переменных x_i будет соответствовать два кратных ребра $e_{i,0}, e_{i,1}$ (взятие $e_{i,0}$ в гамильтонов цикл будет соответствовать случаю $x_i = 0$, взятие $e_{i,1}$ — случаю $x_i = 1$); i-й строке матрицы (которой соответствует уравнение вида $x_{j_1} + \cdots + x_{j_k} = 1$) будет соответствовать k кратных рёбер $f_{i,j_1}, \ldots, f_{i,j_k}$ (взятие в гамильтонов цикл ребра f_{i,j_q} будет соответствовать тому, что $x_{j_q} = 1$, а остальные переменные, входящие в i-е уравнение, равны нулю).

Множество C будет отвечать за то, чтобы соответствия между значениями переменных и тем, какие рёбра взяты в цикл, были согласованы друг с другом. C будет состоять из всех пар рёбер вида $(e_{j,0}, f_{i,j})$: теперь если $f_{i,j}$ взято в гамильтонов цикл, то в него взято и $e_{j,1}$; и наоборот, если $e_{j,1}$ взято в цикл, то и все рёбра вида $f_{i,j}$ тоже в него взяты.

Решения уравнения $\mathbf{Ax} = \mathbf{1}$ находятся во взаимно-однозначном соответствии с гамильтоновыми циклами в построенном графе, проходящими ровно через одно ребро из каждой пары рёбер из C.

Задача о гамильтоновом цикле с парными рёбрами → Задача о гамильтоновом цикле

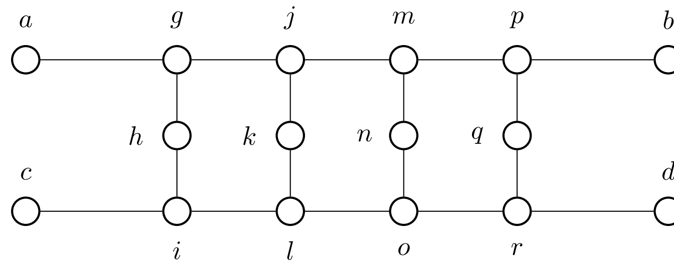


Рис. 5.4: Конструкция, которой мы будем заменять парные рёбра

Будем постепенно уменьшать количество пар рёбер в C, перестраивая граф, пока C не станет пустым. Для этого воспользуемся конструкцией с рис. 5.4: если никакие вершины, кроме a, b, c, d, не связаны с другими вершинами в графе, то есть лишь два способа, которыми через эту конструкцию может проходить гамильтонов путь:

$$a - g - h - i - l - k - j - m - n - o - r - q - p - b$$

или

$$c - i - h - g - j - k - l - o - n - m - p - q - r - d.$$

Возьмём любую пару рёбер $((a, b), (c, d))$ из C и заменим её на такую конструкцию (добавив в граф новые вершины $g, \ldots, r$). При этом любой гамильтонов цикл, проходивший ровно через одно из рёбер (a, b), (c, d), соответствует гамильтонову циклу в новом графе, проходящему через конструкцию первым или вторым способом соответственно (и это соответствие — биекция).

Важно не забыть, что (a, b) и (c, d) могли входить и в другие пары в C. Заменим в каждой такой паре (a, b) на (a, g), а (c, d) на (c, i) (так как гамильтонов цикл, проходивший через (a, b) или (c, d), теперь проходит через (a, g) или (c, i) соответственно).

Будем повторять эту операцию, пока множество C не опустеет. На каждом шаге мы добавляем в граф константное число вершин и рёбер, поэтому весь процесс займёт полиномиальное время.

5.14 Задача о гамильтоновом цикле → Задача о гамильтоновом пути

В задаче о гамильтоновом пути требуется проверить, есть ли в данном неориентированном графе гамильтонов путь из s в t, то есть путь из s в t, проходящий через каждую вершину ровно один раз. Построим сведение задачи о гамильтоновом цикле к задаче о гамильтоновом пути.

Возьмём любую вершину графа v и добавим в граф её копию v': новую вершину, которая будет смежна с теми же вершинами, что и v. Кроме того, добавим в граф две новых вершины s и t, а также рёбра (s, v) и (t, v'). В полученном графе есть гамильтонов путь из s в t тогда и только тогда, когда в исходном графе был гамильтонов цикл.

Задача о гамильтоновом пути → Задача о гамильтоновом цикле

Сведение в обратную сторону устроено ещё проще: нужно добавить в граф новую вершину u и рёбра (s, u) и (t, u).

5.15 Задача о гамильтоновом цикле → Задача коммивояжёра

В decision-версии задачи коммивояжёра требуется проверить, существует ли в данном графе гамильтонов цикл длины не более k. Задача лежит в NP: сертификат — список вершин в искомом гамильтоновом цикле. Построим сведение задачи о гамильтоновом цикле к задаче коммивояжёра.

По данному графу $G = (V, E)$ построим взвешенный полный граф на множестве вершин V: пусть $d(u, v) = 1$, если $(u, v) \in E$; иначе пусть $d(u, v) = 1 + L$, $L > 0$. Если в исходном графе был гамильтонов цикл, то в новом графе есть маршрут коммивояжёра длины $k = |V|$; в противном случае любой маршрут проходит хотя бы по одному ребру веса $1 + L$, то есть имеет длину хотя бы $|V| + L > k$.

5.16 Любая задача из NP → SAT

Теорема 5.16.1 — Теорема Кука-Левина (Cook, 1971; Levin, 1973). Задача SAT является NP-полной.

Строго говоря, мы не докажем эту теорему (для этого требуется формально определить, что такое алгоритм, чего мы не делали). Мы, однако, обсудим идею доказательства. Первый шаг — свести любую задачу из NP к задаче CIRCUIT-SAT, чуть более общей, чем SAT. Второй шаг — свести CIRCUIT-SAT к SAT.

Булева схема (boolean circuit) — ациклический ориентированный граф, в котором каждая вершина относится к одному из следующих типов:

- *гейт (gate)* "И" или "ИЛИ", имеющий входящую степень два;
- гейт отрицания, имеющий входящую степень один;
- константа — вершина входящей степени ноль, помеченная значением `true` или `false`;
- *вход (input)* — вершина входящей степени ноль, помеченная переменной.

Кроме того, одна из вершин является *выходом (output)* и имеет исходящую степень ноль.

Если задать значения всех переменных, то можно определить значение, соотвествующее каждой вершине (значение входа — значение соответствующей переменной, значение константы — эта константа, значение гейта — результат применения соответствующей операции к значениям вершин, из которых в гейт входят рёбра). Значение схемы — это значение вершины-выхода.

В задаче выполнимости булевой схемы (circuit satisfiability problem, CIRCUIT-SAT) нужно по данной булевой схеме выяснить, выполнима ли она, то есть существует ли набор значений переменных, при котором значение схемы истинно.

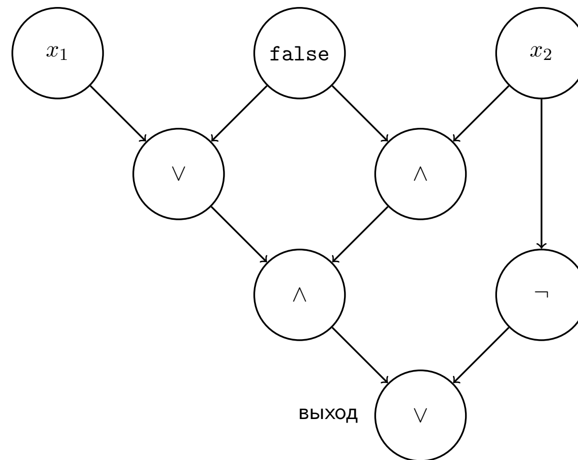


Рис. 5.5: Пример булевой схемы

Любая задача из NP → CIRCUIT-SAT

Как мы помним, для любой задачи L из NP существует алгоритм A, который принимает вход задачи x и сертификат y, $|y| \leqslant q(|x|)$, и работает за время $O(p(|x|))$. При этом алгоритм возвращает "да" для некоторого значения сертификата, если $x \in L$, и возвращает "нет" для любого значения сертификата, если $x \notin L$.

Алгоритм A исполняется компьютером, а компьютер фактически представляет собой булеву схему: она состоит из множества уровней, каждый уровень состоит из бит, хранящихся в памяти компьютера в определённый момент времени, при этом значения вершин одного уровня вычисляются булевыми операциями по значениям вершин предыдущего уровня (это, конечно, не строгое рассуждение).

Поскольку алгоритм A работает за полиномиальное от длины входа время, соответствующая ему булева схема имеет полиномиальный от длины входа размер. Итак, наше сведение выглядит следующим образом: по данному входу x мы строим булеву схему, соответствующую алгоритму A, при этом значения бит x соответствуют вершинам-константам (поскольку эти значения нам известны), а значения бит сертификата y — вершинам-входам. Полученная схема выполнима тогда и только тогда, когда существует значение сертификата, на котором алгоритм A вернёт "да".

CIRCUIT-SAT → SAT

Построим по данной булевой схеме КНФ следующим образом: введём новую переменную для каждого гейта и каждой вершины-константы (то есть теперь каждой вершине графа будет соответствовать своя переменная). При этом добавим в КНФ следующие дизъюнкты (далее x — переменная, соответствующая рассматриваемой вершине):

- для вершины-константы со значением `true` добавим дизъюнкт (x);
- для вершины-константы со значением `false` добавим дизъюнкт $(\neg x)$;
- для гейта отрицания, в который входит ребро из вершины, соответствующей переменной y, добавим дизъюнкты $(x \vee y)$ и $(\neg x \vee \neg y)$;
- для гейта "ИЛИ", в который входят рёбра из вершин, соответствующих переменным y, z, добавим дизъюнкты $(x \vee \neg y)$, $(x \vee \neg z)$ и $(\neg x \vee y \vee z)$;

- для гейта "И", в который входят рёбра из вершин, соответствующих переменным y, z, добавим дизъюнкты $(\neg x \vee y)$, $(\neg x \vee z)$ и $(x \vee \neg y \vee \neg z)$.

Наконец, добавим в КНФ дизъюнкт (x), где x — переменная, соответствующая вершине-выходу. Полученная КНФ выполнима тогда и только тогда, когда выполнима данная булева схема.

IV Приближённые алгоритмы и алгоритмы кэширования

6 Приближённые алгоритмы 49
6.1 Задача о покрытии множества
6.2 Задача о вершинном покрытии
6.3 Кластеризация
6.4 Задача коммивояжёра
6.5 Задача о рюкзаке
6.6 MAX-E3-SAT

7 Алгоритмы кэширования 55
7.1 LRU и другие алгоритмы маркировки
7.2 Рандомизированный алгоритм маркировки

6. Приближённые алгоритмы

На практике часто встречаются задачи, которые не умеют решать за полиномиальное время. Тем не менее, с такими зачадами всё равно нужно как-то справляться. Иногда достаточно решить задачу *приближённо*: найти решение, которое не сильно хуже оптимального.

Пусть мы решаем задачу максимизации или минимизации некоторой величины. Обозначим за $opt(I)$ оптимальное значение этой величины на входе I. Пусть A — алгоритм, который по входу I выдаёт решение со значением величины $A(I)$. Будем считать, что $opt(I)$, $A(I)$ — положительные числа (как правило, даже если это не так, задачу несложно переформулировать так, чтобы это стало верным). Алгоритм A имеет *ошибку приближения (approximation ratio)* ρ_A, если

$$\max_I \max\left(\frac{A(I)}{opt(I)}, \frac{opt(I)}{A(I)}\right) \leqslant \rho_A,$$

где максимум берётся по всем возможным входам I. Иногда удобно рассматривать ошибку приближения, зависящую от каких-то параметров входных данных: например, пусть входные данные представляют собой граф, n — число вершин в графе, тогда в определении $\rho_A(n)$ максимум берётся по всем графам на не более чем n вершинах.

Определение устроено так, что $\rho_A \geqslant 1$ и для задач максимизации, и для задач минимизации (в каждом случае в определении можно оставить только одну из дробей). Алгоритм A в вышеописанной ситуации называют *ρ_A-приближённым (ρ_A-approximation algorithm)*.

6.1 Задача о покрытии множества

В оптимизационной версии задачи о покрытии множества даны множество U, $|U| = n$, и набор его подмножеств $S_1, \ldots, S_m$, объединение которых совпадает с U: $\bigcup_{i=1}^m S_i = U$. Необходимо выбрать минимальное количество подмножеств $S_{i_1}, \ldots, S_{i_k}$ так, чтобы они покрывали все элементы U: $\bigcup_{j=1}^k S_{i_j} = U$.

Будем искать приближённое решение следующим жадным алгоритмом: на каждом шаге будем брать множество S_i с максимальным числом ещё не покрытых элементов. Такой алгоритм легко реализовать за полиномиальное время (можно даже за $O(n + \sum_i |S_i|)$). Покажем, что этот алгоритм является $\lceil \ln n \rceil$-приближённым.

Пусть оптимальное покрытие состоит из k множеств. Пусть после t шагов алгоритма непокрыто n_t элементов U. Найдётся множество, покрывающее хотя бы $\frac{n_t}{k}$ из этих элементов (существуют k множеств, покрывающие все n элементов, то есть и эти n_t тоже). Значит,

$$n_{t+1} \leqslant n_t - \frac{n_t}{k} = n_t\left(1 - \frac{1}{k}\right),$$

откуда

$$n_t \leqslant n_0\left(1 - \frac{1}{k}\right)^t \leqslant n\left(1 - \frac{1}{k}\right)^t.$$

Воспользуемся неравенством $1 - x \leqslant e^{-x}$, которое является строгим при всех $x \neq 0$ (производная выражения слева равна -1, производная выражения справа равна $-e^{-x}$, то есть меньше -1 при $x < 0$, и больше -1 при $x > 0$), и получим

$$n_t < n\left(e^{-\frac{1}{k}}\right)^t = ne^{-\frac{t}{k}}.$$

Тогда

$$n_{k\lceil\ln n\rceil} < ne^{-\frac{k\lceil\ln n\rceil}{k}} \leqslant ne^{-\frac{k\ln n}{k}} = 1,$$

то есть $n_{k\lceil\ln n\rceil} = 0$. Значит, алгоритм найдёт покрытие размера не более $k\lceil\ln n\rceil$.

6.2 Задача о вершинном покрытии

В оптимизационной версии задачи о вершинном покрытии требуется найти в неориентированном графе $G = (V, E)$ минимальное по размеру вершинное покрытие.

Эта задача является частным случаем задачи о покрытии множества ($U = E$, подмножества — множества рёбер, инцидентных одной вершине). Значит, жадный алгоритм, который на каждом шаге выбирает вершину максимальной степени и удаляет её из графа, является $\lceil\ln n\rceil$-приближённым (если $|V| = n$). Существует, однако, и полиномиальный алгоритм с меньшей ошибкой приближения.

Рассмотрим любое максимальное по включению *паросочетание (matching)* M (множество попарно несмежных рёбер). Такое паросочетание легко найти в произвольном графе следующим жадным алгоритмом: будем рассматривать рёбра в некотором порядке и добавлять ребро к паросочетанию, если оно не имеет общих концов с уже добавленными рёбрами.

Пусть S — вершинное покрытие минимального размера. Заметим, что $|S| \geqslant |M|$, поскольку хотя бы один из концов каждого ребра паросочетания M лежит в S. С другой стороны, множество S' всех концов рёбер из M является вершинным покрытием: если бы нашлось ребро e, неинцидентное ни одной из вершин из S', e можно было бы добавить к паросочетанию M. При этом $|S'| \leqslant 2 \cdot |M| \leqslant 2 \cdot |S|$. Получаем 2-приближённый полиномиальный алгоритм поиска минимального по размеру вершинного покрытия.

6.3 Кластеризация

Задача k-кластеризации звучит следующим образом: есть n точек $x_1, \ldots, x_n$, а также метрика на этих точках, то есть такая функция $d(\cdot, \cdot)$, что:

- $d(x, y) \geqslant 0$ для любых x, y;
- $d(x, y) = 0$ тогда и только тогда, когда $x = y$;
- $d(x, y) = d(y, x)$ для любых x, y;
- $d(x, z) \leqslant d(x, y) + d(y, z)$ для любых x, y, z.

Необходимо разбить эти точки на k множеств (*кластеров*) $C_1, \ldots, C_k$ так, чтобы максимальный диаметр кластера

$$\max_{i=1}^{k} \max_{x,y \in C_i} d(x, y)$$

был как можно меньше.

Построим приближённый алгоритм, воспользовавшись следующей идеей: назначим некоторые k точек *центрами кластеров*; каждую из оставшихся $n - k$ точек отнесём к тому же кластеру, что и ближайший к этой точке центр. Центры будем выбирать так: в качестве первого центра возьмём произвольную точку; каждым следующим центром будем выбирать точку, максимально удалённую от уже выбранных центров.

Почему получившееся разбиение на кластеры не сильно хуже оптимального? Пусть центры кластеров — это точки $c_1, \ldots, c_k$; y — самая удалённая от центров кластеров точка из оставшихся $n-k$. Если расстояние от y до ближайшего к ней центра равно r, то диаметр каждого кластера не превосходит $2r$: если точки a, b попали в кластер с центром c_i, то по неравенству треугольника $d(a,b) \leqslant d(a,c_i) + d(c_i,b) \leqslant r + r = 2r$.

С другой стороны, точки $c_1, \ldots, c_k, y$ попарно удалены друг от друга на расстояние хотя бы r. Значит, разбиения на k кластеров с максимальным диаметром меньше, чем r, не существует (в таком разбиении эти $k+1$ точек все попали бы в разные кластеры). Таким образом, вышеописанный полиномиальный алгоритм является 2-приближённым.

6.4 Задача коммивояжёра

Вспомним задачу коммивояжёра: дан полный неориентированный взвешенный граф; расстояние между вершинами u и v обозначим за $d(u,v)$. Необходимо найти гамильтонов цикл, имеющий минимально возможную длину.

Предположим, что функция расстояний между вершинами удовлетворяет неравенству треугольника: $d(x,z) \leqslant d(x,y) + d(y,z)$ для любых x, y, z. Такое предположение выполняется в некоторых реальных приложениях: например, если вершины графа расположены на плоскости, и длина ребра между вершинами совпадает с длиной соответствующего отрезка на плоскости. Существуют полиномиальные приближённые алгоритмы с ошибкой приближения 2, и даже 1.5, на графах, расстояние между вершинами в которых удовлетворяет неравенству треугольника.

2-приближённый алгоритм

Длину цикла C будем обозначать как $l(C)$; для подграфа H под $l(H)$ будем иметь в виду суммарный вес всех рёбер в этом подграфе.

Пусть C — оптимальный маршрут коммивояжёра, то есть гамильтонов цикл минимально возможной длины. Заметим, что C без любого ребра представляет собой остовное дерево. Значит, если T — минимальное остовное дерево графа, то $l(C) \geqslant l(T)$.

Мы умеем строить дерево T за полиномиальное время. Обойдём построенное дерево (например, в порядке поиска в глубину), пройдя по каждому ребру два раза. Получился цикл длины $2 \cdot l(T)$, проходящий через все вершины. Проблема в том, что он проходит по некоторым вершинам больше одного раза. Решим эту проблему следующим образом: просто удалим из этого цикла повторные вхождения каждой вершины, то есть из очередной вершины пойдём напрямую в следующую вершину на цикле, которую ещё не посещали. Другими словами, рассмотрим гамильтонов цикл C', вершины в котором идут в порядке увеличения времён входа при обходе дерева T алгоритмом поиска в глубину.

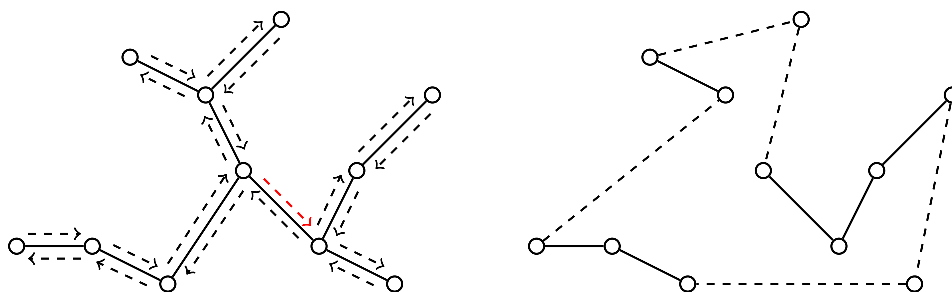


Рис. 6.1: Обход минимального остовного дерева T (начиная с красного ребра) и построенный по нему гамильтонов цикл C'.

При удалении из цикла повторных вхождений вершин мы для некоторых пар вершин a, b заменили какой-то путь из a в b на прямое ребро из a в b. По неравенству треугольника такие замены не могли увеличить длину цикла. Значит, $l(C') \leqslant 2 \cdot l(T) \leqslant 2 \cdot l(C)$; описанный алгоритм является 2-приближённым.

1.5-приближённый алгоритм (Christofides, 1976)

Пусть V_{odd} — множество вершин, имеющих нечётную степень в дереве T; $|V_{odd}| \geqslant 2$, $|V_{odd}|$ чётно. Пусть M — паросочетание минимального веса на вершинах V_{odd}; H — объединение T и M. В H все вершины имеют чётную степень, значит, H — эйлеров подграф; найдётся эйлеров цикл X, проходящий по каждому ребру H ровно один раз: $l(X) = l(H) = l(T) + l(M)$. Удалив из цикла X повторные вхождения вершин, мы получим гамильтонов цикл X', при этом по неравенству треугольника $l(X') \leqslant l(X)$.

Мы знаем, что $l(T) \leqslant l(C)$; остаётся оценить $l(M)$. Пусть C_{odd} — цикл, полученный удалением из цикла C всех вершин, не лежащих в V_{odd}. По неравенству треугольника, $l(C_{odd}) \leqslant l(C)$. При этом C_{odd} — цикл чётной длины, поэтому он представляет собой объединение двух паросочетаний на вершинах V_{odd}; обозначим их за M_1, M_2, тогда $l(M_1) + l(M_2) = l(C_{odd})$. M — паросочетание минимального веса на вершинах W, поэтому

$$l(M) \leqslant \min(l(M_1), l(M_2)) \leqslant \frac{l(C_{odd})}{2} \leqslant \frac{l(C)}{2}.$$

Итак, мы можем построить такой гамильтонов цикл X', что

$$l(X') \leqslant l(T) + l(M) \leqslant l(C) + \frac{l(C)}{2} = \frac{3l(C)}{2}.$$

Паросочетание минимального веса в произвольном графе можно искать за полиномиальное время, например, алгоритмом Эдмондса (Blossom algorithm, Edmonds, 1965) за $O(V^2E)$; известны и алгоритмы с лучшим временем работы.

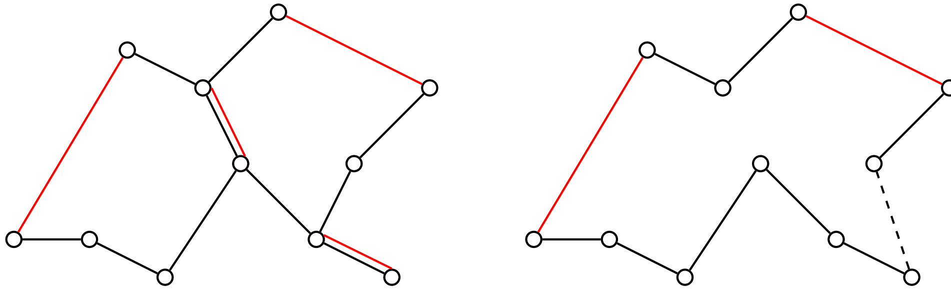


Рис. 6.2: Дерево T и паросочетание M (отмечено красным цветом), а также цикл X' (один из вариантов).

Общий случай

В общем случае, когда неравенство треугольника может не выполняться, задача поиска приближённого решения оказывается намного более сложной. Покажем, что если $\mathrm{P} \neq \mathrm{NP}$, то ни для какого фиксированного ρ не существует полиномиального ρ-приближённого алгоритма решения задачи коммивояжёра в общем случае. Для этого вспомним, как задача коммивояжёра связана с задачей поиска гамильтонова цикла.

Пусть есть граф $G = (V, E)$, и необходимо проверить, есть ли в этом графе гамильтонов цикл. Как мы уже знаем, эта задача является NP-полной. Построим по графу G полный граф H на том же множестве вершин со следующими весами рёбер: $d(u, v) = 1$, если

$(u, v) \in E$, иначе $d(u, v) = 1 + L$; L выберем чуть позже. Пусть C — гамильтонов цикл минимального веса в H. Если в G есть гамильтонов цикл, то $l(C) = |V|$, иначе $l(C) \geqslant |V| + L$.

Если бы существовал полиномиальный ρ-приближённый алгоритм решения задачи коммивояжёра, он нашёл бы в H такой гамильтонов цикл C', что $l(C') \leqslant \rho \cdot l(C)$. Таким образом, выбрав изначально $L = \rho \cdot |V|$, мы смогли бы за полиномиальное время проверить, есть ли в исходном графе G гамильтонов цикл: если цикла нет, то $l(C') \geqslant l(C) \geqslant |V| + L = (\rho + 1) \cdot |V|$; если цикл есть, то $l(C') \leqslant \rho \cdot |V|$.

6.5 Задача о рюкзаке

Вспомним формулировку задачи о рюкзаке: есть n предметов, i-й предмет имеет вес w_i и стоимость p_i; нужно выбрать предметы суммарного веса не более W с максимально возможной суммарной стоимостью. Мы построим алгоритм, который по любому $\varepsilon > 0$ найдёт решение этой задачи с ошибкой приближения $1 + \varepsilon$ за время, полиномиально зависящее от размера входа и от $1/\varepsilon$; такие алгоритмы называют *fully polynomial-time approximation schemes (FPTAS)*.

Задачу о рюкзаке мы умеем решать методом динамического программирования за $O(nW)$: $dp[i, w]$ — максимально возможная суммарная стоимость подмножества предметов с номерами не более i, имеющих суммарный вес w. Похожим образом её можно решить за $O(nP)$, где $P = \sum_{i=1}^{n} p_i$: $dp[i, p]$ — минимально возможный суммарный вес подмножества предметов с номерами не более i, имеющих суммарную стоимость p.

Приближённый алгоритм будет решать задачу последним методом, но используя округлённые стоимости p'_i вместо обычных: пусть

$$p'_i = \left\lfloor \frac{2np_i}{\varepsilon p_{max}} \right\rfloor, \text{ где } p_{max} = \max_{i=1}^{n} p_i,$$

тогда, поскольку $p'_i \leqslant 2n/\varepsilon$, найти оптимальный набор предметов с округлёнными стоимостями можно за $O(n^3/\varepsilon)$ — время, полиномиально зависящее от n и $1/\varepsilon$. Остаётся понять, почему найденное решение не сильно хуже оптимального.

Пусть S' — множество предметов, найденное вышеописанным алгоритмом, S — оптимальное множество предметов. Обозначим $P(S') = \sum_{i \in S'} p_i$, $P(S) = \sum_{i \in S} p_i$. Можно считать, что $w_i \leqslant W$ для любого i: сразу выкинем из рассмотрения предметы, для которых это не так. Тогда $P(S) \geqslant p_{max}$, так как самый дорогой предмет всегда можно взять сам по себе. Нужно доказать, что $P(S) \leqslant (1 + \varepsilon) \cdot P(S')$.

Мы знаем, что множество S' оптимально при использовании округлённых стоимостей, значит,

$$\sum_{i \in S'} p'_i \geqslant \sum_{i \in S} p'_i = \sum_{i \in S} \left\lfloor \frac{2np_i}{\varepsilon p_{max}} \right\rfloor \geqslant \sum_{i \in S} \left(\frac{2np_i}{\varepsilon p_{max}} - 1 \right) \geqslant P(S) \cdot \frac{2n}{\varepsilon p_{max}} - n.$$

Тогда

$$P(S') \geqslant \frac{\varepsilon p_{max}}{2n} \cdot \sum_{i \in S'} p'_i \geqslant P(S) - \frac{\varepsilon p_{max}}{2} \geqslant P(S) \cdot \left(1 - \frac{\varepsilon}{2}\right);$$

последнее неравенство следует из того, что $P(S) \geqslant p_{max}$. Наконец, поскольку неравенство $(1 - \varepsilon/2)(1 + \varepsilon) > 1$ выполняется при $\varepsilon < 1$, мы получаем

$$P(S) \leqslant \frac{1}{1 - \varepsilon/2} \cdot P(S') \leqslant (1 + \varepsilon) \cdot P(S').$$

6.6 MAX-E3-SAT

Иногда приближённые решения проще искать рандомизированными алгоритмами. Пусть мы решаем задачу максимизации или минимизации некоторой величины, $opt(I)$ — оптимальное значение величины на входе I. Пусть A — рандомизированный алгоритм. На входе I он может выдавать разные решения с разным значением величины; математическое ожидание значения величины при запуске алгоритма на входе I обозначим за $A(I)$. Дальше определение совпадает с определением обычного приближённого алгоритма: A — *рандомизированный ρ_A-приближённый*, если

$$\max_I \max\left(\frac{A(I)}{opt(I)}, \frac{opt(I)}{A(I)}\right) \leqslant \rho_A.$$

Задача MAX-E3-SAT состоит в том, чтобы по 3-КНФ, каждый дизъюнкт которой состоит **ровно** из трёх литералов, найти набор значений переменных, выполняющий как можно больше дизъюнктов этой 3-КНФ. Мы рассмотрим очень простой рандомизированный алгоритм: каждой переменной он случайно присваивает значение 0 или 1 с вероятностью $\frac{1}{2}$ каждое.

Какова вероятность того, что получившийся набор переменных выполняет какой-то конкретный дизъюнкт? Дизюнкт не выполнится, только если значения всех трёх его литералов ложны, вероятность чего равна $\frac{1}{8}$. Значит, каждый дизъюнкт выполнится с вероятностью $\frac{7}{8}$.

Пусть 3-КНФ состоит из k дизъюнктов, $X = X_1 + \cdots + X_k$ — случайная величина, равная числу выполненных дизъюнктов, X_i равно 1, если i-й дизъюнкт выполнен, и 0 иначе. Из сказанного выше следует, что $\mathbb{E}X_i = \frac{7}{8}$ для любого i. Тогда $\mathbb{E}X = \frac{7}{8}k$. При этом никакой алгоритм не сможет выполнить больше k дизъюнктов, значит, описанный алгоритм является рандомизированным $\frac{8}{7}$-приближённым.

R Из рассуждений выше следует, что для любой 3-КНФ, состоящей из k дизъюнктов, по три литерала в каждом, найдётся набор значений переменных, выполняющий хотя бы $\frac{7}{8}k$ дизъюнктов.

7. Алгоритмы кэширования

7.1 LRU и другие алгоритмы маркировки

Вернёмся к задаче кэширования. Алгоритм Белади решает задачу оптимально, но он неприменим на практике, поскольку при принятии решений использует информацию о будущих запросах. Обозначим количество кэш-промахов алгоритма Белади на последовательности запросов σ за $opt(\sigma)$. Мы будем оценивать эффективность алгоритмов управления кэшом, сравнивая количество совершаемых ими кэш-промахов на последовательности запросов σ с $opt(\sigma)$.

Назовём алгоритм управления кэшом *алгоритмом маркировки (marking algorithm)*, если он устроен следующим образом: время работы алгоритма разбивается на несколько *фаз*; в начале каждой фазы все фрагменты памяти *не помечены*. При запросе элемента x этот элемент *помечается*. При этом, если x не лежит в кэше, то он помещается в кэш на место одного из непомеченных элементов (если такие есть); какого именно из непомеченных — зависит от конкретного алгоритма. Если все лежащие в кэше элементы помечены, то текущая фаза заканчивается, все элементы становятся непомеченными, а запрос x обрабатывается уже в следующей фазе алгоритма.

Заметим, что в любой момент времени выполнения такого алгоритма любой помеченный элемент был в последний раз запрошен позже, чем любой непомеченный. Это так, поскольку помеченные элементы — это ровно элементы, запрошенные на текущей фазе алгоритма.

Алгоритмом *LRU (least recently used)* будем называть алгоритм, который удаляет из кэша элемент, который не запрашивался дольше всех. Если отсчитывать фазы и отмечать элементы, как описано выше, такой элемент в момент удаления всегда будет непомеченным. Значит, LRU является алгоритмом маркировки.

Теорема 7.1.1 Любой алгоритм маркировки на любой последовательности запросов σ совершает не более $k \cdot opt(\sigma) + k$ кэш-промахов, где k — размер кэш-памяти.

Доказательство. Пусть обработка последовательности σ алгоритмом маркировки состояла из t фаз. Заметим, что на каждой фазе было запрошено не более k различных элементов, так как каждый запрошенный элемент помечался, попадал в кэш, если не был там ранее, и не удалялся из кэша до конца фазы. Кэш-промах мог пройзойти только при первом в течение фазы запросе каждого из этих элементов. Значит, на каждой фазе алгоритм маркировки сделал не более k кэш-промахов, то есть общее количество сделанных им кэш-промахов не превышает kt.

Теперь оценим количество кэш-промахов, сделанных алгоритмом Белади. Пусть i-я фаза алгоритма маркировки соответствует подотрезку запросов $\sigma_{l(i)}, \ldots, \sigma_{r(i)}$.

Рассмотрим последовательность запросов с первого в i-й фазе по первый в $(i+1)$-й фазе включительно, то есть с $l(i)$-го по $l(i+1)$-й. В этой последовательности есть $k+1$ различный элемент (все помеченные в течение i-й фазы, а также $\sigma_{l(i+1)}$). Значит, во время обработки запросов с $(l(i)+1)$-го по $l(i+1)$-ый алгоритм Белади сделает хотя бы один кэш-промах. Тогда за время обработки всей последовательности σ алгоритм Белади

сделает хотя бы $t-1$ кэш-промах, то есть $opt(\sigma) \geqslant t-1$.

Итак, алгоритм маркировки сделает не более

$$kt \leqslant k \cdot (opt(\sigma)+1) = k \cdot opt(\sigma) + k$$

кэш-промахов. ■

7.2 Рандомизированный алгоритм маркировки

Рандомизированный алгоритм маркировки удаляет из кэша каждый из непомеченных элементов равновероятно.

Теорема 7.2.1 Математическое ожидание количества кэш-промахов, сделанных рандомизированным алгоритмом маркировки на последовательности запросов σ, не превосходит $O(\log k) \cdot opt(\sigma)$, где k — размер кэш-памяти.

Доказательство. Количество фаз снова обозначим за t. На i-й фазе будем называть непомеченный элемент *свежим*, если он не запрашивался на $(i-1)$-й фазе, и *несвежим*, если запрашивался; непомеченный элемент на первой фазе будем называть свежим, если он не лежал в кэше в момент начала работы алгоритма, и несвежим, если лежал. Количество запросов свежих элементов на i-й фазе обозначим за c_i. Тогда на i-й фазе было запрошено $k-c_i$ несвежих элементов (не более $k-c_i$ при $i=t$).

При запросе каждого свежего элемента на любой фазе точно произойдёт кэш-промах. Пусть на i-й фазе произошёл запрос несвежего элемента s, при этом до этого на этой фазе уже были запрошены $x < k-c_i$ несвежих элементов и $y \leqslant c_i$ свежих элементов. В начале фазы в кэше лежали k несвежих элементов (s — один из них). С тех пор x из них перестали быть несвежими (и стали помеченными), а из оставшихся $k-x$ в кэше осталось $k-x-y$, причём для каждого из $k-x$ элементов вероятность того, что он остался, одинаковая. Значит, вероятность того, что s был удалён из кэша, равна

$$\frac{y}{k-x} \leqslant \frac{c_i}{k-x}.$$

Тогда математическое ожидание количества кэш-промахов на i-й фазе не превосходит

$$c_i + \sum_{x=0}^{k-c_i-1} \frac{c_i}{k-x} \leqslant c_i \cdot \left(1 + \sum_{x=c_i+1}^{k} \frac{1}{x}\right) \leqslant c_i \cdot \sum_{x=1}^{k} \frac{1}{x}.$$

Снова обозначим границы i-й фазы в последовательности запросов за $l(i)$, $r(i)$. Количество кэш-промахов алгоритма Белади при выполнении запросов с $l(i)$-го по $r(i)$-й обозначим за n_i. Алгоритм Белади совершит кэш-промах на каждом свежем элементе первой фазы, то есть $n_1 \geqslant c_1$. Для $i > 1$ среди запросов с $l(i-1)$-го по $r(i)$-й найдётся $k + c_i$ различных, поэтому $n_{i-1} + n_i \geqslant c_i$. Получаем

$$2 \cdot opt(\sigma) = 2\sum_{i=1}^{t} n_i \geqslant n_1 + \sum_{i=2}^{t}(n_{i-1}+n_i) \geqslant \sum_{i=1}^{t} c_i.$$

Тогда математическое ожидание количества кэш-промахов, сделанных рандомизированным алгоритмом маркировки, не превосходит

$$\left(\sum_{i=1}^{t} c_i\right) \cdot \left(\sum_{x=1}^{k} \frac{1}{x}\right) \leqslant 2 \cdot \left(\sum_{x=1}^{k} \frac{1}{x}\right) \cdot opt(\sigma) = O(\log k) \cdot opt(\sigma).$$

■

V Продвинутые структуры данных

8 Дерево отрезков 58
8.1 Построение
8.2 Реализация операций “снизу”
8.3 Реализация операций “сверху”
8.4 Область применения дерева отрезков
8.5 Изменение элементов на подотрезке
8.6 Динамическое дерево отрезков
8.7 Персистентное дерево отрезков
8.8 Две модельных задачи
8.9 Дерево отрезков сортированных массивов
8.10 Fractional cascading
8.11 Многомерное дерево отрезков
8.12 Метод сканирующей прямой
8.13 k-я порядковая статистика на отрезке

9 AVL-дерево 74
9.1 Двоичное дерево поиска
9.2 Базовые операции
9.3 AVL-дерево
9.4 Балансировка
9.5 Реализация

10 Декартово дерево 80
10.1 Декартово дерево
10.2 Операции Split и Merge
10.3 Остальные операции
10.4 Построение
10.5 Случайное дерево поиска
10.6 Дополнительные операции

11 Краткое отступление про B-деревья 88
11.1 B-дерево
11.2 2-3-дерево
11.3 2-3-4-дерево и RB-дерево

12 Splay-Дерево 90
12.1 Операция Splay
12.2 Остальные операции
12.3 Реализация
12.4 Оценка времени работы

13 Skip-list 96
13.1 Основная идея
13.2 Реализация
13.3 Оценка времени работы

14 RMQ и LCA 100
14.1 Разреженная таблица
14.2 Блочная оптимизация
14.3 Конструкция Фишера-Хойна
14.4 LCA
14.5 Алгоритм Тарьяна

8. Дерево отрезков

Пусть дан массив из n элементов $a_0, \ldots, a_{n-1}$, и требуется уметь выполнять следующие операции:

- **запрос на отрезке:** даны $0 \leqslant l < r \leqslant n$; нужно найти сумму элементов на подотрезке массива $a[l, r)$: $a_l + \cdots + a_{r-1}$;
- **изменение значения в точке:** даны $0 \leqslant pos < n$, x; нужно изменить значение элемента a_{pos} на x.

Дерево отрезков позволяет выполнять каждую из этих операций за $O(\log n)$.

8.1 Построение

Сначала добавим в конец массива несколько нулей так, чтобы его длина стала степенью двойки. При этом длина массива увеличится не более, чем в два раза, поэтому все оценки на используемую память и время работы верны и для исходной длины массива.

Теперь построим полное двоичное дерево, на нижнем уровне которого будет $n = 2^k$ вершин. В эти вершины запишем элементы массива. В каждую из вершин дерева на других уровнях запишем сумму элементов в её детях. Хранить дерево будем в массиве, так же, как хранили двоичную кучу: корень дерева имеет номер 1, дети вершины i имеют номера $2i$, $2i + 1$.

В каждой вершине оказалась записана сумма на одном из подотрезков массива: в корне записана сумма на всём массиве, в его детях — сумма на левой и правой половинах массива, в их детях — на четвертях массива, и так далее; на нижнем уровне в каждой вершине записана сумма на подотрезке массива длины один. Оказывается, зная суммы на этих подотрезках массива, можно найти сумму на произвольном подотрезке за $O(\log n)$.

<table>
<tr><td colspan="8" align="center">17</td></tr>
<tr><td colspan="4" align="center">8</td><td colspan="4" align="center">9</td></tr>
<tr><td colspan="2" align="center">3</td><td colspan="2" align="center">5</td><td colspan="2" align="center">6</td><td colspan="2" align="center">3</td></tr>
<tr><td>1</td><td>2</td><td>4</td><td>1</td><td>0</td><td>6</td><td>3</td><td>0</td></tr>
</table>

Рис. 8.1: Дерево отрезков на массиве $1, 2, 4, 1, 0, 6, 3$

Описанную процедуру построения дерева несложно реализовать за $O(n)$.

```
class Tree:
  int sz # количество элементов на нижнем уровне
  vector<int> x

  # построить дерево отрезков по массиву a длины n
  build(n, a):
    for (sz = 1; sz < n; sz *= 2);
    x.assign(2 * sz, 0)
    for i = 0..(n - 1):
      x[sz + i] = a[i]
    for i = (sz - 1)..1:
      x[i] = x[2 * i] + x[2 * i + 1]
```

8.2 Реализация операций "снизу"

Изменение значения в точке

Для того, чтобы изменить значение одного элемента, нужно пересчитать значения сумм для всех вершин дерева, подотрезки которых содержат этот элемент. Такие вершины — это вершина-лист, соответствующая этому элементу, а также все её предки. Таким образом, операцию можно выполнить за время, пропорциональное глубине дерева, то есть за $O(\log n)$.

```
1  # присвоить значение newVal в a[pos]
2  set(pos, newVal):
3    # вершина-лист, соответствующая a[pos], имеет номер pos + sz
4    pos += sz
5    x[pos] = newVal
6    for (pos /= 2; pos > 0; pos /= 2):
7      x[pos] = x[2 * pos] + x[2 * pos + 1]
```

Запрос на отрезке

Поймём, почему следующий алгоритм корректно найдёт сумму на подотрезке $a[l, r)$:

```
1  # найти сумму элементов в a[l, r)
2  get(l, r):
3    sum = 0
4    for (l += sz, r += sz; l < r; l /= 2, r /= 2):
5      if l % 2 == 1:
6        sum += x[l], l += 1
7      if r % 2 == 1:
8        r -= 1, sum += x[r]
9    return sum
```

Заметим, что на k-м шаге цикла отрезок $[l, r)$ — это ровно множество номеров вершин на k-м снизу уровне, подотрезки которых целиком содержатся в запрошенном отрезке массива. Это верно для самого нижнего уровня; покажем теперь, что если это верно на k-м шаге, то верно и на $(k+1)$-м.

Действительно, если l — левый сын $\lfloor l/2 \rfloor$, то $\lfloor l/2 \rfloor$ является самой левой из вершин на предыдущем уровне, отрезок которой целиком содержится в отрезке запроса (если такие вершины на предыдущем уровне вообще есть). Если же l — правый сын $\lfloor l/2 \rfloor$, то отрезок вершины $\lfloor l/2 \rfloor$ не содержится в отрезке запроса (так как там не содержится $(l-1)$ — левый сын $\lfloor l/2 \rfloor$); зато отрезок вершины $\lfloor (l+1)/2 \rfloor = \lfloor l/2 \rfloor + 1$ в отрезке запроса уже содержится (либо это неверно ни для одной вершины на предыдущем уровне). Аналогичные рассуждения можно провести для r.

Остаётся заметить, что мы возвращаем сумму значений по таким вершинам, подотрезки которых содержатся в отрезке запроса, но для подотрезков родителей которых это уже не так. При этом мы учли каждую такую вершину, поскольку такая вершина всегда самая левая либо самая правая на своём уровне среди вершин, отрезки которых содержатся в отрезке запроса. Наконец, каждому элементу отрезка запроса соответствует ровно одна такая вершина дерева, значит, мы нашли ровно сумму элементов в отрезке запроса.

> **Следствие 8.2.1** Любой отрезок можно разбить на $O(\log n)$ (примерно $2\log_2 n$) отрезков, соответствующих вершинам дерева.

Поскольку на каждом шаге $r - l$ уменьшается хотя бы в два раза, время работы есть $O(1 + \log(r - l))$, или, более грубо, $O(\log n)$.

8.3 Реализация операций “сверху”

Запрос на отрезке

Альтернативный способ реализации запроса на отрезке: будем рекурсивно обходить дерево, начиная с корня. Для текущей вершины возможны три варианта:

- Отрезок вершины не пересекается с отрезком запроса. В этом случае вершина и всё её поддерево не нужны для вычисления ответа на запрос, поэтому нужно просто завершить работу.
- Отрезок вершины целиком содержится в отрезке запроса. В этом случае надо добавить значение в вершине к ответу и завершить работу.
- Отрезок вершины пересекает отрезок запроса. В этом случае рекурсивно запустимся от обоих детей.

```
# текущая вершина - v, ей соответствует отрезок a[L, R)
# l, r - границы запроса
get(v, L, R, l, r):
  if L >= r or l >= R: # отрезки не пересекаются
    return 0
  if l <= L and R <= r: # отрезок вершины вложен в отрезок запроса
    return x[v]
  # остался случай пересечения отрезков
  M = (L + R) / 2
  return get(2 * v, L, M, l, r) + get(2 * v + 1, M, R, l, r)

# найти сумму элементов в a[l, r)
get(l, r):
  return get(1, 0, sz, l, r)
```

На каждом уровне мы посетим не более четырёх вершин. Действительно, на каждом уровне дерева третий случай выполнится для не более чем двух вершин (пересекающей левую границу отрезка запроса и пересекающей правую границу отрезка запроса), значит, на каждом уровне произойдёт не более четырёх рекурсивных запусков. Таким образом, время работы есть $O(\log n)$.

Заметим, что оценка $O(1 + \log(r - l))$ к этой реализации не применима.

Изменение значения в точке

Вторую операцию можно реализовать аналогично.

```
set(v, L, R, pos, newVal):
  if pos < L or R <= pos:
    return
  if L == pos and R == pos + 1:
    x[pos] = newVal
    return
  M = (L + R) / 2
  set(2 * v, L, M, pos, newVal)
  set(2 * v + 1, M, R, pos, newVal)
  x[v] = x[2 * v] + x[2 * v + 1]

set(pos, newVal):
  return set(1, 0, sz, pos, newVal)
```

Этот способ реализации операций работает дольше (из-за рекурсивных вызовов). Однако большинство модификаций дерева отрезков, которые мы будем изучать дальше, применимы лишь к этому способу.

8.4 Область применения дерева отрезков

С помощью дерева отрезков можно быстро вычислять не только сумму элементов на отрезке, но и, например, минимум или максимум на отрезке. Пусть мы хотим вычислять функцию $\oplus$ на отрезке; какие свойства от неё требуются?

Функция $\oplus$ должна быть *ассоциативной* ($a \oplus (b \oplus c) = (a \oplus b) \oplus c$ для любых a, b, c), иначе значение в вершине дерева нельзя будет вычислить по значениям в детях этой вершины. Кроме того, нужно, чтобы для этой функции существовал *нейтральный элемент*: такой ε, что $a \oplus \varepsilon = \varepsilon \oplus a = a$ для любого a. Это нужно, чтобы длину массива можно было увеличить до ближайшей степени двойки.

Заметим, что коммутативности от функции требовать не обязательно (то есть с помощью дерева отрезков можно вычислять, например, произведение матриц на подотрезке).

Множество, на котором определена ассоциативная функция, относительно которой есть нейтральный элемент, называют *моноидом*. Таким образом, в общем случае дерево отрезков применимо в следующей ситуации: есть некоторый моноид M с операцией $\oplus$ и нейтральным элементом ε. Каждому элементу массива a_i некоторым образом сопоставляется элемент этого моноида $m_i \in M$. Запрос на отрезке — нахождение значения $m_l \oplus \dots \oplus m_{r-1}$.

Приведём несколько примеров:

- Запрос — минимум или максимум на отрезке. Нейтральный элемент — бесконечность с нужным знаком (на практике — элемент, заведомо больший или меньший всех остальных).
- Запрос — $\gcd$ элементов на подотрезке. Нейтральный элемент в этом случае — 0.
- Запрос — множество различных элементов на подотрезке. В этом случае $m_i = \{a_i\}$, нейтральный элемент — пустое множество $\emptyset$, $\oplus = \cup$.
- Запрос — номер первого ненулевого элемента на отрезке. В этом случае элемент моноида — пара чисел: длина отрезка и номер первого ненулевого элемента в этом отрезке, или бесконечность, если такого элемента нет. $m_i = (1, 0)$, если $a_i \neq 0$; $m_i = (1, \infty)$ иначе. $(a, b) \oplus (c, d) = (a + c, \min(b, a + d))$. Нейтральный элемент — $(0, \infty)$.

8.5 Изменение элементов на подотрезке

Пусть теперь мы хотим научиться быстро выполнять более общую операцию — изменять значение не одного элемента, а сразу всех элементов на некотором подотрезке. Для определённости сформулируем операцию так: даны $0 \leqslant l < r \leqslant n$ и Δ; нужно увеличить на Δ значения всех элементов в отрезке $a[l, r)$.

Можно было бы вызвать операцию изменения отдельно для каждого элемента на отрезке $a[l, r)$. Хочется, однако, чтобы время выполнения операции не зависело от длины отрезка.

Отложенные операции

Будем пользоваться реализацией операций “сверху”. Заметим, что перед тем, как посетить вершину v, мы обязательно посещаем всех её предков. Заведём дополнительный массив upd, в котором будем хранить отложенные операции: в upd_v будет храниться величина, которую нужно добавить к каждому элементу в подотрезке вершины v; при этом эта величина **пока** не учтена ни одним потомком вершины v, но уже учтена самой вершиной v. Другими словами, настоящее значение суммы на подотрезке любой вершины v равняется

$$x_v + (R_v - L_v) \cdot \sum_{u \text{ — предок } v} upd_u,$$

где $a[L_v, R_v)$ — подотрезок массива, соответствующий вершине v.

Вначале $upd_v = 0$ для любой вершины v. Операцию изменения на отрезке будем делать всё тем же рекурсивным способом "сверху". При этом для вершины, отрезок которой целиком содержится в отрезке операции, будем пересчитывать не только x_v, но и upd_v.

Если в какой-то момент из вершины v нужно сделать рекурсивные вызовы в детей, то перед тем, как делать эти вызовы, пересчитаем значения x_{2v}, x_{2v+1}, upd_{2v}, upd_{2v+1} с учётом значения upd_v, а upd_v заменим на ноль.

Теперь, если мы спускаемся в вершину v, то точно верно, что x_v посчитано корректно. Действительно, все изменения, затрагивающие целиком подотрезок вершины v (или подотрезок предка v), к этому моменту уже оказались учтены в x_v. При применении изменений мы, как и раньше, будем при выходе из рекурсии пересчитывать значение в вершине через значения в её детях, поэтому изменения, затрагивающие лишь часть подотрезка вершины v, тоже будут учтены.

Поскольку мы по-прежнему посещаем $O(\log n)$ вершин при выполнении каждой операции, время работы каждой операции по-прежнему составляет $O(\log n)$.

```
1  push(v, L, R):
2    if upd[v] == 0:
3      return
4    M = (L + R) / 2
5    upd[2 * v] += upd[v]
6    x[2 * v] += upd[v] * (M - L)
7    upd[2 * v + 1] += upd[v]
8    x[2 * v + 1] += upd[v] * (R - M)
9    upd[v] = 0
10
11 add(v, L, R, l, r, delta):
12   if L >= r or l >= R:
13     return
14   if l <= L and R <= r:
15     x[v] += delta * (R - L)
16     upd[v] += delta
17     return
18   push(v, L, R)
19   M = (L + R) / 2
20   add(2 * v, L, M, l, r, delta)
21   add(2 * v + 1, M, R, l, r, delta)
22   x[v] = x[2 * v] + x[2 * v + 1]
23
24 add(l, r, delta):
25   add(1, 0, sz, l, r, delta)
26
27 get(v, L, R, l, r):
28   if L >= r or l >= R:
29     return 0
30   if l <= L and R <= r:
31     return x[v]
32   push(v, L, R)
33   M = (L + R) / 2
34   return get(2 * v, L, M, l, r) + get(2 * v + 1, M, R, l, r)
35
36 get(l, r):
37   return get(1, 0, sz, l, r)
```

Область применения

Таким же способом можно, например, быстро присваивать одно и то же значение всем элементам на подотрезке. В этом случае значение upd_v при изменениях нужно не скла-

дывать, а заменять на новое значение; при этом можно использовать значение ∞ в качестве нейтрального элемента: если $upd_v = \infty$, то на самом деле никаких изменений, затрагивающих весь отрезок v, не было (либо они уже были учтены).

Снова попробуем сформулировать в общем виде, когда такой метод применим. Пусть значения upd — элементы некоторого множества F; значения upd должно быть можно комбинировать друг с другом ("накапливать" отложенные операции в вершине), то есть нужна также некоторая функция $\times$ на множестве F. Эта функция снова должна быть ассоциативной (мы не знаем, в каком порядке будут происходить "склеивания" отложенных операций друг с другом). Кроме того, относительно этой функции должен существовать нейтральный элемент (поскольку отложенных операций в вершине может и не быть). Таким образом, F с операцией $\times$ снова образуют некоторой моноид; нейтральный элемент F обозначим за δ.

Это ещё не всё: изменения нужно уметь применять к значению в вершине, то есть нужна ещё некоторая функция $\wedge : M \times F \to M$. Она должна быть согласована с операцией "склейки" отложенных операций: $a \wedge (f \times g) = (a \wedge f) \wedge g$ для любых $a \in M$, $f, g \in F$. Кроме того, разумеется, нейтральный элемент не должен изменять элемент M: $a \wedge \delta = a$ для любого $a \in M$. Функцию $\wedge$, удовлетворяющую таким свойствам, называют *действием моноида F на множестве M*.

Но и это ещё не всё: действие $\wedge$ должно быть согласовано и с операцией моноида M для того, чтобы изменения можно было "проталкивать" из вершины в её детей: $(a \oplus b) \wedge f = (a \wedge f) \oplus (b \wedge f)$ для любых $f \in F$, $a, b \in M$. Это свойство действия $\wedge$ называют *дистрибутивностью*.

Итак, операция изменения на подотрезке всегда соответствует дистрибутивному действию некоторого моноида F на моноид M.

Вернёмся к рассмотренным выше примерам: требуется уметь вычислять сумму на подотрезке и либо увеличивать значения всех элементов на подотрезке на Δ, либо присваивать новое значение всем элементам на подотрезке. Моноид для операции $\oplus$ состоит из пар чисел: длины отрезка и суммы на нём ($(a, b) \oplus (c, d) = (a + c, b + d)$; $\varepsilon = (0, 0)$).

В случае изменения на отрезке моноид отложенных операций состоит из чисел. При этом $\delta = 0$, $f \times g = f + g$, $(l, b) \wedge f = (l, b + f \cdot l)$.

В случае присвоения на отрезке моноид отложенных операций состоит из чисел и $\delta = \infty$. При этом $f \times g = g$, если $g \neq \delta$; $f \times \delta = f$; $(l, b) \wedge f = (l, f \cdot l)$.

8.6 Динамическое дерево отрезков

Пусть теперь мы хотим работать с массивом a очень большой длины n, такой, что целиком хранить массив в памяти компьютера не получится (например, $n = 10^{18}$). Пусть изначально массив заполнен нейтральными элементами, и мы хотим снова уметь применять операцию $\wedge$ на подотрезке, и находить значение $a_l \oplus \cdots \oplus a_{r-1}$ для произвольного подотрезка $a[l, r)$ (требования на $\wedge$ и $\oplus$ те же, что и раньше).

Первый способ: хеш-таблицы

Вместо массивов будем использовать хеш-таблицы. При первом обращении к вершине инициализируем значения в ней нейтральными элементами (поскольку в начале работы весь массив состоит из нейтральных элементов, то же самое верно и для каждой вершины дерева).

Теперь операции работают за $O(\log n)$ в среднем. При этом каждая операция может посетить не более $O(\log n)$ вершин, которые до этого не посещались. Поэтому после выполнения k операций дерево будет использовать $O(\min(n, k \log n))$ памяти (если число

операций известно заранее, можно сразу создать хеш-таблицу такого размера; иначе можно поддерживать хеш-таблицу размера, пропорционального числу уже созданных вершин, и удваивать её размер по мере надобности; в этом случае оценка времени работы становится амортизированной).

Второй способ: указатели

Можно хранить каждую вершину отдельно, при этом в вершине храня указатели на её детей.

```
class Node:
  Node *l, *r
  int x, upd
```

Вначале достаточно инициализировать только корень дерева. Когда мы хотим сделать рекурсивные вызовы от детей некоторой вершины в первый раз, создадим этих детей и инициализируем значения в них нейтральными элементами. При такой реализации операции работают за $O(\log n)$ в худшем случае, оценка на используемую память снова $O(\min(n, k \log n))$.

Третий способ: сжатие координат

Этот способ работает, лишь если все операции, которые нужно выполнить, даны заранее. Границы всех k операций делят массив на $O(k)$ подотрезков. Эти подотрезки обладают замечательным свойством: каждый из них либо содержится, либо не пересекает любой из k отрезков операций. Тогда заменим каждый из $O(k)$ подотрезков на один элемент, и получим новый массив размера $O(k)$. Любая операция на отрезке исходного массива соответствует операции на некотором подотрезке нового массива. Построим дерево отрезков на новом массиве и будем выполнять каждую операцию за $O(\log k)$.

Один из способов реализации сжатия координат — выписать все границы операций в отдельный массив, отсортировать, удалить повторяющиеся числа. После этого для каждой операции двоичным поиском найти границы её отрезка в полученном массиве; найденные двоичным поиском индексы и будут границами отрезка операции в новом массиве. Возможно, для каждого элемента нового массива потребуется дополнительно запомнить длину отрезка, который был заменён на этот элемент (например, если нужно считать сумму на отрезке).

Получаем $O(k \log k)$ времени на подготовку и $O(\log k)$ времени на выполнение каждой операции. При этом требуется $O(k)$ памяти.

8.7 Персистентное дерево отрезков

Структура данных называется *персистентной (persistent)*, если при изменении структуры сохраняются все её предыдущие версии и доступ к ним. Персистетность бывает *частичной (partial persistence)*, если изменять можно только последнюю версию структуры, а со старыми версиями можно осуществлять лишь не изменяющие их операции; и *полной (full persistence)*, если изменять можно и любую из старых версий.

Самый простой способ реализации персистентности — полностью копировать структуру каждый раз, когда в ней что-то требуется поменять. Разумеется, такой способ почти никогда не является оптимальным ни по используемой памяти, ни по времени выполнения операций.

Реализуем полную персистентность для дерева отрезков следующим образом: будем хранить дерево не в массиве, а при помощи указателей (как в одной из реализаций

динамического дерева отрезков). Каждый раз, когда в вершине v нужно изменить какое-либо значение (x_v, upd_v, либо указатель на ребёнка), создадим новую копию этой вершины v_{new} и изменим значение уже в ней.

При этом в новой версии дерева родитель v должен иметь ребёнком уже не v, а v_{new}, поэтому для него тоже придётся создать новую копию. То же верно, и для родителя родителя, и так далее. Таким образом, для каждой вершины, в которой что-то поменялось, мы создаём новые копии не только этой вершины, но и всех вершин на пути от корня до неё.

Удобно сделать так, чтобы каждая функция возвращала новую версию вершины, от которой она была вызвана. Приведём пример персистентной реализации функций `push` и add:

```
class Node:
  Node *l, *r
  int x, upd

  copy():
    v = new Node()
    v->l = l, v->r = r, v->x = x, v->upd = upd
    return v

  push(L, R):
    if upd == 0:
      return this
    M = (L + R) / 2
    v = copy()
    v->l = l->copy(), v->r = r->copy()
    v->l->upd += upd, v->r->upd += upd
    v->l->x += upd * (M - L), v->r->x += upd * (R - M)
    v->upd = 0
    return v

  add(L, R, l, r, delta):
    if L >= r or l >= R:
      return this
    if l <= L and R <= r:
      v = copy()
      v->x += delta * (R - L)
      v->upd += delta
      return v
    v = push(L, R)
    M = (L + R) / 2
    v->l = v->l->add(L, M, l, r, delta)
    v->r = v->r->add(M, R, l, r, delta)
    v->x = v->l->x + v->r->x
    return v

class Tree:
  int sz
  vector<Node *> roots # список версий

  # увеличить значения a[l, r) на delta в i-й версии дерева
  add(i, l, r, delta):
    newRoot = roots[i]->add(0, sz, l, r, delta)
    roots.push_back(newRoot)
    return newRoot
```

Поскольку каждая операция по-прежнему посещает $O(\log n)$ вершин, время работы любой из операций по-прежнему есть $O(\log n)$. При этом в процессе выполнения каждой

из операций может появиться порядка $\log n$ новых вершин, поэтому после выполнения k операций дерево будет использовать $O(n + k \log n)$ памяти.

Персистентный массив

Теперь мы умеем реализовывать и персистентный массив: просто будем хранить массив на нижнем уровне персистентного дерева; в промежуточных вершинах можно ничего не хранить. В полученном массиве можно получить доступ к элементу за $O(\log n)$; после выполнения k операций он использует $O(n + k \log n)$ памяти.

Имея персистентный массив, несложно сделать персистентной и любую структуру данных, реализуемую с помощью массива (стек, очередь, дек и т.п.) — просто вместо обычного массива будем использовать персистентный. Разумеется, для многих структур существует более простая и эффективная реализация персистентности.

8.8 Две модельных задачи

Сформулируем модельные задачи, на примере которых мы изучим ещё несколько техник, применимых к дереву отрезков.

Первая задача

Даны n точек на плоскости: $(x_1, y_1), \ldots, (x_n, y_n)$. Требуется отвечать на следующие запросы: сколько точек лежит в прямоугольнике $[X_1, X_2) \times [Y_1, Y_2)$?

Вторая задача

Дан массив чисел $a_1, \ldots, a_n$. Требуется отвечать на следующие запросы: сколько в подотрезке массива $a[l, r)$ чисел, лежащих в диапазоне $[d, u)$ (то есть сколько таких $l \leqslant i < r$, что $d \leqslant a_i < u$)?

Связь между задачами

Заметим, что эти задачи в каком-то смысле эквивалентны друг другу. Для того, чтобы свести вторую задачу к первой, достаточно рассмотреть точки $(1, a_1), \ldots, (n, a_n)$.

Сведём теперь первую задачу ко второй. Будем считать, что точки упорядочены по возрастанию x_i (если это не так, отсортируем их). Положим $a_i = y_i$. Для того, чтобы понять, какой подотрезок отсортированного списка точек удовлетворяет $X_1 \leqslant x_i < X_2$, сделаем два двоичных поиска по отсортированному списку x_i. Остаётся на этом подотрезке массива ответить на запрос с $d = Y_1, u = Y_2$.

Заметим также, что достаточно научиться отвечать на такие запросы в первой задаче, в которых $X_1 = -\infty, Y_1 = -\infty$. Действительно, обозначим количество точек в $[X_1, X_2) \times [Y_1, Y_2)$ за $c(X_1, X_2, Y_1, Y_2)$. Тогда

$$c(X_1, X_2, Y_1, Y_2) = c(-\infty, X_2, -\infty, Y_2) - c(-\infty, X_1, -\infty, Y_2)$$

$$-c(-\infty, X_2, -\infty, Y_1) + c(-\infty, X_1, -\infty, Y_1).$$

По тем же причинам во второй задаче достаточно научиться отвечать на запросы, в которых $d = -\infty$ (то есть на запрос “сколько элементов в $a[l, r)$ меньше u?”), и/или $l = 0$.

Мы будем решать первую либо вторую задачу с запросами в исходной либо упрощённой форме, в зависимости от того, какая формулировка удобнее в конкретном случае.

8.9 Дерево отрезков сортированных массивов

Будем решать вторую задачу с запросами “сколько элементов в $a[l, r)$ меньше u?”. В каждой вершине дерева отрезков будем хранить элементы подотрезка массива, соответствующего этой вершине, отсортированные по возрастанию. Уже имея отсортированные подотрезки детей, отсортированный подотрезок в вершине можно найти, используя функцию `merge` (как в сортировке слиянием). Поскольку суммарная длина подотрезков вершин на каждом уровне равна n, дерево можно построить за $O(n \log n)$; при этом оно будет использовать $O(n \log n)$ памяти.

<table>
<tr><td colspan="8">$0, 1, 1, 2, 3, 4, 6, \infty$</td></tr>
<tr><td colspan="4">$1, 1, 2, 4$</td><td colspan="4">$0, 3, 6, \infty$</td></tr>
<tr><td colspan="2">$1, 2$</td><td colspan="2">$1, 4$</td><td colspan="2">$0, 6$</td><td colspan="2">$3, \infty$</td></tr>
<tr><td>1</td><td>2</td><td>4</td><td>1</td><td>0</td><td>6</td><td>3</td><td>∞</td></tr>
</table>

Рис. 8.2: Дерево отрезков сортированных массивов на массиве $1, 2, 4, 1, 0, 6, 3$

```
class Tree:
  int sz
  vector<vector<int> > x

  # построить дерево отрезков сортированных массивов по массиву a длины n
  build(n, a):
    for (sz = 1; sz < n; sz *= 2);
    x.assign(2 * sz)
    for i = 0..(n - 1):
      x[sz + i] = {a[i]}
    for i = (sz - 1)..1:
      x[i] = merge(x[2 * i], x[2 * i + 1])
```

Как теперь отвечать на запрос? Вспомним, что любой отрезок можно поделить на $O(\log n)$ подотрезков, соответствующих вершинам дерева. Если для каждого такого отрезка имеется отсортированный список элементов в нём, достаточно сделать по каждому из них двоичный поиск. Получаем решение за $O(\log^2 n)$ на запрос.

```
  get(v, L, R, l, r, u):
    if L >= r or l >= R:
      return 0
    if l <= L and R <= r:
      return lower_bound(x[v].begin(), x[v].end(), u) - x[v].begin()
    M = (L + R) / 2
    return get(2 * v, L, M, l, r, u) + get(2 * v + 1, M, R, l, r, u)

  get(l, r, u):
    return get(1, 0, sz, l, r, u)
```

8.10 Fractional cascading

Научимся ускорять предыдущее решение. Для этого в каждой вершине дерева будем хранить дополнительный массив: пусть $fromLeft_v[i]$ — количество элементов $x_v[0, i)$, которые “пришли” из левого ребёнка вершины v. Значение $fromLeft_v$ несложно найти в функции `merge`.

Как этот дополнительный массив помогает быстрее отвечать на запросы? Пусть $lb_v(u)$ — количество элементов, меньших u, в подотрезке вершины v. Заметим, что

$$lb_{2 \cdot v}(u) = fromLeft_v[lb_v(u)], \ lb_{2 \cdot v+1}(u) = lb_v(u) - fromLeft_v[lb_v(u)].$$

```
merge(a, b):
  x = vector<int>(), fromLeft = vector<int>()
  for (i = 0, j = 0; i < a.size() or j < b.size();)
    fromLeft.push_back(i)
    if i == a.size() or (j < b.size() and a[i] > b[j]):
      x.push_back(b[j]), j += 1
    else:
      x.push_back(a[i]), i += 1
  fromLeft.push_back(a.size())
  return x, fromLeft
```

<table>
<tr><td colspan="8">0, 0, 1, 2, 3, 3, 4, 4, 4
0, 1, 1, 2, 3, 4, 6, ∞</td></tr>
<tr><td colspan="4">0, 1, 1, 2, 2
1, 1, 2, 4</td><td colspan="4">0, 1, 1, 2, 2
0, 3, 6, ∞</td></tr>
<tr><td colspan="2">0, 1, 1
1, 2</td><td colspan="2">0, 0, 1
1, 4</td><td colspan="2">0, 1, 1
0, 6</td><td colspan="2">0, 1, 1
3, ∞</td></tr>
<tr><td>1</td><td>2</td><td>4</td><td>1</td><td>0</td><td>6</td><td>3</td><td>∞</td></tr>
</table>

Рис. 8.3: Дерево отрезков сортированных массивов с массивом $fromLeft$, построенное на массиве $1, 2, 4, 1, 0, 6, 3$. Массив $fromLeft$ помечен синим цветом. Красным цветом помечены элементы, меньшие 4.

Таким образом, зная количество элементов, меньших u, в отрезке вершины v, легко посчитать количество таких элементов в отрезках её детей. Двоичный поиск достаточно запустить один раз в корне, после чего результаты двоичных поисков в остальных вершинах можно восстанавливать за $O(1)$ в процессе рекурсивного спуска. Получаем решение за $O(\log n)$ на запрос.

```
get(v, L, R, l, r, cnt):
  if L >= r or l >= R:
    return 0
  if l <= L and R <= r:
    return cnt
  M = (L + R) / 2
  return get(2 * v, L, M, l, r, fromLeft[v][cnt])
      + get(2 * v + 1, M, R, l, r, cnt - fromLeft[v][cnt])

get(l, r, u):
  cnt = lower_bound(x[1].begin(), x[1].end(), u) - x[1].begin()
  return get(1, 0, sz, l, r, cnt)
```

8.11 Многомерное дерево отрезков

Вместо отсортированного списка элементов в каждой вершине дерева отрезков можно хранить и что-то более сложное; например, ещё одно дерево отрезков.

Трёхмерная версия первой задачи

Даны n точек в трёхмерном пространстве: $(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)$. Требуется отвечать на следующие запросы: сколько точек лежит в параллелепипеде $[X_1, X_2) \times [Y_1, Y_2) \times [Z_1, Z_2)$?

Решение

Будем считать, что точки упорядочены по возрастанию x_i (отсортируем их, если это не так). Построим дерево отрезков на массиве точек. В каждую вершину v этого дерева отрезков запишем a_v — отсортированный по возрастанию y_i список точек, лежащих в отрезке этой вершины. Кроме того, в каждой вершине v построим t_v — дерево отрезков сортированных массивов на массиве, полученном выписыванием z-координат точек из a_v.

Описанная структура занимает $O(n \log^2 n)$ памяти ($O(n \log n)$ на каждый уровень внешнего дерева); построить её можно тоже за $O(n \log^2 n)$, если пользоваться функцией `merge` и во внешнем, и во внутренних деревьях.

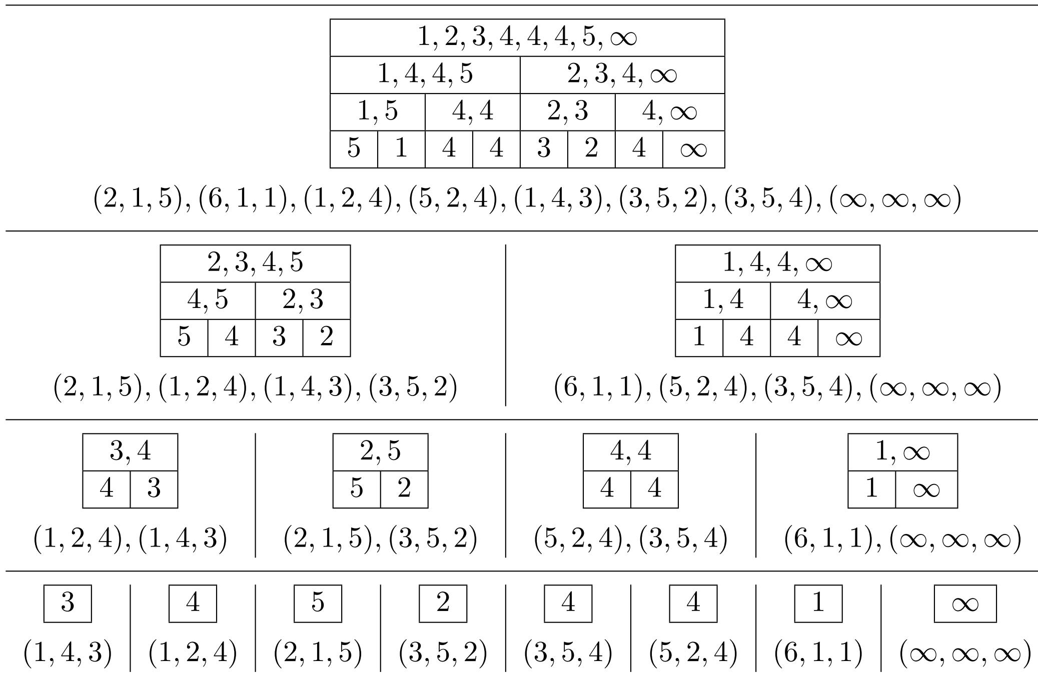


Рис. 8.4: Описанная выше структура, построенная на точках $(1, 4, 3)$, $(1, 2, 4)$, $(2, 1, 5)$, $(3, 5, 2)$, $(3, 5, 4)$, $(5, 2, 4)$, $(6, 1, 1)$

Как теперь узнать, сколько точек лежит в параллелепипеде $[X_1, X_2) \times [Y_1, Y_2) \times [Z_1, Z_2)$? Двоичными поисками по исходному массиву найдём такие l_x, r_x, что $X_1 \leqslant x_i < X_2$ тогда и только тогда, когда $l_x \leqslant i < r_x$. Найдём разбиение отрезка $[l_x, r_x)$ на $O(\log n)$ отрезков, соответствующих вершинам внешнего дерева. С каждой вершиной v, соответствующей отрезку этого разбиения, проделаем следующие действия: двоичными поисками найдём границы $[l_y, r_y)$ подотрезка a_v, точки в котором удовлетворяют условию $Y_1 \leqslant y < Y_2$. Запросом к t_v найдём количество точек в $a_v[l_y, r_y)$, удовлетворяющих условию $Z_1 \leqslant z < Z_2$. Сумма результатов этих запросов по всем v — искомый ответ.

Таким образом, мы научились отвечать на запрос за $O(\log^2 n)$ (если использовать технику fractional cascading во внутренних деревьях). Можно применить технику fractional cascading и ко внешнему дереву, но это не даст лучшей оценки времени (всё равно нужно сделать $O(\log n)$ запросов ко внутренним деревьям).

Многомерная версия первой задачи

Даны n точек в k-мерном пространстве: $(x_1^1, \ldots, x_k^1), \ldots, (x_1^n, \ldots, x_k^n)$. Требуется отвечать на следующие запросы: сколько точек лежит в параллелепипеде $[L_1, R_1) \times \cdots \times [L_k, R_k)$?

Решение

Построим на отсортированном по возрастанию x_1 списке точек дерево отрезков; в каждую его вершину запишем отсортированный по x_2 список точек, лежащих в отрезке этой вершины. На этом списке рекурсивно построим дерево отрезков, решающее задачу для $(k-1)$-мерного пространства (у каждой точки отбросим первую координату). На последнем уровне вложения $(k = 2)$ построим дерево отрезков сортированных массивов.

Полученная структура занимает $O(n \log^{k-1} n)$ памяти ($O(n \log^{k-2} n)$ на каждом уровне внешнего дерева) и позволяет отвечать на запрос за $O(\log^{k-1} n)$: нужно двоичными поисками найти подотрезок исходного массива, точки в котором удовлетворяют условию $X_1 \leqslant x_1 < X_2$; разбить этот отрезок на $O(\log n)$ отрезков, соответствующих вершинам дерева; для каждого из отрезков разбиения рекурсивно решить $(k - 1)$-мерную задачу; при $k = 2$ ответить на запрос за $O(\log n)$, пользуясь техникой fractional cascading. Как и в трёхмерном случае, применение техники к деревьям не на максимальном уровне вложения не улучшает оценку времени работы.

8.12 Метод сканирующей прямой

Вернёмся к модельным задачам. Альтернативный способ решения первой задачи — интерпретировать одну из осей координат как временную ось. Тогда и точку, и запрос можно интерпретировать как происходящее в какой-то момент событие; эти события нужно обработать в том порядке, в котором они происходят. В некотором смысле этот приём позволяет уменьшить размерность задачи на единицу. Для начала изучим этот способ на примере одномерной задачи.

Одномерная версия первой задачи

Даны n точек $x_1, \ldots, x_n$ и m отрезков $[l_1, r_1), \ldots, [l_m, r_m)$. Нужно для каждого отрезка посчитать количество точек, которые он содержит.

Решение одномерной задачи

Как обычно, достаточно научиться решать задачу для лучей: ответ для $[l, r)$ равен разности ответов для $(-\infty, r)$ и $(-\infty, l)$. Будем говорить, что i-я точка появляется в момент времени x_i. Тогда ответ на запрос $(-\infty, l)$ — это количество точек, которые появились до момента времени l. Таким образом, есть события двух типов:

1. в момент времени $t = x_i$ появляется новая точка;
2. в момент времени $t = l$ требуется узнать, сколько точек появилось **до** этого момента.

Выпишем все события в один массив и отсортируем их по возрастанию t (при равенстве t будем считать, что событие второго типа "меньше" события первого типа). Теперь ответить на все запросы очень просто: будем идти по массиву слева направо и увеличивать счётчик, когда появляется новая точка. Тогда ответ на запрос — значение счётчика в момент обработки события, соответствующего этому запросу. Получаем решение за $O((n + m) \log(n + m))$.

Решение offline-версии первой задачи

Теперь решим *offline*-версию первой задачи: пусть заранее известны не только точки, но и все запросы. Можно считать, что все запросы имеют вид $[-\infty, X) \times [Y_1, Y_2)$. Интерпретируем первую координату как время. Тогда есть события двух типов:

1. (t, y): в момент времени $t = x_i$ появляется новая точка с координатой $y = y_i$;
2. (t, l, r): в момент времени $t = X$ требуется узнать, сколько точек с координатами в отрезке $[l, r) = [Y_1, Y_2)$ появилось **до** этого момента.

При этом можно считать, что $0 \leqslant y_i < n$ для любого i: поскольку точки известны заранее, можно произвести сжатие координат.

Пусть a_y — количество точек, в данный момент времени имеющих координату y. Изначально все a_y равны нулю. Снова отсортируем события по возрастанию t; событие второго типа "меньше" события первого типа с таким же t.

Теперь, если случается событие (t, y), нужно увеличить a_y на единицу; если случается событие (t, l, r), то ответ на соответствующий запрос — сумма $a_l + \cdots + a_{r-1}$. Будем поддерживать дерево отрезков на a, тогда любое событие можно обработать за $O(\log n)$. Получаем решение за $O((n + m)\log(n + m))$, где m — число запросов.

Решение первой задачи

Что делать, если запросы заранее не известны? Заметим, что ответ на запрос мы находили, делая запрос к дереву отрезков в какой-то момент времени. Сделаем дерево персистентным, тогда мы будем иметь доступ к версии дерева на любой момент времени, то есть получим возможность отвечать на запросы, не зная их заранее.

Таким образом, теперь у нас есть только события первого типа. Мы обрабатываем их в порядке возрастания времени, при этом сохраняя все версии дерева отрезков. После этого начинаем принимать запросы. Когда приходит очередной запрос, нужно двоичным поиском по массиву событий найти нужную версию дерева, и сделать к ней запрос на отрезке. Таким образом, потратив $O(n \log n)$ времени на подготовку, мы сможем отвечать на приходящие запросы за $O(\log n)$. При этом потребуется $O(n \log n)$ памяти, чтобы сохранить все версии дерева.

8.13 k-я порядковая статистика на отрезке

Рассмотрим ещё одну модельную задачу: пусть дан массив чисел $a_0, \ldots, a_{n-1}$; требуется по l, r, k уметь находить k-ю порядковую статистику на $a[l, r)$.

Решение за $O(\log^2 n)$

Сделаем двоичный поиск по ответу: пусть x фиксировано, тогда ответ на запрос больше или равен x тогда и только тогда, когда в $a[l, r)$ меньше k элементов, меньших x. На такой запрос мы умеем отвечать за $O(\log n)$ (с помощью техники fractional cascading, либо с помощью метода сканирующей прямой и персистентного дерева отрезков). Поскольку ответ — какой-то элемент массива a, двоичный поиск можно делать не по диапазону чисел, а по элементам отсортированной копии массива a (альтернатива — провести сжатие координат, после которого $0 \leqslant a_i < n$ для любого i). Получаем решение за $O(\log^2 n)$.

Оптимизация двоичного поиска со вложенным спуском по дереву отрезков

Алгоритм, делающий запросы к дереву отрезков внутри двоичного поиска, часто можно соптимизировать. Сначала рассмотрим такую оптимизацию на примере задачи попроще: пусть дан массив a из нулей и единиц; требуется уметь менять значение элемента, а также находить номер k-й слева единицы в a.

Решение за $O(\log^2 n)$ устроено очень просто: будем поддерживать на массиве a дерево отрезков, умеющее находить сумму на подотрезке. Тогда номер k-й единицы можно найти двоичным поиском: этот номер больше или равен s тогда и только тогда, когда сумма на отрезке $[0, s)$ меньше k.

Теперь решим задачу быстрее: реализуем функцию, которая по v, k будет находить номер k-й единицы в подотрезке вершины v. В левом ребёнке вершины v хранится x_{2v} — сумма на первой половине отрезка. Если $x_{2v} \geqslant k$, то k-я единица находится в первой

половине отрезка, то есть в подотрезке левого ребёнка v. Тогда просто запустим функцию рекурсивно с параметрами $2v$, k. Если же $x_{2v} < k$, то k-я единица находится в правой половине отрезка, то есть в подотрезке правого ребёнка; найти её номер — то же самое, что и найти номер $(k - x_{2v})$-й единицы в подотрезке правого ребёнка v. Таким образом, в этом случае запустим функцию рекурсивно с параметрами $2v + 1$, $k - x_{2v}$. Получаем решение за $O(\log n)$.

```
# предполагаем, что в массиве есть k единиц, то есть запрос корректен
getKthOne(v, L, R, k):
  if R - L == 1: # пришли в лист
    return 0
  M = (L + R) / 2
  if x[2 * v] >= k:
    return getKthOne(2 * v, L, M, k)
  else:
    return getKthOne(2 * v + 1, M, R, k - x[2 * v]) + (M - L)

getKthOne(k):
  return getKthOne(1, 0, sz, k)
```

Поиск k-й порядковой статистики на отрезке за $O(\log n)$

Попытаемся точно так же соптимизировать алгоритм поиска k-й порядковой статистики на отрезке за $O(\log^2 n)$ методом сканирующей прямой. Для начала более подробно опишем, как работает этот алгоритм.

Можно считать, что $0 \leqslant a_i < n$ (если это не так, сделаем сжатие координат). Мы строим $n + 1$ версию дерева отрезков на массиве длины n: в нулевой версии дерева все элементы равны нулю; $(i + 1)$-я версия дерева отличается от i-й увеличением ячейки с номером a_i на один. Количество элементов в $a[l, r)$, меньших x — это разность результатов запросов на сумму на отрезке $[0, x)$ к r-й и l-й версиям дерева.

Вместо того, чтобы делать двоичный поиск со вложенными запросами к деревьям, будем одновременно спускаться по r-й и l-й версиям дерева. Реализуем функцию, по k, v_r, v_l находящую значение k-го по возрастанию элемента среди тех элементов $a[l, r)$, чьи значения лежат в отрезке вершин v_r, v_l (это вершины из r-й и l-й версий дерева, которым соответствует один и тот же отрезок).

Пусть q — разность значений, хранящихся в левых детях v_r, v_l; q — количество элементов в $a[l, r)$, чьи значения лежат в левой половине отрезка вершин v_r, v_l. Если $q \geqslant k$, то ответ лежит в отрезке левых детей v_r, v_l; тогда рекурсивно запустим функцию с тем же k от левых детей v_r, v_l. Если же $q < k$, то ответ лежит в отрезке правых детей v_r, v_l; тогда рекурсивно запустим функцию с параметром $k - q$ от правых детей v_r, v_l.

Когда мы попадаем на нижний уровень деревьев, единственное значение, лежащее в отрезке текущих вершин — это ответ на запрос.

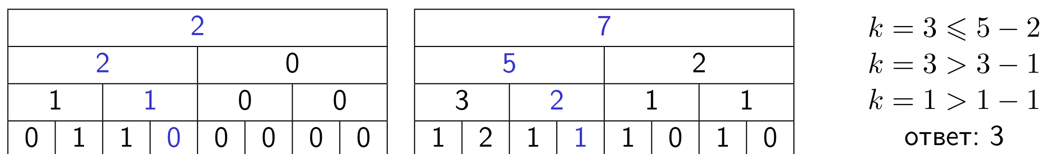


Рис. 8.5: Пример выполнения функции с параметрами $l = 2$, $r = 7$, $k = 3$ на массиве $1, 2, 4, 1, 0, 6, 3$. Слева изображеная 2-я версия дерева, справа — 7-я версия дерева. Пройденный функцией путь изображён синим цветом.

```
getKthOrderStatistic(vl, vr, L, R, k):
  if R - L == 1:
    return L
  M = (L + R) / 2
  q = vr->l->x - vl->l->x
  if q >= k:
    return getKthOrderStatistic(vl->l, vr->l, L, M, k)
  else:
    return getKthOrderStatistic(vl->r, vr->r, M, R, k - q)

# предполагаем, что k >= r - l
getKthOrderStatistic(k, l, r):
  return getKthOrderStatistic(roots[l], roots[r], 0, sz, k)
```

9. AVL-дерево

9.1 Двоичное дерево поиска

Мы уже умеем поддерживать неупорядоченное множество ключей с операциями поиска, добавления и удаления элементов, в среднем работающими за $O(1)$ (с помощью хеш-таблиц).

Часто возникает потребность поддерживать **упорядоченное** множество ключей, поддерживающее больший набор операций: например, поиск следующего по значению ключа во множестве после данного; вычисление количества ключей во множестве, имеющих значения в заданном диапазоне, и т.п.

Один из способов реализации такого множества — *двоичное дерево поиска (binary search tree, BST)*. BST — это двоичное дерево, в каждой вершине которого хранится один из ключей. При этом для каждой вершины выполняется следующее свойство: если в вершине хранится ключ x, то в её левом поддереве (то есть в поддереве её левого ребёнка) значения всех ключей меньше x, а в правом поддереве (поддереве правого ребёнка) — больше x.

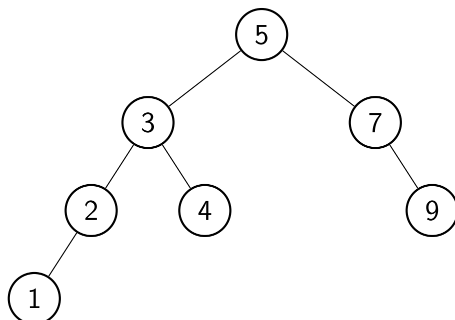


Рис. 9.1: Пример BST на множестве ключей $1, 5, 2, 4, 3, 7, 9$

R Мы для удобства будем далее считать, что все ключи различны. Есть несколько способов хранить равные ключи в BST:

- Можно пронумеровать все элементы по времени их первого появления, и для i-го по счёту элемента x в качестве ключа использовать пару (x, i); тогда все ключи будут различны.
- Можно хранить в вершине с ключом x количество таких ключей во множестве. Если ключи используются как индексы для хранения некоторой информации, как в ассоциативном массиве (т.е. дерево строится на парах (x, y); x — ключ, y — значение), то в вершине можно хранить список значений всех элементов с данным ключом.
- Можно требовать, чтобы в левом поддереве вершины x все ключи были меньше **или равны** x; или чтобы в правом поддереве все ключи были больше **или равны** x.

Помимо x, в вершине нужно хранить ссылки на левого и правого ребёнка. Также иногда хранят ссылку на родителя вершины.

9.2 Базовые операции

- `traverse`: требуется выписать все элементы в порядке увеличения ключей. Сначала рекурсивно выпишем все элементы в левом поддереве корня, потом выпишем элемент, лежащий в корне дерева, и, наконец, рекурсивно выпишем все элементы в правом поддереве корня.
- `find(x)`: требуется найти элемент с ключом x, если такой есть. Будем спускаться по дереву, начиная из корня. На каждом шаге пойдём в левое или в правое поддерево, в зависимости от того, больше или меньше x ключа в текущей вершине. Остановимся, когда придём в вершину с ключом x, либо когда попытаемся пойти в несуществующего ребёнка.
- `insert(x)`: требуется вставить в дерево новый элемент x. Так же, как в `find`, будем спускаться по дереву, пока не придём в несуществующего ребёнка; сделаем этим ребёнком новую вершину с ключом x.
- `next(v)`: требуется найти следующую за v вершину в порядке увеличения ключей. Если у v есть правый ребёнок, то искомая вершина — самая левая в его поддереве. Иначе искомая вершина — ближайший к v предок, такой, что v находится в его левом поддереве.
  Эту операцию можно реализовать и не поддерживая ссылки на родителя, если предварительно спуститься к v из корня и выписать все вершины на пути.
- `prev(v)`: требуется найти предыдущую перед v вершину в порядке увеличения ключей. Реализация аналогична `next`.
- `delete(x)`: требуется удалить вершину с ключом x (если такая есть). Сначала сделаем поиск вершины с помощью функции `find`; дальше считаем, что нашлась вершина v. Если у v нет детей, то просто удалим её. Если у v есть только один ребёнок, то удалим v, а ребёнка поставим на её место. Наконец, если у v есть оба ребёнка, то найдём следующую по значению ключа за v вершину w с помощью функции `next`. Эта вершина находится в правом поддереве v, у неё нет левого ребёнка. Тогда поменяем v и w местами, после чего удалим v одним из предыдущих способов.
- `lower_bound(x)` (`upper_bound(x)`): требуется найти вершину с минимальным ключом, больше или равным x (строго большим x). Будем спускаться по дереву, начиная из корня. Из текущей вершины пойдём в левое поддерево, если значение ключа в ней больше или равно x (больше x), и в правое поддерево иначе. На каждом шаге верно, что искомая вершина — либо текущая, либо находится в поддереве, куда мы пошли. Значит, вернуть нужно последнюю посещённую вершину, которая удовлетворяла требуемому условию.

BST несложно сделать полностью персистентным: нужно, как и в случае с деревом отрезков, создавать новую копию вершины каждый раз, когда в ней требуется что-то поменять. Важный момент: в такой реализации персистентного BST нельзя поддерживать ссылку на родителя, поскольку у одной вершины могут быть разные родители в разных версиях дерева. Тем не менее, поскольку все операции можно реализовать и не используя ссылки на родителя, персистентное BST поддерживает все те же операции, что и обычное BST.

Операция `traverse` работает за $O(n)$, где n — число вершин в дереве. Остальные операции работают за $O(h)$, где h — высота дерева. Таким образом, для того, чтобы операции работали быстро, нужно научиться как-то контролировать высоту дерева. Будем называть BST *сбалансированным*, если существует такое C, что в любой момент времени высота дерева не превосходит $C \log n$, где n — текущее число вершин в дереве. Другими словами, в сбалансированном BST на n вершинах высота есть $O(\log n)$.

9.3 AVL-дерево

AVL-дерево (Адельсон-Вельский, Ландис, 1962) — такое BST, что для любой вершины высоты её левого и правого поддеревьев отличаются не больше, чем на единицу.

Теорема 9.3.1 Высота AVL-дерева на n вершинах есть $O(\log n)$.

Доказательство. Пусть m_h — минимально возможное число вершин в AVL-дереве высоты h. Покажем по индукции, что $m_h = F_{h+1} - 1$, где F_n — n-е число Фибоначчи.

Действительно, дерево высоты 0 — пустое дерево, которое состоит из $F_1 - 1 = 0$ вершин; дерево высоты 1 — это дерево, состоящее из $F_2 - 1 = 1$ вершины. Далее, заметим, что в дереве высоты $h \geqslant 2$ хотя бы одно из поддеревьев детей корня имеет высоту $h - 1$; тогда второе поддерево имеет высоту хотя бы $h - 2$. Значит,

$$m_h = 1 + m_{h-1} + m_{h-2} = 1 + F_h - 1 + F_{h-1} - 1 = F_{h+1} - 1.$$

Таким образом,

$$m_h = F_{h+1} - 1 \geqslant 2^{\lfloor (h+1)/2 \rfloor} - 1 \geqslant 2^{h/2} - 1$$

(на самом деле даже $m_h = \Omega(\varphi^h)$, где $\varphi = \frac{1+\sqrt{5}}{2}$). Пусть h — высота AVL-дерева на n вершинах, тогда $2^{h/2} - 1 \leqslant m_h \leqslant n$. Тогда $h \leqslant \log_{\sqrt{2}}(n+1) = O(\log n)$. ■

9.4 Балансировка

Будем обозначать высоту поддерева вершины v за $h(v)$, а разность высот поддеревьев детей v за $diff(v) = h(v{\to}l) - h(v{\to}r)$. В AVL-дереве $diff(v) \in \{-1, 0, 1\}$ для любой вершины v. После вставки или удаления вершины свойство AVL-дерева может нарушиться: для вершины v на пути от корня к вставленной (удалённой) вершине значение $diff(v)$ может оказаться равно ± 2. Для того, чтобы восстановить свойство AVL-дерева, после вставки (удаления) вершины будем подниматься от неё к корню и при необходимости проводить *балансировку*.

Операция балансировки принимает дерево, для корня v которого $diff(v) = \pm 2$; при этом для всех потомков корня свойство AVL-дерева выполняется. Операция перестраивает дерево так, чтобы оно стало AVL-деревом. Перестройка заключается в выполнении одного из следующих вращений, каждое из которых можно выполнить за $O(1)$:

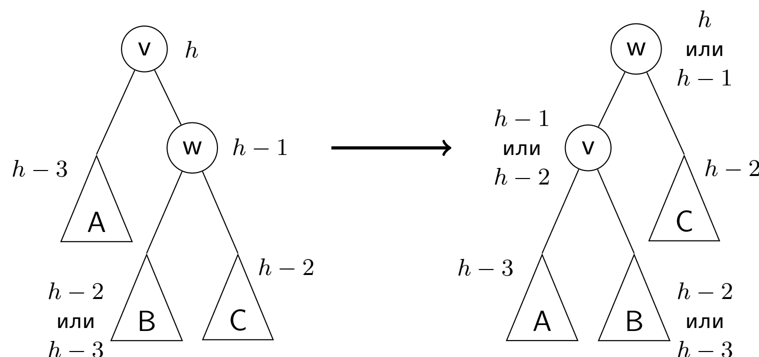


Рис. 9.2: **Малое левое вращение**: выполняется при $diff(v) = -2$, $diff(w) \leqslant 0$, где $w = v{\to}r$.

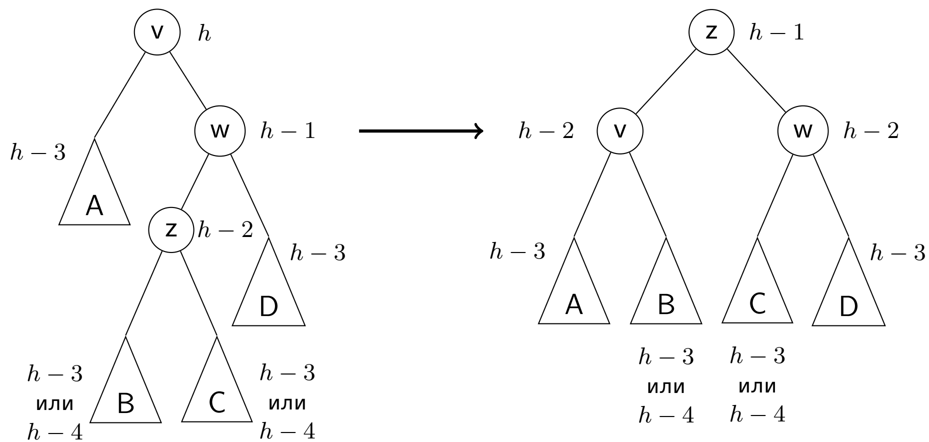


Рис. 9.3: **Большое левое вращение**: выполняется при $diff(v) = -2$, $diff(w) = 1$, где $w = v{\to}r$. Используется обозначение $z = w{\to}l$.

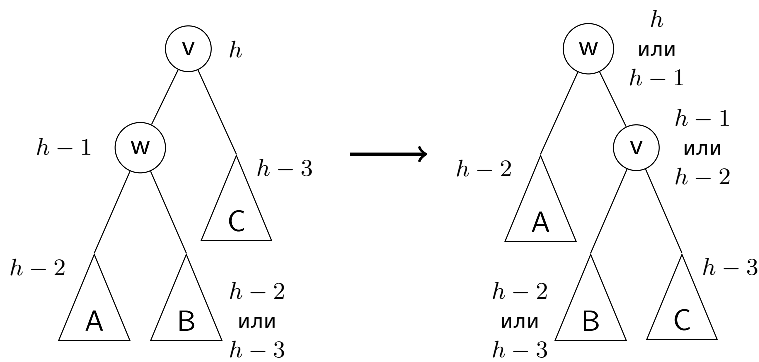


Рис. 9.4: **Малое правое вращение**: выполняется при $diff(v) = 2$, $diff(w) \geqslant 0$, где $w = v{\to}l$.

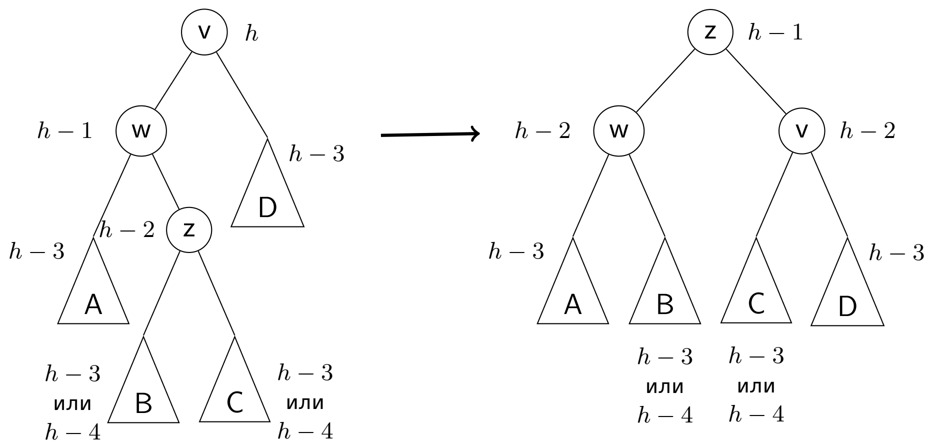


Рис. 9.5: **Большое правое вращение**: выполняется при $diff(v) = 2$, $diff(w) = -1$, где $w = v{\to}l$. Используется обозначение $z = w{\to}r$.

9.5 Реализация

При реализации удобно пользоваться тем, что большое левое вращение является комбинацией малого правого вращения относительно вершины w и малого левого вращения относительно вершины v. Аналогично, большое правое вращение является комбинацией малого левого вращения относительно вершины w и малого правого вращения относительно вершины v. В качестве указателя на пустое поддерево будем использовать указатель на специальную фиктивную вершину `empty`, такую, что `empty->h = 0`.

```
rotateLeft(v):
  w = v->r
  v->r = w->l
  w->l = v
  v->h = max(v->l->h, v->r->h) + 1
  w->h = max(w->l->h, w->r->h) + 1

rotateRight(v):
  w = v->l
  v->l = w->r
  w->r = v
  v->h = max(v->l->h, v->r->h) + 1
  w->h = max(w->l->h, w->r->h) + 1

bigRotateLeft(v):
  rotateRight(v->r)
  rotateLeft(v)

bigRotateRight(v):
  rotateLeft(v->l)
  rotateRight(v)

rebalance(v):
  diff = v->l->h - v->r->h
  if abs(diff) <= 1:
    v->h = max(v->l->h, v->r->h) + 1
  else if diff == -2:
    if v->r->l->h - v->r->r->h <= 0:
      rotateLeft(v)
    else:
      bigRotateLeft(v)
  else if diff == 2:
    if v->l->l->h - v->l->r->h >= 0:
      rotateRight(v)
    else:
      bigRotateRight(v)

# возвращает корень текущего поддерева после балансировки
insert(v, x):
  if v == empty:
    v = new Node(x)
    return v
  if v->x > x:
    v->l = insert(v->l, x)
  else:
    v->r = insert(v->r, x)
  rebalance(v)
  return v

# вставить в дерево ключ x; считаем, что такого ключа в дереве ещё нет
root = insert(root, x)
```

Операцию `delete` нужно модифицировать так же, как и `insert`: при выходе из рекурсии запускать балансировку текущей вершины. В остальных изученных нами операциях структура дерева не изменяется, поэтому запускать балансировку не требуется.

Поскольку все операции по-прежнему можно реализовать, не поддерживая ссылок на родителей, AVL-дерево можно сделать полностью персистентным стандартным для BST способом. При этом каждая операция, меняющая дерево, будет создавать $O(\log n)$ новых вершин.

Заметим, что в описанной выше реализации в каждой вершине дерева требуется дополнительно хранить высоту поддерева этой вершины. При более аккуратной реализации вращений достаточно хранить в вершине не абсолютную высоту, а лишь разность высот её детей (то есть одно из трёх значений $\{-1, 0, 1\}$). Таким образом, при аккуратной реализации AVL-дерево использует лишь два дополнительных бита памяти на вершину.

10. Декартово дерево

10.1 Декартово дерево

Декартово дерево (treap) (Seidel, Aragon, 1989) на множестве пар $(x_1, y_1), \ldots, (x_n, y_n)$ — это двоичное дерево, в каждой вершине которого лежит одна из пар; при этом оно является BST по x_i и двоичной кучей с минимумом в корне по y_i. x_i называют ключами, y_i — приоритетами.

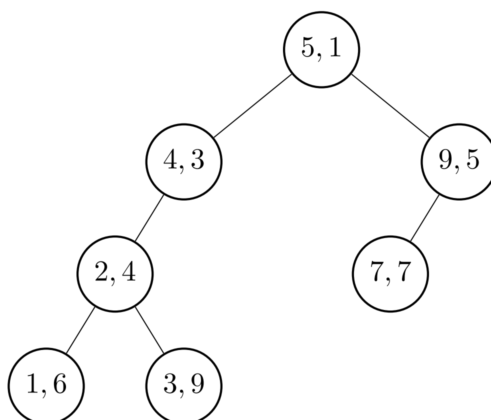


Рис. 10.1: Декартово дерево на множестве пар $(1, 6)$, $(5, 1)$, $(2, 4)$, $(4, 3)$, $(3, 9)$, $(7, 7)$, $(9, 5)$

Лемма 10.1.1 Декартово дерево на множестве пар $(x_1, y_1), \ldots, (x_n, y_n)$ единственно, если все x_i попарно различны, и все y_i попарно различны.

Доказательство. По свойству кучи корнем является вершина (x_i, y_i) с минимальным y_i. В левом поддереве корня лежат вершины с $x_j < x_i$; в правом поддереве — с $x_j > x_i$. Поддеревья единственны по индукции. ∎

Чуть позже мы покажем, что если в качестве приоритетов использовать случайные числа, то высота декартова дерева в среднем будет оцениваться как $O(\log n)$.

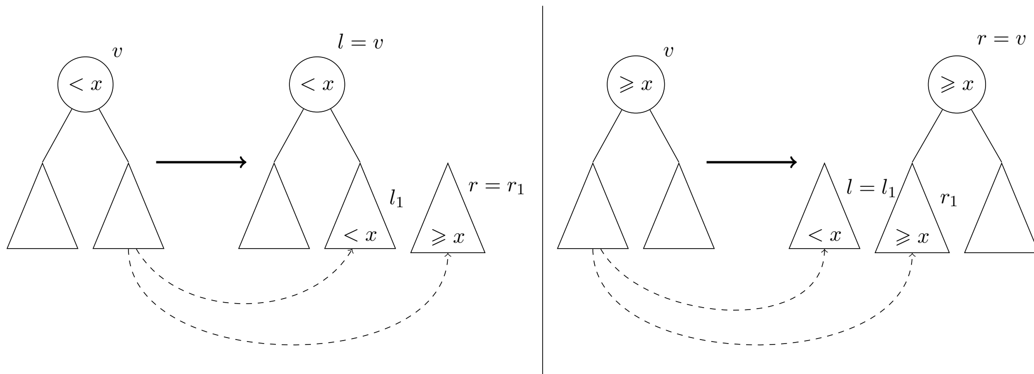
10.2 Операции Split и Merge

Преимущество декартова дерева — простая реализация операций `split` и `merge`, с помощью которых можно в том числе выразить все базовые операции.

Для краткости с этого момента иногда будем писать "дерево v", имея в виду дерево с корнем в вершине v.

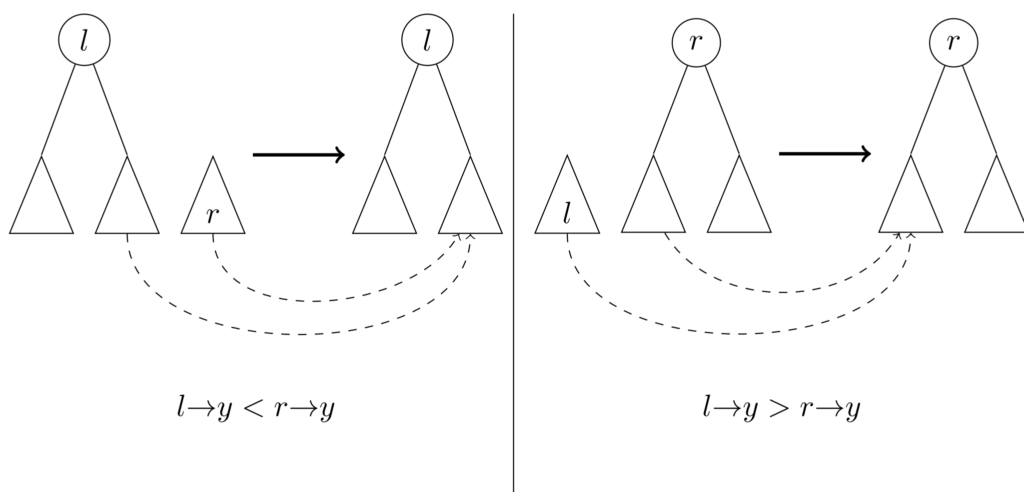
- `split(v, x)`: требуется разделить дерево v на два дерева l, r: в l должны оказаться все ключи, меньшие x, в r — все ключи, больше или равные x.
  Если $v{\to}x < x$, то все вершины с ключами, больше или равными x, находятся в поддереве $v{\to}r$. Тогда рекурсивно разделим дерево $v{\to}r$ по ключу x на два дерева l_1 и r_1, и подвесим l_1 правым ребёнком к v; $l = v$ и $r = r_1$ — искомые деревья.

Если же $v \to x \geqslant x$, то все вершины с ключами, меньшими x, находятся в поддереве $v \to l$. Тогда рекурсивно разделим дерево $v \to l$ по ключу x на два дерева l_1 и r_1, и подвесим r_1 левым ребёнком к v; $l = l_1$ и $r = v$ — искомые деревья.
Заметим, что в обоих случаях условия на приоритеты не нарушились.




```
1  # возвращает l, r
2  split(v, x):
3    if v == empty:
4      return empty, empty
5    if v->x < x:
6      v->r, r = split(v->r, x)
7      return v, r
8    else:
9      l, v->l = split(v->l, x)
10     return l, v
```

- `merge(l, r)`: любой ключ из дерева l меньше любого ключа из дерева r; требуется объединить l и r в одно дерево.
  Корнем нового дерева будет та из вершин l, r, у которой меньше приоритет. Если $l \to y < r \to y$, то рекурсивно сольём $l \to r$ и r, и подвесим результат правым ребёнком к l. Если же $r \to y < l \to y$, то, наоборот, рекурсивно сольём l и $r \to l$, и подвесим результат левым ребёнком к r. В обоих случаях условия на ключи окажутся выполненными.



```
merge(l, r):
  if l == empty:
    return r
  if r == empty:
    return l
  if l->y < r->y:
    l->r = merge(l->r, r)
    return l
  else:
    r->l = merge(l, r->l)
    return r
```

R Операции `split` и `merge` можно реализовать и для AVL-деревьев, но для них они устроены несколько сложнее.

10.3 Остальные операции

В каждом случае способы перечислены в порядке усложнения реализации и уменьшения константы в оценке времени работы.

- `find`, `lower_bound`, `upper_bound`, `next`, `prev`:
  - Можно выразить эти операции через `split` и `merge`. Например, `lower_bound(x)` можно реализовать так: запустим `split` по x, он вернёт l и r; искомая вершина — самая левая в r. Нужно не забыть в конце сделать `merge(l, r)`, чтобы восстановить исходное дерево.
  - Можно реализовать эти операции так же, как и в обычном BST.
- `insert(x, y)` (далее v — новая вершина с ключом x и приоритетом y):
  - Можно разделить дерево с помощью `split` по x на l и r, после чего вызвать `merge(l, merge(v, r))`.
  - Можно спускаться по дереву, начиная из корня (идти в левое или правое поддерево, в зависимости от результата сравнения ключей текущей и вставляемой вершины), пока не придём в вершину w с приоритетом, большим y (или в пустое поддерево). Сделаем `split` поддерева w по ключу x, и подвесим результаты к v левым и правым ребёнком, а v подвесим туда, где раньше находилась w.
  - Можно сделать `insert` так, как он делается в обычном BST. Вставленная вершина v может не удовлетворять условию на приоритеты, поэтому будем поднимать её вверх вращением вокруг ребра из v в родителя (то есть малым левым или малым правым вращением), пока приоритет v не окажется больше приоритета текущего родителя, либо пока v не окажется корнем.
- `delete(x)`:
  - Можно "вырезать" вершину, которую требуется удалить, сделав `split` всего дерева по ключу x на l и r, и `split` дерева r по ключу $x+1$ на v и r_1. После этого нужно вызвать `merge(l, r1)` и освободить память, занимаемую v.
  - Можно найти в дереве вершину, которую требуется удалить, после чего записать на её место результат `merge` её детей.
  - Наконец, можно найти вершину v, которую требуется удалить, после чего опускать её вниз, выполняя вращение вокруг ребра между v и тем из её детей, чей приоритет меньше. Удалим v в момент, когда у неё станет не больше одного ребёнка (если один ребёнок есть, поместим его на место v).

Поскольку все операции можно реализовать, не поддерживая ссылки на родителей, декартово дерево можно сделать полностью персистентным стандартным способом. При этом операция, меняющая дерево, будет в среднем создавать $O(\log n)$ новых вершин.

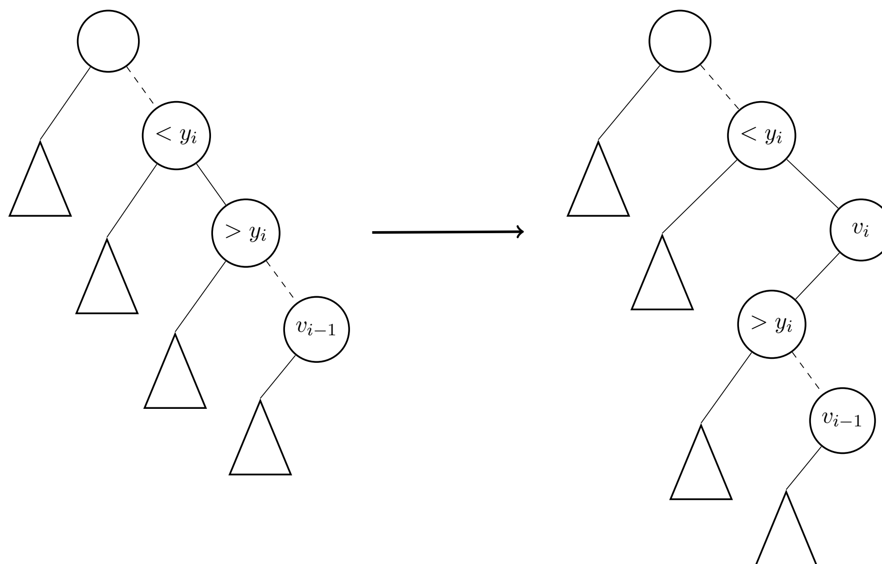
R Можно показать, что математическое ожидание числа вращений в последних версиях реализации `insert` и `delete` есть $O(1)$. Это полезно тем, что позволяет реализовать **частично** персистентное декартово дерево, создающее в среднем $O(1)$ новых вершин за операцию. При этом используется метод, несколько более сложный, чем изученный нами способ сделать BST персистентным.

10.4 Построение

Пусть мы хотим построить декартово дерево на заданном наборе ключей $x_1, \ldots, x_n$ (и случайных приоритетах $y_1, \ldots, y_n$). Простой способ построить дерево — это начать с пустого дерева и поочерёдно вставлять новые вершины операцией `insert`. Такой способ в среднем потребует $O(n \log n)$ времени.

Если ключи $x_1, \ldots, x_n$ **даны в отсортированном порядке**, то построить декартово дерево можно за $O(n)$. Будем вставлять вершины в порядке увеличения ключа, при этом будем поддерживать указатель на последнюю вставленную вершину.

Пусть мы хотим вставить очередную вершину v_i с ключом x_i и приоритетом y_i. Поскольку последняя вставленная вершина v_{i-1} имеет максимальный ключ в дереве, путь из корня в неё всегда состоит только из рёбер в правых детей. Если $y_{i-1} < y_i$, то просто сделаем v_i правым ребёнком v_{i-1}. Иначе будем подниматься из v_{i-1} вверх, пока не окажемся в корне, либо пока приоритет родителя текущей вершины не окажется меньше y_i. Поставим v_i на место текущей вершины, а её сделаем левым ребёнком v_i.



Заметим, что если в процессе алгоритма мы поднялись из какой-то вершины наверх, то больше мы в эту вершину не попадём. Значит, алгоритм работает за $O(n)$.

Также заметим, что алгоритм можно реализовать, не поддерживая ссылки на родителей: достаточно поддерживать в стеке путь из корня в текущую вершину.

10.5 Случайное дерево поиска

Случайное дерево поиска (random binary search tree, RBST) на множестве попарно различных ключей $x_1, \ldots, x_n$ — BST, построенное на этом множестве следующим алгоритмом: в корень помещается ключ, выбранный среди всех ключей случайно равновероятно; после этого на меньших ключах тем же алгоритмом рекурсивно строится левое поддерево, а на больших ключах — правое поддерево.

Предложение 10.5.1 Математическое ожидание высоты RBST на n вершинах есть $O(\log n)$.

Доказательство. Рассмотрим следующие случайные величины: пусть h_n — высота RBST на n вершинах, $Y_n = 2^{h_n}$ — *экспоненциальная высота* RBST на n вершинах. Заметим, что $Y_1 = 2$. Пусть теперь $n \geqslant 2$, χ_i — случайная величина, равная единице, если i-й по возрастанию ключ попал в корень, и нулю иначе. Тогда

$$Y_n = \sum_{i=1}^{n} \chi_i \cdot 2 \cdot \max(Y_{i-1}, Y_{n-i}).$$

Тогда, поскольку χ_i и Y_{i-1}, Y_{n-i} независимы, получаем

$$\mathbb{E}Y_n = \sum_{i=1}^{n} \mathbb{E}\chi_i \cdot 2 \cdot \mathbb{E}(\max(Y_{i-1}, Y_{n-i})) \leqslant \frac{2}{n}\left(\sum_{i=2}^{n-1}(\mathbb{E}Y_{i-1} + \mathbb{E}Y_{n-i}) + 2\mathbb{E}Y_{n-1}\right) = \frac{4}{n}\sum_{i=1}^{n-1}\mathbb{E}Y_i$$

(мы пользуемся тем, что $\max(Y_0, Y_{n-1}) = Y_{n-1}$). Пользуясь полученным неравенством, докажем по индукции, что $\mathbb{E}Y_n \leqslant 2n^3$. Действительно, $\mathbb{E}Y_1 = 2 \leqslant 2 \cdot 1^3$. Если же $n \geqslant 2$, то

$$\mathbb{E}Y_n \leqslant \frac{4}{n}\sum_{i=1}^{n-1}\mathbb{E}Y_i \leqslant \frac{4}{n}\sum_{i=1}^{n-1} 2i^3 = \frac{8}{n} \cdot \frac{(n-1)^2 n^2}{4} = 2(n-1)^2 n \leqslant 2n^3.$$

Поскольку функция $f(x) = 2^x$ выпукла, к ней применимо *неравенство Йенсена*: если $p_i > 0$, $\sum_i p_i = 1$, то

$$f\left(\sum_i p_i x_i\right) \leqslant \sum_i p_i f(x_i);$$

или, в терминах математического ожидания, $f(\mathbb{E}X) \leqslant \mathbb{E}(f(X))$. Тогда

$$2^{\mathbb{E}h_n} \leqslant \mathbb{E}2^{h_n} = \mathbb{E}Y_n \leqslant 2n^3,$$

откуда

$$\mathbb{E}h_n \leqslant 1 + 3\log_2 n = O(\log n).$$

■

Связь с декартовым деревом

Заметим, что если приоритеты для вершин в декартовом дереве были выбраны случайно, то каждый ключ имеет одну и ту же вероятность оказаться в корне; при фиксированном корне каждый ключ в левом поддереве имеет одну и ту же вероятность оказаться в корне этого поддерева, и так далее. Значит, такое декартово дерево — RBST. В частности, для декартова дерева со случайно выбранными приоритетами выполняется оценка $O(\log n)$ на математическое ожидание высоты.

R Важно, чтобы все приоритеты в декартовом дереве были попарно различны. На практике, как правило, достаточно выбирать приоритеты случайно из достаточно большого диапазона целых чисел. Так, если приоритет — случайное число от 0 до n^2q-1, то вероятность совпадения приоритетов для каждой пары вершин равна $\frac{1}{n^2q}$, тогда математическое ожидание числа совпавших пар приоритетов не превышает $\frac{n(n-1)}{2} \cdot \frac{1}{n^2q} \leqslant \frac{1}{2q}$. Значит, по неравенству Маркова, вероятность того, что хоть какая-то пара приоритетов совпала, также не превышает $\frac{1}{2q}$.

10.6 Дополнительные операции

R Дальнейшие рассуждения применимы к любым BST, а не только к декартовым деревьям.

Вычисление функции на отрезке

Пусть помимо ключа в каждой вершине хранятся некоторые дополнительные данные: $v{\to}data$. Тогда можно быстро вычислять, например, минимум этих данных на подотрезке отсортированного по ключам списка элементов. В общем случае можно вычислять произвольную ассоциативную функцию на подотрезке (как и в случае дерева отрезков).

Будем дополнительно поддерживать в вершине минимум на её поддереве: в момент, когда у вершины v произошли изменения в поддереве, будем вызывать специальную функцию `update`, обновляющую значение этого минимума.

```
# empty->min = inf
update(v):
  v->min = min(v->data, v->l->min, v->r->min)
```

Теперь несложно за $O(\log n)$ по l, r найти

$$\min_{L \leqslant v\to x < R} v{\to}data.$$

- Можно сделать `split` дерева по ключу L на l и r, `split` r по ключу R на m и r_1, после чего извлечь ответ на запрос из m. Нужно не забыть склеить части дерева обратно в одно целое.
- Можно реализовать спуск по дереву из корня, аналогичный спуску по дереву отрезков.

Изменение значения в точке

Изменить значение $data$ в вершине с ключом x очень просто: нужно найти вершину, изменить значение, и пересчитать значения min для всех её предков.

Изменение значений на отрезке

Изменения на отрезке делаются методом отложенных операций полностью аналогично тому, как они делались в дереве отрезков (и с теми же требованиями к функции изменения). Рассмотрим для примера операцию увеличения на Δ значений $data$ для всех вершин с ключами в диапазоне $[L, R)$. Будем поддерживать в каждой вершине $v{\to}upd$ — величину, которую нужно добавить к $data$ в каждой вершине поддерева v; при этом эта величина ещё не учтена в потомках v, но уже учтена в самой v. Прежде, чем переходить из v в её детей, будем вызывать `push(v)`.

```
push(v):
  if v == empty or v->upd == 0:
    return
  for u in [v->l, v->r]:
    if u != empty:
        u->upd += v->upd
        u->min += v->upd
        u->data += v->upd
  v->upd = 0
```

Операцию увеличения значений на отрезке снова можно реализовать двумя способами: “вырезать” требуемый подотрезок, после чего изменить upd корня вырезанного поддерева, и склеить всё обратно; либо написать рекурсивный спуск по дереву, аналогичный спуску по дереву отрезков.

Запросы по индексам элементов

Будем поддерживать в каждой вершине размер её поддерева (его нужно пересчитывать в `update`, как и значение ассоциативной функции на поддереве). Используя информацию о размере поддерева, можно выполнять запросы не по значению ключа, а по *индексу* вершины (номеру в списке вершин, упорядоченном по возрастанию ключа; нумерация с нуля). В частности, можно реализовать следующие операции:

- `findByIndex(k)`: найти вершину с индексом k. Искомая вершина в левом поддереве, если его размер больше k; корень, если размер левого поддерева равен k; в правом поддереве, если размер левого поддерева меньше k. Таким образом, реализация функции аналогична `find`, только вместо неравенств на ключи проверяются неравенства на размеры поддеревьев.
- `getIndex(x)`: найти индекс вершины с ключом x. Выполняем `find(x)`, при этом добавляя к ответу размер левого поддерева и единицу каждый раз, когда идём в правое поддерево.
- `deleteByIndex(k)`: удалить вершину с индексом k. Реализация аналогична обычному `delete`, только ищется вершина с помощью `findByIndex`, а не `find`.
- `splitByIndex(v, k)`: разделить дерево v на два, в первом должны оказаться первые k вершин. Как и в `findByIndex`, нужно неравенства на ключи заменить на неравенства на размеры поддеревьев.
- Точно так же можно изменить операции запроса функции на отрезке и изменения значений на отрезке так, чтобы аргументами были не границы диапазона значений ключей, а границы индексов.
- Наконец, аналогично можно было бы изменить операцию `insert`, чтобы вставлять элемент на позицию с заданным индексом. Тут, однако, возникает проблема: вставляя элемент на произвольную позицию, очень легко нарушить условие двоичного дерева поиска.

Приведём для примера реализацию `splitByIndex`:

```
1 splitByIndex(v, k):
2   if v == empty:
3     return empty, empty
4   if v->l->sz < k:
5     v->r, r = splitByIndex(v->r, k - v->l->sz - 1)
6     return v, r
7   else:
8     l, v->l = splitByIndex(v->l, k)
9     return l, v
```

Неявный ключ

Пусть мы хотим поддерживать массив a, в который можно было бы вставлять или удалять элементы по индексу, а также считать функцию на подотрезке и делать изменения значений элементов в точке или на подотрезке; все эти операции хочется уметь выполнять за $O(\log n)$, где n — текущая длина массива. Будем поддерживать декартово дерево (или другое сбалансированное BST), ключами в котором будут индексы элементов массива, а хранящимися данными — значения этих элементов ($v{\to}x = i$, $v{\to}data = a_i$). Тогда мы уже умеем делать практически все вышеописанные операции; единственная проблема — при вставке элемента в середину массива нужно изменить индексы у всех элементов справа от него, то есть значения ключей для всех этих элементов; та же проблема возникает при удалении.

Заметим, однако, что операции, принимающие в качестве аргументов индексы, никак не используют ключи в вершинах. Тогда решить проблему очень просто: не будем хранить

ключи. Получившееся дерево — дерево двоичного поиска *по неявному ключу*. Операции `split`, `insert`, `delete` в нём в ходе работы проверяют неравенства на размеры поддеревьев вместо неравенств на ключи.

На самом деле мы умеем делать ещё больше операций: мы умеем делить массив на части (`split`); объединять массивы, дописывая один в конец ко второму (`merge`).

Приведём пример ещё одной операции: переворот подотрезка массива. Даются l, r; требуется "перевернуть" подотрезок массива $a[l,r)$, то есть произвести присвоения $a[l] = a[r-1]$, $a[l+1] = a[r-2]$,$\ldots$,$a[r-1] = a[l]$. Реализовать эту операцию можно методом отложенных операций: будем поддерживать в каждой вершине булеву величину $reverse$, имеющую истинное значение, если подотрезок, соответствующий поддереву вершины, надо перевернуть. В функции `push(v)` нужно в случае, когда $v{\to}reverse$ истинно, поменять местами левого и правого детей v, после чего изменить значения $reverse$ в детях на противоположные. Отметим, что такая операция осмысленна только в дереве по неявному ключу.

R Существует ещё один способ реализации BST с высотой, в среднем оценивающейся как $O(\log n)$. *Randomized binary search trees* (Martinez, Roura, 1997) вместо приоритетов поддерживают в каждой вершине v размер её поддерева $v{\to}sz$. При этом при выборе корня нового дерева в `merge` l выбирается корнем с вероятностью $\frac{l\to sz}{l\to sz + r\to sz}$, а r — с вероятностью $\frac{r\to sz}{l\to sz + r\to sz}$. Можно показать, что построенное таким образом дерево также является RBST. Преимуществом таких деревьев перед декартовыми является меньшее количество используемой памяти: на хранение приоритета, как правило, требуется больше бит, чем на поддержание размера поддерева; в BST по неявному ключу размер поддерева придётся поддерживать в любом случае. Недостатком таких деревьев является то, что при исполнении `merge` требуется $O(\log n)$ раз сгенерировать случайное число (что является достаточно трудоёмкой операцией).

11. Краткое отступление про B-деревья

11.1 B-дерево

> The more you think about what the B in B-trees means, the better you understand B-trees.
>
> Edward M. McCreight

B-дерево (Bayer, McCreight, 1972) — в некотором роде обобщение двоичного дерева поиска.

В каждой вершине B-дерева может храниться не один, а несколько ключей. При этом, если в вершине v, не являющейся листом дерева, хранится k ключей $a_1, \ldots, a_k$, то она имеет ровно $k+1$ ребёнка $u_1, \ldots, u_{k+1}$, причём значения ключей в поддереве u_1 меньше a_1, значения ключей в поддереве u_2 больше a_1, но меньше a_2, и так далее.

Также обычно требуют, чтобы во всех вершинах, кроме корня, хранилось от d до $2d$ ключей (где d — заранее выбранная константа), и чтобы все листья имели одинаковую глубину. Ясно, что такое дерево имеет высоту $O(\log_d n)$.

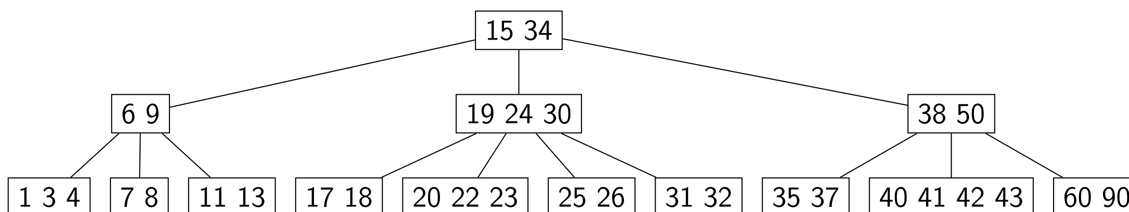


Рис. 11.1: Пример B-дерева с $d = 2$

Преимущество B-деревьев перед другими деревьями поиска проявляется при работе с большим объёмом данных. Если дерево не может храниться целиком в оперативной памяти, то приходится делать медленные запросы к жёсткому диску. B-дерево позволяет за один запрос к вершине получать не одну запись, а сразу много. Также оно имеет меньшую высоту, и позволяет реже производить балансировки. В том числе по этим причинам B-деревья часто используются в базах данных и в файловых системах.

11.2 2-3-дерево

2-3-дерево (Hopcroft, 1970) есть ни что иное, как частный случай B-дерева при $d = 1$ (2-3 — это возможные количества детей у вершины).

11.3 2-3-4-дерево и RB-дерево

2-3-4-дерево (Bayer, 1972) — это частный случай B-дерева (когда у вершины может быть от двух до четырёх детей).

В 1978 году на основе 2-3-4-дерева была предложена конструкция красно-чёрного дерева.

Красно-чёрное дерево (RB-дерево) (Guibas, Sedgewick, 1978) — двоичное дерево поиска, в котором у каждой вершины 0 или 2 ребёнка, при этом все листья — фиктивные вершины (в них не хранятся данные). Все вершины покрашены в два цвета (красный и чёрный) так, что корень и все листья покрашены в чёрный цвет, у любой красной вершины все дети чёрные, и на любом пути от корня до листа встречается одинаковое количество чёрных вершин.

Из последних двух свойств следует, что глубины любых двух листьев отличаются не более, чем в два раза. Можно показать, что из этого факта следует сбалансированность дерева.

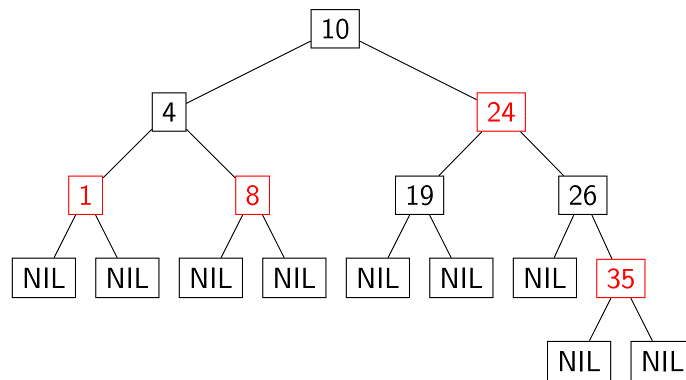


Рис. 11.2: Пример RB-дерева

При чём тут 2-3-4-деревья? Если склеить каждую красную вершину с её чёрным родителем, то получится как раз 2-3-4-дерево. Наоборот, если в 2-3-4-дереве у каждой вершины с двумя ключами создать правого красного ребёнка, а у каждой вершины с тремя ключами создать двух красных детей, получится RB-дерево. Одинаковая глубина листьев в 2-3-4-дереве соответствует последнему свойству RB-деревьев.

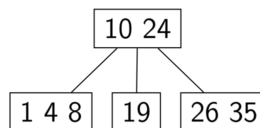


Рис. 11.3: 2-3-4-дерево, соответствующее RB-дереву с рис. 11.2

Операции добавления, удаления и поиска на RB-деревьях можно осуществлять за $O(\log n)$ в **худшем** случае. Благодаря этому, а также в целом высокой скорости работы по сравнению со многими деревьями поиска, RB-деревья достаточно часто применяются на практике. Так, они используются в некоторых имплементациях STL в `C++` (через них реализованы `set` и `map`). Планировщик процессов в современных ядрах Linux также использует RB-деревья.

R Любое AVL-дерево является RB-деревом (можно показать, что вершины любого AVL-дерева можно корректно покрасить в красный и чёрный цвета).

12. Splay-Дерево

12.1 Операция Splay

Splay-дерево (Sleator, Tarjan, 1985) — BST, обладающее дополнительным свойством: оно позволяет быстро получать доступ к данным, которые использовались недавно.

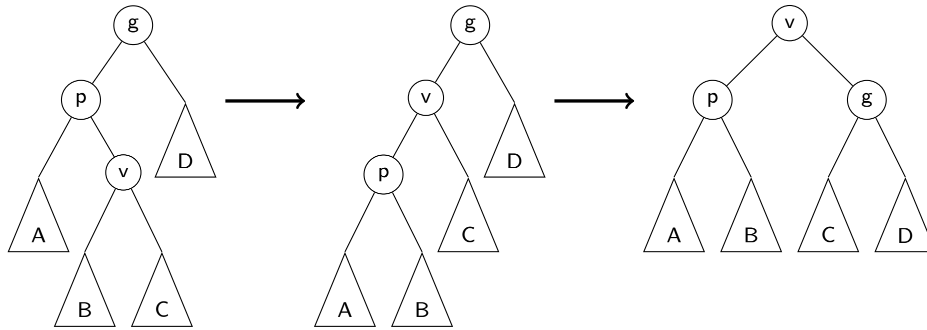
Все базовые операции со splay-деревом делаются так же, как и с обычным BST, но к ним добавляется одна новая операция: `splay(v)` перестраивает дерево так, что вершина v становится корнем дерева.

`splay(v)` последовательными вращениями (похожими на те, что используются для балансировки в AVL-дереве) проталкивает v к корню. На каждом шаге выполняется один из трёх типов вращений, в зависимости от того, как расположены друг относительно друга v, p — родитель v, и g — родитель p:

- **Zig.** Если p — корень дерева (то есть g не существует), то выполняется вращение вокруг ребра между v и p.

- **Zig-Zig.** Если v и p — оба левые или оба правые дети своих родителей, то выполняется вращение вокруг ребра между p и g, а затем вокруг ребра между v и p.

- **Zig-Zag.** Наконец, если v — левый ребёнок p, а p — правый ребёнок g, или наоборот, то выполняется вращение вокруг ребра между v и p, а затем вокруг ребра между v и g.



Операция splay(v) занимает $O(d)$ времени, где d — глубина вершины v.

В принципе, можно было бы просто выполнять вращение вокруг ребра между v и её родителем, пока v не окажется корнем (назовём такую операцию moveToRoot(v)). Единственный случай, в котором мы делаем не так — это случай Zig-Zig. Оказывается, что такой чуть более хитрый способ позволяет делать дерево "в среднем сбалансированным". Формально мы это докажем чуть позже, а пока можно посмотреть на следующие две картинки.

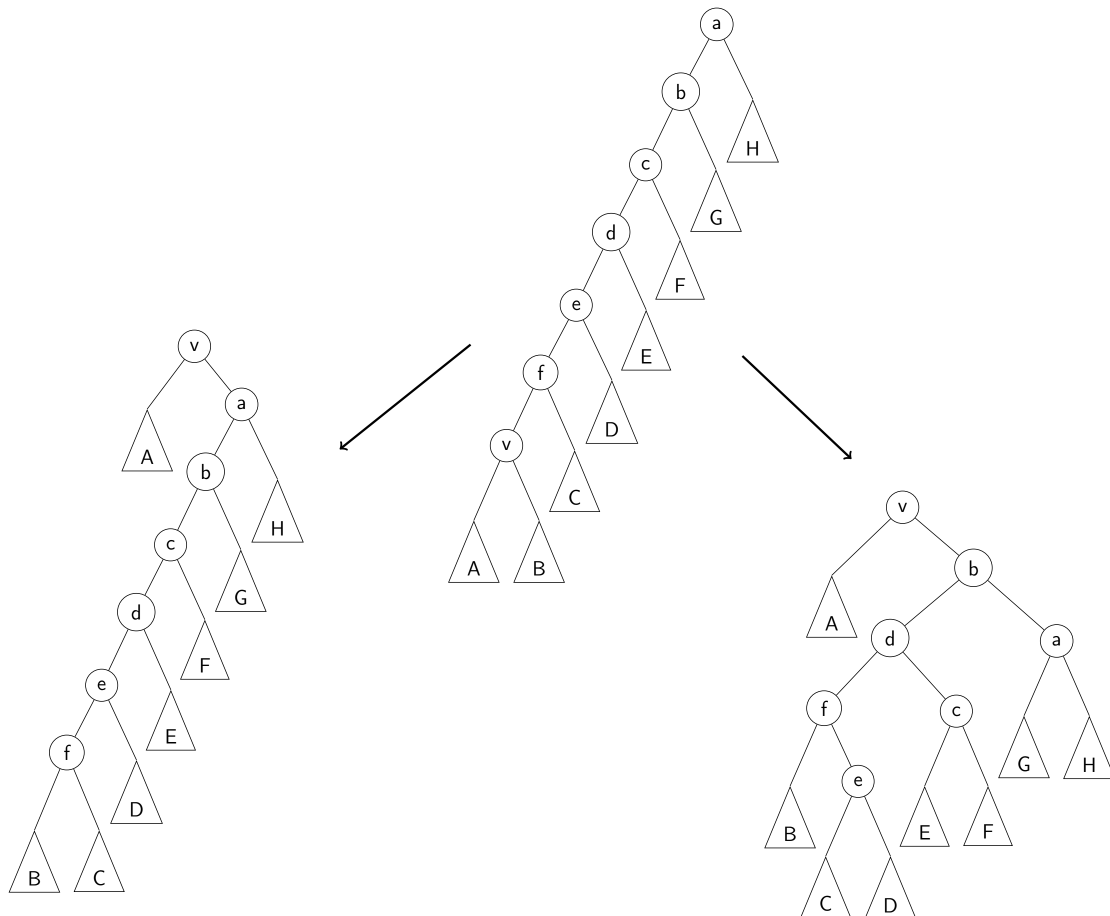


Рис. 12.1: Результат применения moveToRoot(v) (слева) и splay(v) (справа) к одному и тому же дереву.

12.2 Остальные операции

Базовые операции со splay-деревом делаются так же, как и с обычным двоичным деревом поиска, только после выполнения операции запускается `splay(v)`, где v — самая глубокая вершина, которая была посещена в течение операции. Поговорим о конкретных операциях подробнее:

- `find`, `lower_bound`, `upper_bound`, `next`, `prev` выполняются ровно так же, как и в обычном двоичном дереве поиска, после чего от найденной вершины (или от последней посещённой, если подходящей вершины не нашлось) запускается `splay`.
- `merge(l, r)`: вызовем `splay` от вершины, содержащей самый большой ключ в l. Эта вершина окажется корнем, причём у неё не будет правого ребёнка. Тогда просто подвесим к ней r правым ребёнком.
- `split(t, x)`: сделаем `splay(lower_bound(x))`, после чего отрежем левое поддерево от корня; оно окажется деревом l, а оставшееся дерево — деревом r.
- `insert` также можно делать, как в обычном двоичном дереве поиска, с последующим запуском `splay` от свежесозданной вершины. Альтернативный вариант — запустить `split` по ключу x, после чего создать новую вершину с ключом x, и подвесить результаты `split` к ней левым и правым поддеревом.
- `delete` снова можно делать, как обычно, с последующим запуском `splay` от родителя свежеудалённой вершины. Альтернативный вариант — запустить `splay` от вершины, которую надо удалить, после чего сделать `merge` её детей.

12.3 Реализация

Мы будем для удобства считать, что в каждой вершине поддерживается ссылка на родителя. Альтернативный вариант — при спуске по дереву выписывать текущий путь, и использовать в `splay` этот путь вместо ссылок на родителей. Существует так же так называемый top-down подход, в котором перестройка дерева осуществляется во время спуска (но мы о нём подробно говорить не будем). Заметим, что при использовании последних двух способов реализации в вершине не нужно хранить вообще никакой дополнительной информации, что выгодно отличает splay-дерево от других сбалансированных деревьев поиска.

Также заметим, что на splay-дерево легко адаптируются ранее изученные техники — операции по неявному ключу, вычисление ассоциативной функции на подотрезке и так далее. Проблемы возникают лишь с персистентностью: амортизированную оценку времени работы, которую мы вскоре докажем, не удастся перенести на операции с персистентным деревом.

Реализация операции `splay`:

```
1  rotate(v): # вращение вокруг ребра между v и её родителем
2    p = v->par # p - родитель v
3    if p->l == v:
4      p->l = v->r, v->r = p
5      if p->l != empty:
6        p->l->par = p
7    else:
8      p->r = v->l, v->l = p
9      if p->r != empty:
10       p->r->par = p
11   v->par = p->par, p->par = v
12   if v->par != empty:
13     if v->par->l == p:
14       v->par->l = v
15     else:
16       v->par->r = v
```

```
bigRotate(v):
  p = v->par
  if p->par != empty: # если p - не корень дерева
    if (p->l == v) == (p->par->l == p): # случай Zig-Zig
      rotate(p)
    else:                               # случай Zig-Zag
      rotate(v)
  rotate(v)

splay(v):
  while v->par != empty: # пока v - не корень дерева
    bigRotate(v)
```

12.4 Оценка времени работы

Если в дереве n вершин, то какие-то из вызовов splay могут иметь время работы $\Theta(n)$. Тем не менее, сейчас мы покажем, что среднее время работы splay — $O(\log n)$ с помощью амортизационного анализа. Воспользуемся методом потенциалов: пусть *ранг* вершины $r(v) = \log_2(s(v))$, где $s(v)$ — количество вершин в поддереве v, тогда *потенциалом* дерева назовём сумму рангов его вершин: $\phi = \sum_v r(v)$.

Для удобства будем считать, что простое вращение вокруг ребра занимает единицу реального времени. Пусть операция splay заняла t реального времени. Оценим *амортизированное время работы* операции $a = t + \Delta\phi$, где $\Delta\phi$ — изменение потенциала дерева в результате выполнения операции.

Предложение 12.4.1 Амортизированное время работы операции splay(v) в дереве с корнем t не превосходит $3(r(t) - r(v)) + 1 = O(\log(s(t)/s(v)) + 1) = O(\log n)$.

Доказательство. Сначала оценим изменение потенциала в результате выполнения какой-то одной итерации splay (то есть каждого типа вращения). Для понимания происходящего полезно смотреть на картинки с вращениями. В дальнейших рассуждениях будем считать, что r, r' и s, s' — это ранги и размеры поддеревьев до и после применения соответствующего вращения. Покажем, что изменение потенциала не превосходит $r'(v) - r(v)$ для Zig, $3(r'(v) - r(v)) - 2$ для Zig-Zig и $2(r'(v) - r(v)) - 2$ для Zig-Zag.

- **Zig.** Изменение потенциала равно $r'(p)+r'(v)-r(p)-r(v) = r'(p)-r(v) \leqslant r'(v)-r(v)$. Здесь пользуемся тем, что $r'(v) = r(p)$, $r'(p) \leqslant r'(v)$.
- **Zig-Zig.** Изменение потенциала равно $r'(v) + r'(p) + r'(g) - r(v) - r(p) - r(g) = r'(p) + r'(g) - r(v) - r(p) \leqslant r'(g) + r'(v) - 2r(v)$. Мы воспользовались тем, что $r'(v) = r(g)$, $r'(p) \leqslant r'(v)$, $r(p) \geqslant r(v)$.
  Мы хотим доказать, что $r'(g) + r'(v) - 2r(v) \leqslant 3(r'(v) - r(v)) - 2$, что равносильно $r'(g) + r(v) - 2r'(v) \leqslant -2$, то есть
  $$\log_2\left(\frac{s'(g)}{s'(v)}\right) + \log_2\left(\frac{s(v)}{s'(v)}\right) \leqslant -2.$$
  Обозначим $a = \frac{s'(g)}{s'(v)}$, $b = \frac{s(v)}{s'(v)}$. Заметим, что $s'(g) + s(v) + 1 = s'(v)$, то есть $a+b \leqslant 1$. По неравенству между средним арифметическим и средним геометрическим $ab \leqslant \left(\frac{a+b}{2}\right)^2 \leqslant \frac{1}{4}$. Тогда $\log_2 a + \log_2 b = \log_2(ab) \leqslant \log_2 \frac{1}{4} = -2$.
- **Zig-Zag.** Изменение потенциала равно $r'(v) + r'(p) + r'(g) - r(v) - r(p) - r(g) = r'(p) + r'(g) - r(v) - r(p) \leqslant r'(p) + r'(g) - 2r(v)$. Здесь мы пользуемся тем, что $r'(v) = r(g)$, $r(p) \geqslant r(v)$.

Аналогично случаю Zig-Zig, $r'(p) + r'(g) - 2r(v) \leqslant 2(r'(v) - r(v)) - 2$ равносильно $r'(p) + r'(g) - 2r'(v) \leqslant -2$, то есть

$$\log_2\left(\frac{s'(p)}{s'(v)}\right) + \log_2\left(\frac{s'(g)}{s'(v)}\right) \leqslant -2.$$

Обозначим $a = \frac{s'(p)}{s'(v)}$, $b = \frac{s'(g)}{s'(v)}$. Снова $a+b \leqslant 1$, поэтому работают те же рассуждения, что и в прошлом случае.

Zig состоит из одного вращения, поэтому амортизированное время работы Zig не превосходит $(r'(v) - r(v)) + 1 \leqslant 3(r'(v) - r(v)) + 1$. Zig-Zig и Zig-Zag состоят из двух вращений, поэтому амортизированное время их работы не превосходит $3(r'(v) - r(v))$.

Пусть операция `splay(v)` состояла из k итераций. Пусть r — ранг до начала выполнения операции, r_k — ранг после k-й итерации. Вспомним, что Zig мог произойти только на последней итерации. Тогда амортизированное время работы всех операций вместе не превосходит

$$3(r_1(v) - r(v)) + 3(r_2(v) - r_1(v)) + \cdots + 3(r_k(v) - r_{k-1}(v)) + 1 = 3(r_k(v) - r(v)) + 1.$$

Поскольку в конце v стала корнем, $r_k(v) = r(t)$. ∎

Теорема 12.4.2 Суммарное время работы m операций `splay` есть $O((m+n)\log n)$.

Доказательство. Пусть i-я операция заняла t_i реального времени. $a_i = t_i + (\phi_i - \phi_{i-1})$ — амортизированное время работы i-й операции.

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m}(a_i + \phi_{i-1} - \phi_i) = \sum_{i=1}^{m} a_i + \phi_0 - \phi_m.$$

По предложению 12.4.1, $\sum_{i=1}^{m} a_i$ не превосходит $O(m \log n)$. Осталось заметить, что $\phi_0 - \phi_m = O(n \log n)$, так как ранг любой вершины больше нуля и не больше $\log n$. ∎

Следствие 12.4.3 Пусть над несколькими splay-деревьями, общее количество вершин в которых равно n, было совершено m операций следующих типов: `find`, `lower_bound`, `upper_bound`, `next`, `prev`, `merge`, `split`, `insert`, `delete`. Тогда суммарное время работы этих операций есть $O((m+n)\log n)$.

Доказательство. Размер каждого из деревьев в любой момент времени не превосходит n.

Достаточно показать, что амортизированное время выполнения каждой из этих операций так же, как и для `splay`, не превосходит $O(\log n)$. Заметим, что реальное время выполнения каждой из операций оценивается так же, как и время внутреннего вызова `splay`. Остаётся оценить изменение потенциала в течение каждой операции.

Заметим, что ранги вершин (то есть и потенциал дерева) могут меняться **вне внутреннего вызова** `splay` только в операциях `merge`, `split`, `insert`, `delete`.

В `merge` и `split` меняется только ранг одной вершины (корня), значит изменение потенциала не превосходит $O(\log n)$.

Будем считать, что `insert` и `delete` реализованы через `split` и `merge`, тогда в них (помимо вызова `split/merge`) тоже меняется ранг лишь одной вершины (корня), также не больше, чем на $O(\log n)$ (для другого варианта реализации `insert` и `delete` утверждение тоже верно, но доказательство этого факта мы опустим). ∎

Элемент, к которому мы обращаемся, перемещается в корень, поэтому следующие обращения к нему выполнятся быстрее. По этой причине splay-деревья используют в менеджерах памяти, сборщиках мусора и тому подобных приложениях.

Теорема 12.4.4 — О статической оптимальности splay-дерева. Пусть было выполнено m запросов к splay-дереву, при этом i-й ключ был запрошен $q_i > 0$ раз. Тогда суммарное время выполнения запросов есть $O(m + \sum_i q_i \log(m/q_i))$.

Доказательство. Заметим, что если назначить каждой вершине положительный вес $w(v)$ и взять за $s(v)$ сумму весов вершин в поддереве, то все рассуждения в предложении 12.4.1 останутся корректными (кроме, возможно, равенства $O(\log(s(t)/s(v)) + 1) = O(\log n)$).

Возьмём в качестве весов вершин $w_i = q_i/m > 0$. Тогда амортизированное время запроса к вершине i (если t — корень дерева в момент запроса) по предложению 12.4.1 не превосходит $O(\log(s(t)/s(i)) + 1) = O(\log(1/w_i) + 1) = O(\log(m/q_i) + 1)$ (так как $s(t) = 1$, $s(i) \geqslant w_i$).

При этом $\phi_0 - \phi_m = O(\sum_i \log(m/q_i))$, так как ранг i-й вершины не больше нуля и не меньше $\log w_i = \log(q_i/m) = -\log(m/q_i)$.

Как и в прошлой теореме, $\sum_i t_i = \sum_i a_i + \phi_0 - \phi_m = O(m + \sum_i q_i \log(m/q_i))$. ■

Можно показать, что любое статическое (не меняющее структуру при выполнении запросов) BST затратит $\Omega(m + \sum_i q_i \log(m/q_i))$ времени на выполнение той же последовательности запросов (поэтому теорема называется теоремой о статической оптимальности).

Существует гипотеза, что splay-дерево оптимально в ещё более широком смысле.

Гипотеза 12.4.5 — О динамической оптимальности splay-дерева. Рассмотрим последовательность S запросов к элементам двоичного дерева поиска с n вершинами. Пусть A — некий алгоритм, который отвечает на запрос к элементу x, проходя путь от корня дерева к этому элементу за стоимость, равную глубине x плюс один. Кроме того, A может между запросами делать вращения вокруг произвольных рёбер в дереве, платя единицу за каждое вращение. Пусть $A(S)$ — стоимость, за которую A выполняет последовательность запросов S. Тогда splay-дерево выполняет ту же последовательность запросов за $O(n + A(S))$.

Поговорим и о недостатках splay-деревьев. Во-первых, время выполнения одной конкретной операции может оказаться линейным, поэтому splay-деревья плохо подходят для применения в ситуациях, где важно, чтобы **каждая** операция выполнялась быстро. Во-вторых, структура splay-дерева меняется даже при запросах на чтение. Это, например, усложняет многопоточное использование splay-деревьев.

13. Skip-list

Как мы уже знаем, с помощью двоичных деревьев поиска можно поддерживать упорядоченное множество из n элементов, а также совершать над ним операции `find`, `insert`, `delete`, `split`, `merge` со сложностью $O(\log n)$ каждая. *Skip-list* (Pugh, 1989) не является деревом поиска, но позволяет делать всё то же самое.

13.1 Основная идея

Вспомним, что обычные односвязные списки позволяют выполнять все вышеперечисленные операции, кроме `find`, за сложность выполнения `find` плюс $O(1)$ (если позиция, в которой нужно произвести действие, уже найдена, то остаётся поменять константное число ссылок). Проблема в том, что `find` имеет сложность $O(n)$.

Для того, чтобы ускорить `find`, хочется научиться перемещаться по списку сразу на много позиций вперёд за одну операцию. Например, можно было бы для каждого элемента хранить ссылки не только на следующий элемент, но и на элементы, находящиеся на расстоянии 2^k вперёд для $k = 1, \ldots, \log n$. Тогда поиск x можно было бы осуществлять за $O(\log n)$, каждый раз делая прыжок максимальной длины, всё ещё ведущий в элемент, меньше или равный x. Проблема этого подхода в том, что при изменении списка придётся пересчитать сразу много таких ссылок.

Идея skip-list'а очень похожа. Давайте хранить $\lceil \log n \rceil$ списков. В нулевом списке будут лежать все элементы. Каждый элемент $(i-1)$-го списка положим в i-й список с вероятностью $\frac{1}{2}$. Тогда математическое ожидание размера i-го списка равняется $\frac{n}{2^i}$, а математическое ожидание суммарного размера всех списков равно

$$n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) \leqslant 2n = O(n).$$

`find(x)` будем осуществлять следующим образом: будем двигаться вперёд по самому верхнему l-му списку ($l = \lceil \log n \rceil - 1$), пока не окажемся в его последнем элементе, или пока следующий элемент не окажется больше или равен x. Элемент, в котором мы окажемся в этот момент, есть в l-м списке, тогда он есть и в $(l-1)$-м списке. Продолжим двигаться вперёд из того же элемента, но уже по $(l-1)$-му списку. В тот момент, когда мы дойдём до конца списка, или следующий элемент окажется больше или равен x, опустимся ещё на уровень ниже и так далее.

В конце мы окажемся в нулевом списке в таком элементе, что он сам меньше x, а следующий за ним уже больше или равен x. Тогда для того, чтобы понять, если ли x в нашей последовательности, достаточно проверить значение следующего элемента.

Чуть позже мы покажем, что время работы такого поиска в среднем не превосходит $O(\log n)$.

Остальные операции выполняются очень просто: ищем с помощью `find` позицию, в которой нужно выполнить операцию, после чего выполняем её в каждом из $\lceil \log n \rceil$ списков за $O(1)$, то есть суммарно за $O(\log n)$.

13.2 Реализация

Удобно хранить все списки вместе: для элемента, лежащего в списках с номерами $0, 1, \ldots, k$ будем хранить $k + 1$ ссылку на следующие элементы в соответствующих списках. Также удобно сделать фиктивный головной элемент, который будет лежать во всех списках.

```
class Node:
  vector<Node *> next # next[i] - ссылка на следующий элемент в i-м списке
  int x

class List:
  Node *head # фиктивный головной элемент
  int lvlCnt # количество списков

  init(_lvlCnt): # можно запускать с _lvlCnt = ceil(log(maxN))
    head = new Node()
    lvlCnt = _lvlCnt
    for i = 0..(lvlCnt - 1):
      head->next.push_back(NULL)

  find(x):
    v = head
    for i = (lvlCnt - 1)..0:
      while v->next[i] and v->next[i]->x < x:
        v = v->next[i]
    # сейчас v->x - максимальный, меньший x
    v = v->next[0]
    # теперь v->x - минимальный, больше или равный x
    if v and v->x == x:
      return v
    return NULL

  insert(x):
    vector<Node *> upd(lvlCnt)
    # запомним на каждом уровне максимальный элемент, меньший x
    v = head
    for i = (lvlCnt - 1)..0:
      while v->next[i] and v->next[i]->x < x:
        v = v->next[i]
      upd[i] = v
    v = v->next[0]
    if v and v->x == x:
      return # x уже есть в списке
    w = new Node()
    w->x = x
    for i = 0..(lvlCnt - 1):
      w->next.push_back(upd[i]->next[i])
      upd[i]->next[i] = w
      if rand() % 2: # с вероятностью 1/2 не кладём w в следующий список
        break

  delete(x):
    ... # находим upd и v так же, как и в insert
    # теперь v->x - минимальный, больше или равный x
    if !v or v->x != x:
      return # x в списке нет
    for i = 0..(lvlCnt - 1):
      if upd[i]->next[i] != v:
        break
      upd[i]->next[i] = v->next[i]
    delete v
```

```
split(l, x):
  ... # находим upd так же, как и в insert
  r = new List()
  r.init(l.lvlCnt)
  for i = 0..(l.lvlCnt - 1):
    r.head->next[i] = upd[i]->next[i]
    upd[i]->next[i] = NULL
  return l, r

merge(l, r): # считаем, что lvlCnt у l и r одинаковый
  v = l.head
  for i = (l.lvlCnt - 1)..0:
    while v->next[i]:
      v = v->next[i]
    # v - максимальный в l элемент в i-м списке
    v->next[i] = r.head->next[i]
  delete r
  return l
```

Можно выполнять и операции по неявному ключу. Для этого нужно для каждой ссылки хранить её длину: на сколько позиций вперёд она указывает в полном списке. Все ссылки из фиктивной головы списка можно считать имеющими длину один. Приведём для примера реализации `find` и `delete` по неявному ключу:

```
class Node:
  vector<Node *> next # next[i] - ссылка на следующий элемент в i-м списке
  vector<int> len # len[i] - длина ссылки в i-м списке
  int x

class List:
  Node *head # фиктивный головной элемент
  int lvlCnt # количество списков

  Node *find(k): # найти k-ый элемент (нумерация с нуля)
    v = head
    k += 1 # учтём длину ссылки из head
    for i = (lvlCnt - 1)..0:
      while v->next[i] and v->len[i] < k:
        k -= v->len[i]
        v = v->next[i]
    # v - элемент перед k-м либо последний, если k-го нет
    return v->next[0]

  delete(k): # удалить k-й элемент (нумерация с нуля)
    vector<Node *> upd(lvlCnt) # найдём на каждом уровне элемент перед k-м
    v = head
    k += 1 # учтём длину ссылки из head
    for i = (lvlCnt - 1)..0:
      while v->next[i] and v->len[i] < k:
        k -= v->len[i]
        v = v->next[i]
      upd[i] = v
    v = v->next[0]
    if !v:
      return # k-го элемента нет
    for i = 0..(lvlCnt - 1):
      if upd[i]->next[i] == v:
        upd[i]->len[i] += v->len[i]
        upd[i]->next[i] = v->next[i]
      upd[i]->len[i] -= 1
    delete v
```

13.3 Оценка времени работы

Теорема 13.3.1 Математическое ожидание времени работы `find` — $O(\log n)$.

Доказательство. Пусть результат поиска — элемент v. В каждом списке мы остановились на последнем элементе, находящемся левее v в нулевом списке. Посмотрим на процесс поиска с конца. Получается, что мы идём по спискам влево, начиная из v. Мы движемся по нулевому списку, пока не придём в элемент, который лежит не только в нулевом, но и в первом списке. Из него мы движемся влево по первому списку, пока не придём в элемент, лежащий во втором списке, и так далее, пока не придём в фиктивную головную вершину.

Поскольку для любого элемента в i-м списке вероятность того, что он попал в $(i+1)$-й список, равна $\frac{1}{2}$, математическое ожидание количества вершин, посещённых во время движения по любому списку, кроме самого верхнего, равно

$$1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \cdots\right)\right)\right) = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \leqslant 2.$$

По самому верхнему списку мы пройдём не больше шагов, чем его размер, математическое ожидание которого равно $\frac{n}{2^{\lceil \log n \rceil - 1}} = O(1)$.

Таким образом на каждом уровне мы совершим в среднем $O(1)$ действий, всего уровней $O(\log n)$, что завершает доказательство. ■

Skip-list'ы быстро работают на практике и относительно просто пишутся. Кроме того, их удобно использовать в параллельных вычислениях, например, одновременно вставляя элементы в разные части skip-list'а. По этим причинам skip-list'ы часто используются в базах данных (например, в MemSQL).

14. RMQ и LCA

В задаче RMQ (Range Minimum Query) дан массив $a_0, \ldots, a_{n-1}$, и требуется отвечать на запросы нахождения минимума на подотрезке массива: $RMQ_a(l, r) = \min_{l \leqslant i < r} a_i$. Можно рассматривать *статическую* и *динамическую* версии задачи: в статической версии массив не меняется, в динамической же бывают также запросы на изменение элемента.

Введём следующее обозначение: если структура данных требует $p(n)$ времени на предварительную обработку данных (предподсчёт), после чего позволяет отвечать на каждый запрос за время $q(n)$, то будем кратко обозначать время её работы как $\langle p(n), q(n) \rangle$.

Мы уже умеем решать динамическую версию задачи за время $\langle O(n), O(\log n) \rangle$, например, при помощи дерева отрезков или сбалансированного дерева поиска. Далее мы сфокусируемся на статической версии задачи.

14.1 Разреженная таблица

Пусть мы хотим отвечать на запросы за $O(1)$. Самый простой способ сделать это — предподсчитать ответы на все возможные запросы заранее; время работы такого способа составит $\langle O(n^2), O(1) \rangle$ (при этом потребуется использовать $O(n^2)$ памяти).

Заметим, однако, что для того, чтобы уметь отвечать на любой запрос за $O(1)$, совсем не обязательно предподсчитывать ответы на **все** запросы. Для любых $l < r$ выполняется равенство $[l, r) = [l, l + 2^k) \cup [r - 2^k, r)$, где $2^k \leqslant r - l < 2^{k+1}$. Следовательно,

$$RMQ_a(l, r) = \min(RMQ_a(l, l + 2^k), RMQ_a(r - 2^k, r)).$$

Таким образом, достаточно предподсчитать ответы лишь на запросы $RMQ_a(l, r)$, длина отрезка в которых $r - l$ есть степень двойки; а также для каждой возможной длины отрезка предподсчитать соответствующее ей k. Получается структура данных со временем работы $\langle O(n \log n), O(1) \rangle$; её называют *разреженной таблицей (sparse table)*.

```
1  # 2 ** log[i] <= i < 2 ** (log[i] + 1)
2  log[1] = 0
3  for i = 2..n:
4    log[i] = log[i - 1]
5    if (1 << (log[i] + 1)) <= i:
6      log[i] += 1
7
8  # в rmq[k][i] запишем ответ на запрос RMQ_a(i, i + 2 ** k)
9  for i = 0..(n - 1):
10   rmq[0][i] = a[i]
11 for k = 0..(log[n] - 1):
12   for i = 0..(n - (1 << (k + 1))):
13     rmq[k + 1][i] = min(rmq[k][i], rmq[k][i + (1 << k)])
14
15 getRMQ(l, r):
16   k = log[r - l]
17   return min(rmq[k][l], rmq[k][r - (1 << k)])
```

14.2 Блочная оптимизация

Поделим массив на блоки длины k (конкретное значение k выберем позже); если надо, добавим элементов со значением $+\infty$ в конец, чтобы массив поделился на целое число блоков. Любой отрезок массива разбивается на несколько подряд идущих блоков, а также на не более чем два подотрезка (слева и справа), каждый из которых содержится в одном из блоков. Предподсчитаем минимум на каждом блоке и выпишем эти минимумы в отдельный массив b. Построим на этом массиве, а также на каждом блоке, отдельную структуру данных, решающую задачу RMQ. Тогда запрос на исходном массиве сводится к одному запросу на массиве b, и не более чем двум запросам на блоках.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	4	2	7	6	9	3	8	5	6	9	10	14	7	3	5
b		2			3			5			9			3	

Рис. 14.1: Разбиение массива на блоки длины $k = 3$. Синим цветом отмечено разбиение на части запроса на отрезке $[2, 11)$; он сводится к запросу $[1, 3)$ на массиве b, и двум запросам на подотрезках к 0-му и 3-му блокам.

Пусть структура данных на массиве b имеет время работы $\langle p_1(n), q_1(n)\rangle$, а структура данных на каждом блоке — $\langle p_2(n), q_2(n)\rangle$. Тогда время работы построенной нами структуры данных есть

$$\langle O(n + p_1(n/k) + (n/k) \cdot p_2(k)), O(q_1(n/k) + q_2(k))\rangle.$$

Посмотрим, каких оценок на время работы мы сможем добиться, подставляя различные структуры для массива b и блоков:

- Можно не строить никаких структур, а просто отвечать на каждый запрос за линейное время:
  $$\langle p_1(n), q_1(n)\rangle = \langle p_2(n), q_2(n)\rangle = \langle O(1), O(n)\rangle.$$
  Получаем оценку на время работы блочной структуры
  $$\langle O(n + n/k), O(n/k + k)\rangle = \langle O(n), O(n/k + k)\rangle;$$
  оценка на время выполнения запроса оптимальна при $k = \sqrt{n}$. Получаем структуру данных со временем работы $\langle O(n), O(\sqrt{n})\rangle$.
- Теперь построим на массиве b разреженную таблицу, а на запросы в блоках по-прежнему будем отвечать за линейное время:
  $$\langle p_1(n), q_1(n)\rangle = \langle O(n \log n), O(1)\rangle, \quad \langle p_2(n), q_2(n)\rangle = \langle O(1), O(n)\rangle.$$
  Тогда время работы всей структуры при $k = \log n$ оценивается как
  $$\langle O(n + (n/\log n) \cdot \log(n/\log n) + n/\log n), O(1 + \log n)\rangle = \langle O(n), O(\log n)\rangle.$$
- Теперь используем разреженную таблицу и для массива b, и для блоков:
  $$\langle p_1(n), q_1(n)\rangle = \langle p_2(n), q_2(n)\rangle = \langle O(n \log n), O(1)\rangle.$$
  При размере блока $k = \log n$ получаем
  $$\langle O(n + (n/\log n) \cdot \log(n/\log n) + (n/\log n) \cdot (\log n \log\log n)), O(1)\rangle =$$
  $$\langle O(n \log\log n), O(1)\rangle.$$

- Будем использовать разреженную таблицу для массива b, и дерево отрезков (или одну из блочных структур, описанных выше) для блоков:

$$\langle p_1(n), q_1(n)\rangle = \langle O(n\log n), O(1)\rangle, \quad \langle p_2(n), q_2(n)\rangle = \langle O(n), O(\log n)\rangle.$$

  При размере блока $k = \log n$ получаем время работы

$$\langle O(n + (n/\log n)\cdot\log(n/\log n) + (n/\log n)\cdot\log n), O(1+\log\log n)\rangle =$$

$$\langle O(n), O(\log\log n)\rangle.$$

Можно ли таким способом добиться оптимального времени работы: $\langle O(n), O(1)\rangle$? На первый взгляд кажется, что нет: из $(n/k)\cdot p_2(k) = O(n)$ и $q_2(k) = O(1)$ сразу следует $p_2(n) = O(n)$, $q_2(n) = O(1)$; то есть для того, чтобы получить оптимальную структуру таким способом, нужно её уже иметь.

14.3 Конструкция Фишера-Хойна

Оказывается, оптимального времени работы добиться всё-таки можно. С этого момента будем в качестве ответа на запрос требовать не минимальное значение на отрезке, а индекс минимального элемента на отрезке (по индексу элемента несложно восстановить его значение). Также будем считать, что все элементы массива попарно различны (всегда можно вместо исходных элементов работать с парами (a_i, i), которые точно попарно различны; на оценку времени работы это никак не повлияет).

Конструкция, которую мы изучим (Fischer, Heun, 2006), использует следующее наблюдение: иногда для ответа на запросы в разных блоках можно использовать одну и ту же структуру данных; будем называть такие блоки "одинаковыми".

Рис. 14.2: Индекс минимального элемента на любом подотрезке этих двух блоков совпадает.

Как по двум блокам понять, "одинаковые" ли они? Заметим, что минимальные элементы в таких блоках должны находиться на одной и той же позиции. Кроме того, то же свойство должно рекурсивно выполняться для подотрезков блоков слева и справа от минимума. Мы уже сталкивались с подобными рассуждениями, когда изучали декартовы деревья! Мы только что описали рекурсивное построение декартова дерева на каждом блоке с использованием индексов как ключей, а значений элементов как приоритетов. При этом получается, что блоки "одинаковы" тогда и только тогда, когда построенные по ним декартовы деревья изоморфны (то есть совпадают как корневые двоичные деревья).

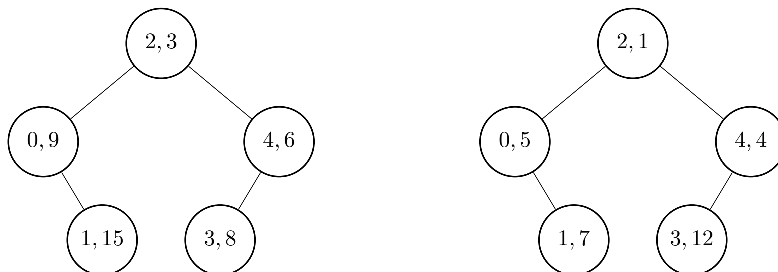


Рис. 14.3: Декартовы деревья, построенные на массивах из рис. 14.2.

Как быстро понять, изоморфны ли два декартовых дерева? Вспомним, что мы умеем строить декартово дерево за линейное время алгоритмом со стеком (в этом алгоритме

требовалось, чтобы ключи уже были упорядочены по возрастанию, что в нашем случае выполняется). Будем строить дерево для каждого блока таким алгоритмом, выписывая в отдельную последовательность 1, когда кладём новую вершину в стек, и 0, когда достаём из стека какую-либо вершину. Например, каждому из деревьев на рис. 14.3 соответствует последовательность бит 11001101. Дополним получившуюся последовательность нулями до длины $2k$, где k — размер дерева (это соответствует удалению всех оставшихся вершин из стека в конце работы алгоритма). При этом получится правильная скобочная последовательность (если 1 считать открывающей скобкой, а 0 — закрывающей.)

Декартовы деревья изоморфны тогда и только тогда, когда соответствующие им последовательности бит совпадают. Действительно, последовательности бит для двух деревьев совпадают тогда и только тогда, когда при построении деревьев для любого i вершина с ключом i вставлялась на одну и ту же позицию в обоих деревьях.

Итак, теперь мы знаем, как проверить, являются ли два блока "одинаковыми" за $O(k)$, где k — размер блоков. Можно даже не строить явно декартово дерево: будем проходить по каждому блоку со стеком; извлекать из стека все элементы, большие текущего, после чего класть текущий элемент в стек; в конце извлечём из стека все оставшиеся элементы. При этом будем выписывать в отдельную последовательность 1 при вставке и 0 при извлечении. Интерпретируем эту последовательность бит как число, записанное в двоичной системе счисления; будем называть это число *декартовым номером* блока. Тогда блоки "одинаковы", если их декартовы номера равны.

Пусть $k = \lceil (\log_2 n)/4 \rceil$, тогда декартов номер любого блока есть $O(2^{2k}) = O(\sqrt{n})$. Посчитаем декартов номер для каждого блока; если такой номер ещё не встречался (это можно проверить с помощью отдельного массива размером $O(\sqrt{n})$), то просто предподсчитаем ответы на все возможные запросы для этого блока; это потребует $O(k^2)$ времени и памяти. Если же такой номер уже встречался, то переиспользуем уже предподсчитанные ответы на запросы. Поскольку всего различных декартовых номеров $O(\sqrt{n})$, нам понадобится $O(n + \sqrt{n}k^2) = O(n + \sqrt{n}\log^2 n) = O(n)$ времени и памяти на предварительную обработку блоков.

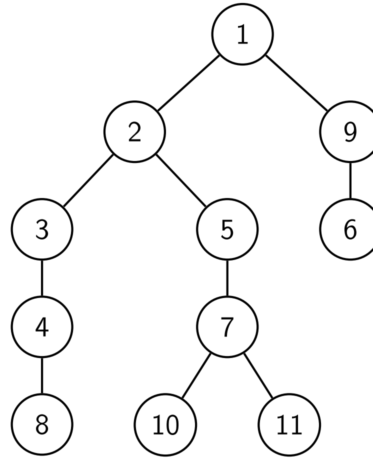
Кроме того, построим разреженную таблицу на массиве b, для чего потребуется $O((n/k)\log(n/k)) = O(n/\log n \cdot \log n) = O(n)$ времени и памяти. На каждый запрос мы теперь можем ответить за $O(1)$. Итак, мы получили структуру, решающую задачу RMQ за $\langle O(n), O(1) \rangle$.

R Идею деления на маленькие блоки и предобработки всех возможных блоков такого размера называют *методом четырёх русских*; применение этого метода часто позволяет уменьшить степень вхождения логарифма в оценку времени работы.

14.4 LCA

В задаче LCA (Lowest Common Ancestor, наименьший (самый глубокий) общий предок) дано корневое дерево, и требуется по паре вершин a, b находить $LCA(a, b)$ — самую глубокую вершину, являющуюся общим предком a и b. Мы научимся сводить эту задачу к задаче RMQ за линейное время; таким образом, все уже изученные структуры данных (и их оценки времени работы) для задачи RMQ применимы и к этой задаче.

Выпишем *эйлеров обход (Euler tour representation)* дерева: будем обходить его алгоритмом поиска в глубину, начиная из корня; при этом каждый раз, когда мы попадаем в вершину (в том числе, когда мы возвращаемся из её потомка), будем выписывать вершину и её глубину. Кроме того, для каждой вершины v запомним позицию pos_v любого (например, первого) её вхождения в полученную последовательность.



1	2	3	4	8	4	3	2	5	7	10	7	11	7	5	2	1	9	6	9	1
0	1	2	3	4	3	2	1	2	3	4	3	4	3	2	1	0	1	2	1	0

Рис. 14.4: Дерево и его эйлеров обход; синим цветом отмечены первые вхождения каждой вершины.

Чем полезен эйлеров обход? Заметим, что $LCA(a,b)$ — это вершина с минимальной высотой в подотрезке эйлерова обхода $[\min(pos_a, pos_b), \max(pos_a, pos_b)]$. Почему это так? Пусть $c = LCA(a,b)$, тогда все вершины в описанном подотрезке эйлерова обхода лежат в поддереве c. В частности, все вершины в подотрезке, кроме c, имеют глубину больше c. Остаётся показать, что c точно встретится в подотрезке. Если $c = a$ или $c = b$, то доказывать нечего. Если же $c \neq a, b$, то a и b находятся в поддеревьях разных детей c, значит, между любыми вхождениями a и b найдётся вхождение c.

Итак, для того, чтобы ответить на запрос LCA, достаточно ответить на запрос RMQ на эйлеровом обходе дерева.

R Заметим, что соседние высоты в эйлеровом обходе отличаются не более, чем на единицу. Таким образом, на самом деле мы свели задачу LCA к упрощённой версии задачи RMQ; её называют задачей RMQ$\pm$1. При решении такой задачи не обязательно вычислять декартовы номера блоков: можно считать, что нулевой элемент блока равен нулю (вычтем из всех элементов блока значение нулевого элемента); тогда каждому блоку можно сопоставить номер из $k-1$ бита (i бит равен 1, если $(i+1)$-й элемент блока на единицу больше предыдущего, и равен 0, если на единицу меньше). Этот метод называют алгоритмом Фарах-Колтона и Бендера (Farach-Colton, Bender, 2000).

Задачу RMQ можно свести к задаче LCA (минимум на отрезке — это запрос LCA на декартовом дереве, построенном на индексах-ключах и приоритетах-значениях массива) и решать этим методом. Изученная нами конструкция Фишера-Хойна является в каком-то смысле упрощением такого способа.

14.5 Алгоритм Тарьяна

Рассмотрим offline-версию задачи LCA: дано корневое дерево на n вершинах и m запросов LCA; при этом все запросы известны заранее. Существует простой алгоритм (Tarjan, 1979), решающий задачу за $O((n+m)\alpha(n))$, где α — обратная функция Аккермана.

Для каждой вершины v выпишем в список все содержащие её запросы. Будем обходить дерево поиском в глубину. При этом будем поддерживать на вершинах структуру DSU:

вначале каждая вершина находится в своём множестве; при выходе из вершины будем объединять множество вершины и множество её родителя. Также для каждого множества будем поддерживать вершину с минимальной глубиной в этом множестве (она всегда единственна; номер такой вершины несложно поддерживать в корне каждого множества).

В момент, когда мы приходим в вершину v, посмотрим на все содержащие её запросы; если вторая вершина в запросе u уже была посещена, то $LCA(u, v)$ — это вершина с минимальной глубиной в множестве вершины u. Действительно, если u ещё не закончила обрабатываться, то u — предок v, то есть $LCA(u, v) = u$; при этом u имеет минимальную глубину в своём множестве. Если же u уже закончила обрабатываться, то сейчас она лежит в множестве какой-то из вершин на пути из корня в v; обозначим эту вершину за c. Вершина c всё ещё обрабатывается, значит, она имеет минимальную глубину в своём множестве. Кроме того, u и v находятся в поддеревьях разных детей c, так как u уже закончила обрабатываться, а v ещё нет. Значит, $c = LCA(u, v)$.

Получаем очень короткое решение, состоящее из поиска в глубину и структуры DSU.

```
# q[v] - список номеров запросов, содержащих v
# i-й запрос - LCA(a[i], b[i]); ответ запишем в res[i]
# dsu.getMin(v) возвращает вершину с минимальной глубиной в множестве вершины v

dfs(v):
  used[v] = True
  for i in q[v]:
    u = (b[i] if a[i] == v else a[i])
    if used[u]:
      res[i] = dsu.getMin(u)
  for u in to[v]:
    if not used[u]:
      dfs(u)
      dsu.join(v, u)
```

R Поскольку в данном случае заранее известно, какие объединения и в каком порядке будут происходить, можно реализовать DSU так, что алгоритм будет работать за $O(n + m)$ (Tarjan, Gabow, 1983).

Offline-версия задачи RMQ

Поскольку задачу RMQ можно свести к задаче LCA построением декартова дерева, offline-версию задачи RMQ также можно решать алгоритмом Тарьяна. При этом обход дерева можно производить одновременно с его построением; более того, явно строить декартово дерево необязательно. Опишем полученный таким образом алгоритм в терминах исходной задачи: будем отвечать на запросы $RMQ_a(l, r)$ в порядке увеличения r. При этом будем поддерживать стек минимумов на префиксе массива $a[0, r)$: пусть в стеке s лежат такие $m_1 < m_2 < \cdots < m_k$, что $a[m_1] < a[m_2] < \cdots < a[m_k]$; $a[m_i]$ — минимум на $a(m_{i-1}, r)$ (считаем $m_0 = -1$). Также будем при помощи DSU поддерживать множества $(m_{i-1}, m_i]$. Тогда ответ на запрос $RMQ_a(l, r)$ — это такое m_i, что $l \in (m_{i-1}, m_i]$.

```
# q[i] - список номеров запросов с правой границей i
# i-й запрос - RMQ на отрезке [l[i], r[i]); ответ запишем в res[i]
# dsu.getMin(i) возвращает минимум в множестве, содержащем i
for i = 1..n:
  for j in q[i]:
    res[j] = dsu.getMin(l[j])
  while s.size() > 0 and a[s.top()] >= a[i]:
    j = s.pop()
    dsu.join(j, i)
  s.push(i)
```