SPb HSE, 1 курс магистратуры, осень 2021/22
Конспект лекций по алгоритмам

Собрано 20 декабря 2021 г. в 11:14

Содержание

Лекция #0: Технические замечания

Сентябрь 2021

См. также основной файл про технические моменты.

0.1. C++

• Warnings

1. Сделайте, чтобы компилятор `g++/clang` отображал вам как можно больше warning-ов:

```
-Wall -Wextra -Wshadow
```

2. Пишите код, чтобы при компиляции не было warning-ов.

• Range check errors

Давайте рассмотрим стандартную багу: `int a[3]; a[3] = 7;`
В результате мы получаем *undefined behavior*. Код иногда падает по *runtime error*, иногда нет. Чтобы такого не было, во-первых, используйте вектора, во-вторых, включите debug-режим (но при отправке в проверяющую систему его лучше убрать).

```
#define _GLIBCXX_DEBUG // должна быть до всех #include
vector<int> a(3);
a[3] = 7; // Runtime Error!
```

Для пользователей linux есть более профессиональное решение: valgrind.

0.2. Неасимптотические оптимизации

При написании программы, если хочется, чтобы она работала быстро, стоит обращать внимание не только на асимптотику, но и избегать использования некоторых операций, которые работают дольше, чем кажется.

1. Доступ к памяти. Существует два способа прохода по массиву:
   Random access: `for (i = 0; i < n; i++) sum += a[p[i]];` здесь p – случайная перестановка
   Sequential access: `for (i = 0; i < n; i++) sum += a[i];`
2. Вызов функций. Пример, который при $n = 10^7$ работает секунду и использует $\geqslant 320$ `mb`.

```
void go( int n ) {
  if (n <= 0) return;
  go(n - 1); // компилируйте с -O0, чтобы оптимизатор не раскрыл хвостовую рекурсию в цикл
}
```

   Для оптимизации можно использовать `inline` – указание оптимизатору, что функцию следует не вызывать, а попытаться вставить в код.

• История про кеш

В нашем распоряжении есть примерно такие объёмы

1. Жёсткий диск. Самая медленная память, 1 терабайт.

2. Оперативная память. Средняя, 8 гигабайта.
3. Кеш L3. Быстрая, 4 мегабайта.
4. Кеш L1. Сверхбыстрая, 32 килобайта.

Отсюда вывод. Если у нас есть два алгоритма $\langle T_1, M_1\rangle$ и $\langle T_2, M_2\rangle$: $T_1 = T_2 = \mathcal{O}(n^2); M_1 = \mathcal{O}(n^2); M_2 = \Theta(n)$, то второй алгоритм будет работать быстрее для больших значений n, так как у первого будут постоянные промахи мимо кеша.

И ещё один. Если у нас есть два алгоритма $\langle T_1, M_1\rangle$ и $\langle T_2, M_2\rangle$: $T_1 = T_2 = \Theta(2^n); M_1 = \Theta(2^n); M_2 = \Theta(n^2)$, То первый в принципе не будет работать при $n \approx 40$, ему не хватит памяти. Второй же при больших $n \approx 40$ неспешно, за несколько часов, но отработает.

• Быстрые операции

`memcpy(a, b, n)` (скопировать n байт памяти), `strcmp(s, t)` (сравить строки).
Работают в 8 раз быстрее цикла `for` за счёт 128-битных SSE и 256-битных AVX регистров!

Лекция #1: Асимптотика

Сентябрь 2021

1.1. $\mathcal{O}$-обозначния

Рассмотрим функции $f, g\colon \mathbb{N} \to \mathbb{R}^{>0}$.

Def 1.1.1. $f = \Theta(g) \quad \exists N > 0,\, C_1 > 0, C_2 > 0\colon \forall n \geqslant N, C_1 \cdot g(n) \leqslant f(n) \leqslant C_2 \cdot g(n)$

Def 1.1.2. $f = \mathcal{O}(g) \quad \exists N > 0,\, C > 0\colon \forall n \geqslant N, f(n) \leqslant C \cdot g(n)$

Def 1.1.3. $f = \Omega(g) \quad \exists N > 0,\, C > 0\colon \forall n \geqslant N, f(n) \geqslant C \cdot g(n)$

Def 1.1.4. $f = o(g) \quad \forall C > 0\ \exists N > 0\colon \forall n \geqslant N, f(n) \leqslant C \cdot g(n)$

Def 1.1.5. $f = \omega(g) \quad \forall C > 0\ \exists N > 0\colon \forall n \geqslant N, f(n) \geqslant C \cdot g(n)$

Понимание Θ: "равны с точностью до константы", "асимптотически равны".
Понимание $\mathcal{O}$: "не больше с точностью до константы", "асимптотически не больше"
Понимание o: "асимптотически меньше", "для сколь угодно малой константы не больше"

Θ	$\mathcal{O}$	Ω	o	ω
$=$	$\leqslant$	$\geqslant$	$<$	$>$

Замечание 1.1.6. $f = \Theta(g) \Leftrightarrow g = \Theta(f)$

Замечание 1.1.7. $f = \mathcal{O}(g), g = \mathcal{O}(f) \Leftrightarrow f = \Theta(g)$

Замечание 1.1.8. $f = \Omega(g) \Leftrightarrow g = \mathcal{O}(f)$

Замечание 1.1.9. $f = \omega(g) \Leftrightarrow g = o(f)$

Замечание 1.1.10. $f = \mathcal{O}(g), g = \mathcal{O}(h) \Rightarrow f = \mathcal{O}(h)$

Замечание 1.1.11. Обобщение: $\forall \beta \in \{\mathcal{O}, o, \Theta, \Omega, \omega\}\colon f = \beta(g), g = \beta(h) \Rightarrow \boxed{f = \beta(h)}$

Замечание 1.1.12. $\forall C > 0 \quad C \cdot f = \Theta(f)$

Докажем для примера 1.1.6.

Доказательство. $C_1 \cdot g(n) \leqslant f(n) \leqslant C_2 \cdot g(n) \Rightarrow \frac{1}{C_2} f(n) \leqslant g(n) \leqslant \frac{1}{C_1} g(n) \leqslant f(n)$ ■

Упражнение 1.1.13. $f = \mathcal{O}(\Theta(\mathcal{O}(g))) \Rightarrow f = \mathcal{O}(g)$

Упражнение 1.1.14. $f = \Theta(o(\Theta(\mathcal{O}(g)))) \Rightarrow f = o(g)$

Упражнение 1.1.15. $f = \Omega(\omega(\Theta(g))) \Rightarrow f = \omega(g)$

Упражнение 1.1.16. $f = \Omega(\Theta(\mathcal{O}(g))) \Rightarrow f$ может быть любой функцией

Lm 1.1.17. $g = o(f) \Rightarrow f \pm g = \Theta(f)$

Доказательство. $g = o(f)\ \exists N\colon \forall n \geqslant N\ g(n) \leqslant \frac{1}{2} f(n) \Rightarrow \frac{1}{2} f(n) \leqslant f(n) \pm g(n) \leqslant \frac{3}{2} f(n)$ ■

Lm 1.1.18. $n^k = o(n^{k+1})$

Доказательство. $\forall C\, \forall n \geqslant C \quad n^{k+1} \geqslant C \cdot n^k$ ■

Lm 1.1.19. $P(x)$ – многочлен, тогда $P(x) = \Theta(x^{\deg P})$ при старшем коэффициенте > 0.

Доказательство. $P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k$. По леммам 1.1.12, 1.1.18 имеем, что все слагаемые кроме $a_k x^k$ являются $o(x^{\deg P})$. Поэтому по лемме 1.1.17 вся сумма является $\Theta(x^k)$. ■

1.2. Мастер-теорема

Теорема 1.2.1. *Мастер-теорема* (теорема о простом рекуррентном соотношении)
Пусть $T(n) = aT(\frac{n}{b}) + f(n)$, где $f(n) = n^c$. При этом $a > 0, b > 1, c \geqslant 0$. Определим глубину рекурсии $k = \log_b n$. Тогда верно одно из трёх:

$$\begin{cases} T(n) = \Theta(a^k) = \Theta(n^{\log_b a}) & a > b^c \\ T(n) = \Theta(f(n)) = \Theta(n^c) & a < b^c \\ T(n) = \Theta(k \cdot f(n)) = \Theta(n^c \log n) & a = b^c \end{cases}$$

Доказательство. Раскроем рекуррентность:
$T(n) = f(n) + aT(\frac{n}{b}) = f(n) + af(\frac{n}{b}) + a^2 f(\frac{n}{b^2}) + \cdots = n^c + a(\frac{n}{b})^c + a^2(\frac{n}{b^2})^c + \ldots$
Тогда $T(n) = f(n)(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^k)$. При этом в сумме $k+1$ слагаемых.
Обозначим $q = \frac{a}{b^c}$ и оценим сумму $S(q) = 1 + q + \cdots + q^k$.
Если $q = 1$, то $S(q) = k + 1 = \log_b n + 1 = \Theta(\log_b n) \Rightarrow T(n) = \Theta(f(n) \log n)$.
Если $q < 1$, то $S(q) = \frac{1-q^{k+1}}{1-q} = \Theta(1) \Rightarrow T(n) = \Theta(f(n))$.
Если $q > 1$, то $S(q) = q^k + \frac{q^k-1}{q-1} = \Theta(q^k) \Rightarrow T(n) = \Theta(a^k(\frac{n}{b^k})^c) = \Theta(a^k)$. ■

Теорема 1.2.2. *Обобщение мастер-теоремы*
Мастер Теорема верна и для $f(n) = n^c \log^d n$
$T(n) = aT(\frac{n}{b}) + n^c \log^d n$. При $a > 0, b > 1, c \geqslant 0, d \geqslant 0$.

$$\begin{cases} T(n) = \Theta(a^k) = \Theta(n^{\log_b a}) & a > b^c \\ T(n) = \Theta(f(n)) = \Theta(n^c \log^d n) & a < b^c \\ T(n) = \Theta(k \cdot f(n)) = \Theta(n^c \log^{d+1} n) & a = b^c \end{cases}$$

Без доказательства. ■

1.3. Доказательства по индукции

Lm 1.3.1. *Доказательство по индукции*
Есть простой метод решения рекуррентных соотношений: угадать ответ, доказать его по индукции. Рассмотрим на примере $T(n) = \max\limits_{x=1..n-1}\big(T(x) + T(n-x) + x(n-x)\big)$.
Докажем, что $T(n) = \mathcal{O}(n^2)$, для этого достаточно доказать $T(n) \leqslant n^2$:
База: $T(1) = 1 \leqslant 1^2$.
Переход: $T(n) \leqslant \max\limits_{x=1..n-1}\big(x^2 + (n-x)^2 + x(n-x)\big) \leqslant \max\limits_{x=1..n-1}\big(x^2 + (n-x)^2 + 2x(n-x)\big) = n^2$

1.4. o-обозначение через предел

Def 1.4.1. $f = o(g)$ *Определение через предел:* $\lim\limits_{n \to +\infty} \frac{f(n)}{g(n)} = 0$

Lm 1.4.2. Определения o эквивалентны

Доказательство. Вспомним, что речь о положительных функциях f и g.
Распишем предел по определению: $\forall C > 0 \quad \exists N \quad \forall n \geqslant N \quad \frac{f(n)}{g(n)} \leqslant C \Leftrightarrow f(n) \leqslant Cg(n)$. ■

1.5. Примеры по теме асимптотики

• **Вложенные циклы for**

```
1 #define forn(i, n) for (int i = 0; i < n; i++)
2 int counter = 0, n = 100;
3 forn(i, n)
4   forn(j, i)
5     forn(k, j)
6       forn(l, k)
7         forn(m, l)
8           counter++;
9 cout << counter << endl;
```

Чему равен `counter`? Во-первых, есть точный ответ: $\binom{n}{5} \approx \frac{n^5}{5!}$. Во-вторых, мы можем сходу посчитать число циклов и оценить ответ как $\mathcal{O}(n^5)$, правда константа $\frac{1}{120}$ важна, оценка через $\mathcal{O}$ не даёт полное представление о времени работы.

• **Сумма гармонического ряда**

Докажем более простым способом, что $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$

$$1 + \lfloor \log_2 n \rfloor \geqslant \tfrac{1}{1} + \overbrace{\tfrac{1}{2} + \tfrac{1}{2}}^{1} + \overbrace{\tfrac{1}{4} + \tfrac{1}{4} + \tfrac{1}{4} + \tfrac{1}{4}}^{1} + \overbrace{\tfrac{1}{8} + \ldots}^{1\ldots} \geqslant \sum_{k=1}^{n} \tfrac{1}{k} = \tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} + \tfrac{1}{6} + \tfrac{1}{7} + \tfrac{1}{8} + \ldots \geqslant$$
$$\tfrac{1}{1} + \underbrace{\tfrac{1}{2}}_{1/2} + \underbrace{\tfrac{1}{4} + \tfrac{1}{4}}_{1/2} + \underbrace{\tfrac{1}{8} + \tfrac{1}{8} + \tfrac{1}{8} + \tfrac{1}{8}}_{1/2} + \ldots \geqslant 1 + \tfrac{1}{2}\lfloor \log_2 n \rfloor \Rightarrow \sum_{k=1}^{n} \tfrac{1}{k} = \Theta(\log n)$$

1.6. Сравнение асимптотик

Def 1.6.1. *Линейная сложность* $\mathcal{O}(n)$

Def 1.6.2. *Квадратичная сложность* $\mathcal{O}(n^2)$

Def 1.6.3. *Полиномиальная сложность* $\exists\, k > 0 \colon \mathcal{O}(n^k)$

Def 1.6.4. *Полилогарифм* $\exists\, k > 0 \colon \mathcal{O}(\log^k n)$

Def 1.6.5. *Экспоненциальная сложность* $\exists\, c > 0 \colon \mathcal{O}(2^{cn})$

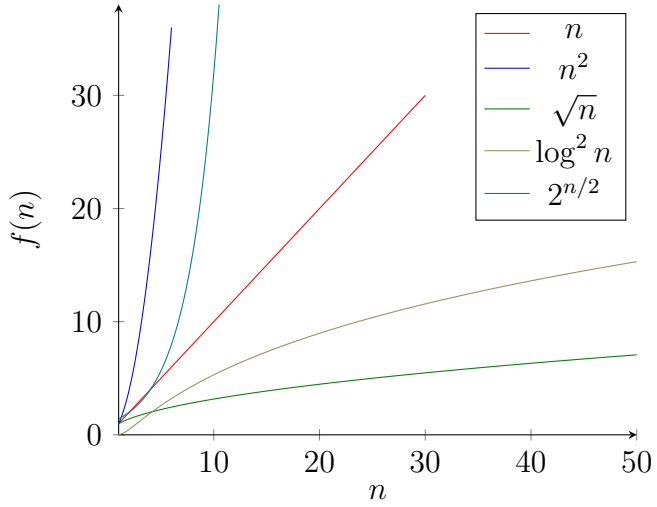
Теорема 1.6.6. $\forall x, y > 0, z > 1\ \exists N\ \forall n > N \colon \ \log^x n < n^y < z^n$

Следствие 1.6.7. $\forall x, y > 0, z > 1 \colon \ \log^x n = \mathcal{O}(n^y), n^y = \mathcal{O}(z^n)$

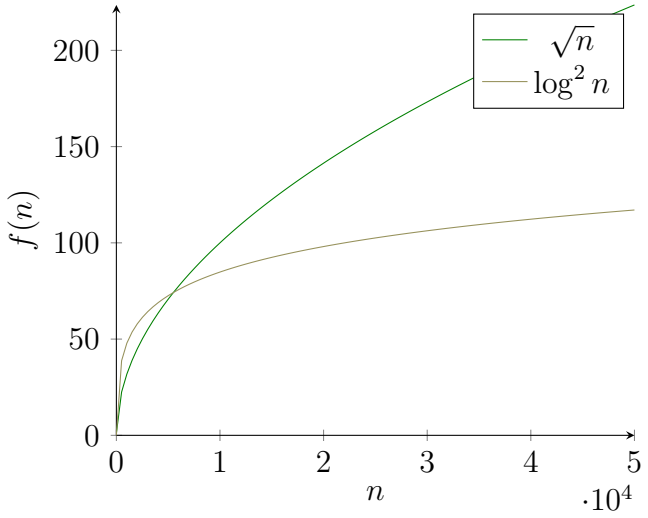
Доказательство. Возьмём константу 1. ■

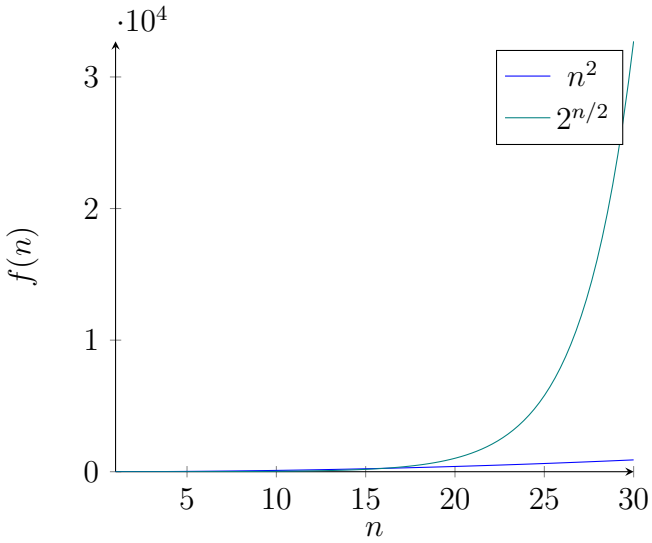
Следствие 1.6.8. $\forall x, y > 0, z > 1 \colon \ \log^x n = o(n^y), n^y = o(z^n)$

• **Посмотрим как ведут себя функции на графике**



Заметим, что $2^{n/2}, n^2$ и $\log^2 n, \sqrt{n}$ на бесконечности ведут себя иначе:





Лекция #2: Структуры данных

Сентябрь 2021

2.1. Частичные суммы

Дан массив `a[]` длины n, нужно отвечать на большое число запросов `get(l, r)` – посчитать сумму на отрезке $[l, r]$ массива `a[]`.
Наивное решение: на каждый запрос отвечать за $\Theta(r - l + 1) = \mathcal{O}(n)$.
Префиксные или частичные суммы:

```
1 void precalc() { // предподсчёт за O(n)
2   sum[0] = 0;
3   for (int i = 0; i < n; i++) sum[i + 1] = sum[i] + a[i]; // sum[i+1] = [0..i]
4 }
5 int get(int l, int r) { // [l..r]
6   return sum[r+1] - sum[l]; // [0..r] – [0..l), O(1)
7 }
```

2.2. Массив

Создать массив целых чисел на n элементов: `int a[n];`
Индексация начинается с 0, массивы имеют фиксированный размер. Функции:

1. `get(i)` – $a[i]$, обратиться к элементу массива с номером i, $\mathcal{O}(1)$
2. `set(i,x)` – $a[i] = x$, присвоить элементу под номером i значение x, $\mathcal{O}(1)$
3. `find(x)` – найти элемент со значением x, $\mathcal{O}(n)$
4. `add_begin(x)`, `add_end(x)` – добавить элемент в начало, в конец, $\mathcal{O}(n)$, $\mathcal{O}(n)$
5. `del_begin(x)`, `del_end(x)` – удалить элемент из начала, из конца, $\mathcal{O}(n)$, $\mathcal{O}(1)$

Последние команды работают долго т.к. нужно найти новый кусок памяти нужного размера, скопировать весь массив туда, удалить старый.
Другие названия для добавления: `insert`, `append`, `push`.
Другие названия для удаления: `remove`, `erase`, `pop`.

2.3. Вектор (расширяющийся массив)

Обычный массив не удобен тем, что его размер фиксирован заранее и ограничен. Идея улучшения: выделим заранее *size* ячеек памяти, когда реальный размер массива n станет больше *size*, удвоим *size*, перевыделим память. Операции с вектором:

`get(i), set(i, x)`	$\mathcal{O}(1)$ (как и у массива)
`find(x)`	$\mathcal{O}(n)$ (как и у массива)
`push_back(x)`	$\Theta(1)$ (в среднем)
`pop_back()`	$\Theta(1)$ (в худшем)

```
1 int size, n, *a;
2 void push_back(int x) {
3   if (n == size) {
4     int *b = new int[2 * size];
5     copy(a, a + size, b);
```

```
6       a = b, size *= 2;
7     }
8     a[n++] = x;
9   }
10  void pop_back() { n--; }
```

Теорема 2.3.1. Среднее время работы одной операции $\mathcal{O}(1)$

Доказательство. Заметим, что перед удвоением размера $n \to 2n$ будет хотя бы $\frac{n}{2}$ операций `push_back`, значит среднее время работы последней всех `push_back` между двумя удвоениями, включая последнее удвоение $\mathcal{O}(1)$ ■

2.4. Стек, очередь, дек

Это названия интерфейсов (множеств функций, доступных пользователю)

`Стек (stack)`	`push_back` за $\mathcal{O}(1)$, `pop_back` за $\mathcal{O}(1)$. First In Last Out.
`Очередь (queue)`	`push_back` за $\mathcal{O}(1)$, `pop_front` за $\mathcal{O}(1)$. First In First Out.
`Дек (deque)`	все 4 операции добавления/удаления.

Реализовывать все три структуры можно, как на списке, так и на векторе.
Деку нужен двусвязный список, очереди и стеку хватит односвязного.
Вектор у нас умеет удваиваться только при `push_back`. Что делать при `push_front`?

1. Можно удваиваться в другую сторону.
2. Можно использовать циклический вектор.

• **Дек на циклическом векторе**

`deque:` `{ vector<int>a; int start, end; }`, данные хранятся в `[start, end)`
`sz():` `{ return a.size(); }`
`n():` `{ return end - start + (start <= end ? 0 : sz()); }`
`get(i):` `{ return a[(i + start) % sz()]; }`
`push_front(x):` `{ start = (start - 1 + sz()) % sz(), a[start] = x; }`

2.5. Очередь, стек и дек с минимумом

В стеке можно поддерживать минимум.
Для этого по сути нужно поддерживать два стека – стек данных и стек минимумов.

• **Стек с минимумом** – это два стека.

`push(x): a.push(x), m.push(min(m.back(), x)`
Здесь `m` – "частичные минимумы", стек минимумов.

• **Очередь с минимумом через два стека**

Чтобы поддерживать минимум на очереди проще всего представить её, как два стека a и b.

```
1   Stack a, b;
2   void push(int x) { b.push(x); }
3   int pop() {
4     if (a.empty()) // стек a закончился, пора перенести элементы b в a
5       while (b.size())
6         a.push(b.pop());
7     return a.pop();
```

```
8 }
9 int getMin() { return min(a.getMin(), b.getMin()); }
```

Лекция #3: Бинарный поиск

Сентябрь 2021

3.1. Биппоиск

3.1.1. Обыкновенный

Дан отсортированный массив. Сортировать мы пока умеем только так:

```
int a[n]; sort(a, a + n);
vector<int> a(n); sort(a.begin(), a.end());
```

Сейчас мы научимся за $\mathcal{O}(\log n)$ проверять, есть ли в массиве число x.

```
bool find(int x) {
  int l = 0, r = n - 1;
  while (l <= r) { // ищем на отрезке [l, r]
    int m = (l + r) / 2;
    if (a[m] == x) return true;
    if (a[m] < x) l = m + 1;
    else r = m - 1;
  }
  return false;
}
```

Lm 3.1.1. Время работы $\mathcal{O}(\log n)$

Доказательство. Каждый раз мы уменьшаем длину отрезка $[l, r]$ как минимум в 2 раза. ■

В языке C++ есть стандартная функция `binary_search(a, a + n, x)`.

• **Lowerbound.** Можно искать более сложную величину $\min i$: $a_i \geqslant x$. Если все элементы меньше x, мы хотим, чтобы `lower_bound` вернул n: как будто бы после конца массива записано $+\infty$.

```
int lower_bound(int x) {
  int l = 0, r = n;           // мысленно считаем, что a[n] = +∞
  while (l < r) {             // ищем i на отрезке [l, r]
    int m = (l + r) / 2;      // m < r, поэтому никогда не обратимся к a[n]
    if (a[m] < x) l = m + 1; // i точно больше, чем m
    else r = m;               // i точно не больше, чем m
  }
  // i лежит на отрезке [l, r], l == r, поэтому i == l == r
  return l;
}
```

Время работы также $\mathcal{O}(\log n)$.

Заметим, что этот бинпоиск строго сильнее, функцию `find` теперь можно реализовать так:

```
bool find(int x) {
  int p = lower_bound(x);
  return p < n && a[p] == x;
}
```

Можно писать `lower_bound` немного по-другому и с другим инвариантом, см. вики-конспекты ИТМО.

В языке C++ есть стандартные функции

```
int i = lower_bound(a, a + n, x) - a; // min i: a[i] >= x
int i = upper_bound(a, a + n, x) - a; // min i: a[i] > x
```

Через них легко найти $[\max i\colon a_i \leqslant x] =$ `upper_bound` $- 1$ и $[\max i\colon a_i < x] =$ `lower_bound` $- 1$.

3.1.2. По предикату

Можно написать ещё более общий бинпоиск, при этом сделать код более простым. Предикат – функция, принимающая значения только 0 и 1. Пусть предикат $f(x) =$ (`x < i ? 0 : 1`). Тогда мы бинпоиском можем найти такие $l + 1 = r$, что $f(l) = 0, f(r) = 1$.

```
void find_predicate(int &l, int &r) { // f(l) = 0, f(r) = 1
  while (r - l > 1) {
    int m = (l + r) / 2;
    (f(m) ? r : l) = m; // короткая запись if (f(m)) r=m; else l=m;
  }
}
```

Как это использовать для решении задачи `lower_bound`?

```
bool f(int i) { return a[i] >= x; }
int l = -1, r = n; // мысленно добавим a[-1] = −∞, a[n] = +∞
find_predicate(l, r); // f() будет вызываться только для элементов от l+1 до r-1
return r; // f(r) = 1, f(r-1) = 0
```

Лекция #4: Куча

Сентябрь 2021

4.1. Бинарная куча

Рассмотрим массив $a[1..n]$. Его элементы образуют бинарное дерево с корнем в 1.
Дети i – вершины $2i$, $2i+1$. Отец i – вершина $\lfloor \frac{i}{2} \rfloor$.

Def 4.1.1. *Бинарная куча – массив, индексы которого образуют описанное выше дерево, в котором верно основное свойство кучи: для каждой вершины i значение $a[i]$ является минимумом в поддереве i.*

Lm 4.1.2. Высота кучи равна $\lfloor \log_2 n \rfloor$

Доказательство. Высота равна длине пути от n до корня.
Заметим, что для всех чисел от 2^k до $2^{k+1}-1$ длина пути в точности k. ■

• Интерфейс

Бинарная куча за $\mathcal{O}(\log n)$ умеет делать следующие операции.

1. `GetMin()`. Нахождение минимального элемента.
2. `Add(x)`. Добавление элемента.
3. `ExtractMin()`. Извлечение (удаление) минимума.

Если для элементов хранятся "обратные указатели", позволяющие за $\mathcal{O}(1)$ переходить от элемента к ячейке кучи, содержащей элемент, то куча также за $\mathcal{O}(\log n)$ умеет:

4. `DecreaseKey(x, y)`. Уменьшить значение ключа x до y.
5. `Del(x)`. Удалить из кучи x.

4.1.1. GetMin, Add, ExtractMin

Реализуем сперва три простые операции.
Наша куча: `int n, *a;`. Память выделена, её достаточно.

```
1 void Init()       { n = 0; }
2 int GetMin()    { return a[1]; }
3 void Add(int x) { a[++n] = x, siftUp(n); }
4 void ExtractMin()   { swap(a[1], a[n--]), siftDown(1); }
5 // DelMin перед удалением сохранил минимум в a[n]
```

Здесь `siftUp` – проталкивание элемента вверх, а `siftDown` – проталкивание элемента вниз. Обе процедуры считают, что дерево обладает свойством кучи везде, кроме указанного элемента.

```
1 void siftUp(int i) {
2   while (i > 1 && a[i / 2] > a[i]) // пока мы не корень и отец нас больше
3     swap(a[i], a[i / 2]), i /= 2;
4 }
5 void siftDown(int i) {
6   while (1) {
7     int l = 2 * i;
```

```
8       if (l + 1 <= n && a[l + 1] < a[l]) l++; // выбрать меньшего из детей
9       if (!(l <= n && a[l] < a[i])) break; // если все дети не меньше нас, это конец
10      swap(a[l], a[i]), i = l; // перейти в ребёнка
11    }
12  }
```

Lm 4.1.3. Обе процедуры корректны

Набросок доказательства. По индукции на примере `siftUp`. В каждый момент времени верно, что поддерево i – корректная куча. Когда мы выйдем из `while`, у i нет проблем с отцом, поэтому вся куча корректна из предположения "корректно было всё кроме i". ■

Lm 4.1.4. Обе процедуры работают за $\mathcal{O}(\log n)$

Доказательство. Они работают за высоту кучи, которая по 4.1.2 равна $\mathcal{O}(\log n)$. ■

4.1.2. Build, HeapSort

```
1  void Build(int n, int *a) {
2    for (int i = n; i >= 1; i--)
3      siftDown(i);
4  }
```

Lm 4.1.5. Функция `Build` построит корректную бинарную кучу.

Доказательство. Когда мы проталкиваем i, по индукции слева и справа уже корректные бинарные кучи. По корректности операции `sift_down` после проталкивания i, поддерево i является корректной бинарной кучей. ■

Lm 4.1.6. Время работы функции `Build` $\Theta(n)$

Доказательство. Пусть $n = 2^k - 1$, тогда наша куча – полное бинарное дерево. На самом последнем (нижнем) уровне будет 2^{k-1} элементов, на предпоследнем 2^{k-2} элементов и т.д. `sift_down(i)` работает за $\mathcal{O}$(глубины поддерева i), поэтому суммарное время работы $\sum\limits_{i=1}^{k} 2^{k-i} i = 2^k \sum\limits_{i=1}^{k} \frac{i}{2^i} \overset{(*)}{=} 2^k \cdot \Theta(1) = \Theta(n)$. `(*) доказано на практике.` ■

```
1  void HeapSort() {
2    Build(n, a); // строим очередь с максимумом, O(n)
3    forn(i, n) DelMax(); // максимум окажется в конце и т.д., O(nlogn)
4  }
```

Lm 4.1.7. Функция `HeapSort` работает за $\mathcal{O}(n \log n)$, использует $\mathcal{O}(1)$ дополнительной памяти.

Доказательство. Важно, что функция `Build` не копирует массив, строит кучу прямо в `a`. ■

4.2. Пополняемые структуры

Все описанные в этом разделе идеи применимы не ко всем структурам данных. Тем не менее к любой структуре любую из описанных идей можно *попробовать* применить.

4.2.1. Ничего → Удаление

Вид "ленивого удаления". Пример: куча. Есть операция `DelMin`, хотим операцию удаления произвольного элемента, ничего не делая. Будем хранить две кучи – добавленные элементы и удалённые элементы.

```
1 Heap a, b;
2 void Add(int x) { a.add(x); }
3 void Del(int x) { b.add(x); }
4 int DelMin() {
5   while (b.size() && a.min() == b.min())
6     a.delMin(), b.delMin(); // пропускаем уже удалённые элементы
7   return a.delMin();
8 }
```

Время работы `DelMin` осталось тем же, стало амортизированным.
В худшем случае все `DelMin` в сумме работают $\Theta(n \log n)$.
Зачем это нужно? Например, `std::priority_queue`.

4.2.2. Поиск → Удаление

Вид "ленивого удаления". Таким приёмом мы уже пользовались при удалении из хеш-таблицы с открытой адресацией. Идея: у нас есть операция `Find`, отлично, найдём элемент, пометим его, как удалённый. Удалять прямо сейчас не будем.

Лекция #5: Сортировки

Сентябрь 2021

5.1. Квадратичные сортировки

Def 5.1.1. *Сортировка называется стабильной, если одинаковые элементы она оставляет в исходном порядке.*

Пример: сортируем людей по имени. Люди с точки зрения сортировки считаются равными, если у них одинаковое имя. Тем не менее порядок людей в итоге важен. Во всех таблицах (гуглдок и т.д.) сортировки, которые вы применяете к данным, стабильные.

Def 5.1.2. *Инверсия – пара* $i < j : a_i > a_j$

Def 5.1.3. I *– обозначение для числа инверсий в массиве*

Lm 5.1.4. Массив отсортирован $\Leftrightarrow \mathtt{I} = 0$

• **Selection sort** (сортировка выбором)

На каждом шаге выбираем минимальный элемент, ставим его в начале.

```
1 for (int i = 0; i < n; i++) {
2   j = index of min on [i..n);
3   swap(a[j], a[i]);
4 }
```

• **Insertion sort** (сортировка вставками)

Пусть префикс длины i уже отсортирован, возьмём a_i и вставим куда надо.

```
1 for (int i = 0; i < n; i++)
2   for (int j = i; j > 0 && a[j] > a[j-1]; j--)
3     swap(a[j], a[j-1]);
```

Корректность: по индукции по i ■

Заметим, что можно ускорить сортировку, место для вставки искать бинпоиском.
Сортировка всё равно останется квадратичной.

• **Bubble sort** (сортировка пузырьком)

Бесполезна. Изучается, как дань истории. Простая.

```
1 for (int i = 0; i < n; i++)
2   for (int j = 1; j < n; j++)
3     if (a[j-1] > a[j])
4       swap(a[j-1], a[j]);
```

Корректность: на каждой итерации внешнего цикла очередной максимальный элемент встаёт на своё место, “всплывает”.

• **Сравним пройденные сортировки.**

Название	`<`	`swap`	`stable`
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	-
Insertion	$\mathcal{O}(n + \mathtt{I})$	$\mathcal{O}(\mathtt{I})$	+
Ins + B.S.	$\mathcal{O}(n \log n)$	$\mathcal{O}(\mathtt{I})$	+
Bubble	$\mathcal{O}(n^2)$	$\mathcal{O}(\mathtt{I})$	+

Три нижние стабильны, т.к. `swap` применяется только к соседям, образующим инверсию. Ко-

личество `swap`-ов в `Insertion` равно `I` (каждый `swap` ровно на 1 уменьшает `I`).

Чем ценна сортировка выбором? `swap` может быть дорогой операцией. Пример: мы сортируем 10^3 тяжёлых для `swap` объектов, не имея дополнительной памяти.

Чем ценна сортировка вставками? Малая константа. Самая быстрая по константе.

5.2. Оценка снизу на время сортировки

Пусть для сортировки объектов нам разрешено общаться с этими объектами единственным способом – сравнивать их на больше/меньше. Такие сортировки называются *основанные на сравнениях*.

Lm 5.2.1. Сортировка, основанная на сравнениях, делает на всех тестах $o(n \log n)$ сравнений $\Rightarrow \exists$ тест, на котором результат сортировки **не** корректен.

Доказательство. Докажем, что $\exists$ тест вида "перестановка". Всего есть $n!$ различных перестановок. Пусть сортировка делает не более k сравнений. Заставим её делать ровно k сравнений (возможно, несколько бесполезных). Результат каждого сравнения – меньше (0) или больше (1). Сортировка получает k бит информации, и результат её работы зависит только от этих k бит. То есть, если для двух перестановок она получит одни и те же k бит, одну из этих двух перестановок она отсортирует неправильно. Сортировка корректна $\Rightarrow 2^k \geqslant n!$.
$2^k < n! \Rightarrow$ сортировка не корректна. Осталось вспомнить, что $\log(n!) = \Theta(n \log n)$. ■

Мы доказали *нижнюю оценку* на время работы произвольной сортировки сравнениями. Доказали, что любая детерминированная (без использования случайных чисел) корректная сортировка делает хотя бы $\Omega(n \log n)$ сравнений.

5.3. Быстрые сортировки

Мы уже знаем одну сортировку за $\mathcal{O}(n \log n)$ – `HeapSort`.
Отметим её замечательные свойства: не использует дополнительной памяти, детерминирована.

5.3.1. MergeSort (слиянием)

Идея: отсортируем левую половину массива, правую половину массива, сольём два отсортированных массива в один методом двух указателей.

```
1 void MergeSort(int l, int r, int *a, int *buffer) { // [l, r)
2   if (r - l <= 1) return;
3   int m = (l + r) / 2;
4   MergeSort(l, m, a, buffer);
5   MergeSort(m, r, a, buffer);
6   Merge(l, m, r, a, buffer); // слияние за O(r-l), используем буффер
7 }
```

`buffer` – дополнительная память, которая нужна функции `Merge`. Функция `Merge` берёт отсортированные куски $[l, m)$, $[m, r)$, запускает метод двух указателей, который отсортированное объединение записывает в `buffer`. Затем `buffer` копируется обратно в `a`$[l, r)$.

Lm 5.3.1. Время работы $\mathcal{O}(n \log n)$

Доказательство. $T(n) = 2T(\frac{n}{2}) + n = \Theta(n \log n)$ (по мастер-теореме) ■

• **Нерекурсивная версия без копирования памяти**

Оставим ту же процедуру `Merge`, перепишем только рекурсивную функцию:

```
int n;
vector<int> a(n), buffer(n);
for (int k = 0; (1 << k) < n; k++)
  for (int i = 0; i < n; i += 2 * (1 << k))
    Merge(i, min(n, i + (1 << k)), min(n, i + 2 * (1 << k)), a, buffer)
  swap(a, bufffer); // O(1)
return a; // результат содержится именно тут, указатель может отличаться от исходного a
```

5.3.2. QuickSort (реально быстрая)

Идея: выберем некий x, разобьём наш массив a на три части $< x$, $= x$, $> x$, сделаем два рекурсивных вызова, чтобы отсортировать первую и третью части. Утверждается, что сортировка будет быстро работать, если как x взять случайный элемент a

```
def QuickSort(a):
  if len(a) <= 1: return a
  x = random.choice(a)
  b0 = select (< x).
  b1 = select (= x).
  b2 = select (> x).
  return QuickSort(b0) + b1 + QuickSort(b2)
```

Этот псевдокод описывает общую идею, но обычно, чтобы `QuickSort` была реально быстрой сортировкой, используют другую версию разделения массива на части.

Код 5.3.2. Быстрый `partition`.

```
void Partition(int l, int r, int x, int *a, int &i, int &j) { // [l,r], x ∈ a[l,r]
  i = l, j = r;
  while (i <= j) {
    while (a[i] < x) i++;
    while (a[j] > x) j--;
    if (i <= j) swap(a[i++], a[j--]);
  }
}
```

Этот вариант разбивает отрезок $[l, r]$ массива a на части $[l, j](j, i)[i, r]$.

Замечание 5.3.3. $a[l, j] \leqslant x$, $a(j, i) = x$, $a[i, r] \geqslant x$

Замечание 5.3.4. Алгоритм не выйдет за пределы $[l, r]$

Доказательство. $x \in a[l, r]$, поэтому выполнится хотя бы один `swap`. После `swap` верно $l < i \geqslant j < r$. Более того $a[l] \leqslant x, a[r] \geqslant x \Rightarrow$ циклы в строках (4)(5) не выйдут за l, r. ■

Код 5.3.5. Собственно код быстрой сортировки:

```
void QuickSort( int l, int r, int *a ) { // `[l, r]`
  if (l >= r) return;
  int i, j;
  Partition(l, r, a[random [l, r]], i, j);
```

```
5   QuickSort(l, j, a); // j < i
6   QuickSort(i, r, a); // j < i
7 }
```

• **Глубина (элиминация хвостовой рекурсии)**

Можно делать не два рекурсивных вызова, а только один, от меньшей части.
Тогда в худшем случае доппамять = глубина = $\mathcal{O}(\log n)$.
Вместо второго вызова (l_2, r_2) сделаем `l = l2, r = r2, goto start`.

• **Выбор** x

Можно показать, что при любом детерминированом выборе x или даже как медианы элементов любых трёх фиксированных элементов, $\exists$ тест, на котором время работы сортировки $\Theta(n^2)$. (Мы явным образом показали только для `x = a[l]`.) Чтобы на любом тесте `QuickSort` работал $\mathcal{O}(n \log n)$, нужно выбирать `x = a[random l..r]`. Тем не менее, так как рандом – медленная функция, иногда для скорости пишут версию без рандома.

5.3.3. Оценка времени работы

Будем оценивать `QuickSort`, основанный на `partition`, который делит элементы на $(< x)$, x, $(> x)$. Также мы предполагаем, что все элементы различны.

• **Доказательство.**

Время работы вероятностного алгоритма — среднее арифметическое по всем рандомам. Время QuickSort пропорционально числу сравнений. Число сравнений – сумма по всем парам $i < j$ характеристической функции "сравнивали ли мы эту пару", каждую пару мы сравним не более одного раза.

$$\frac{1}{R}\sum_{random} T(i) = \frac{1}{R}\sum_{random}\Big(\sum_{i<j} is(i,j)\Big) = \sum_{i<j}\Big(\frac{1}{R}\sum_{random} is(i,j)\Big) = \sum_{i<j} Pr[\text{сравнения}(i,j)]$$

Где Pr – вероятность. Осталось оценить вероятность. Для этого скажем, что у каждого элемента есть его индекс в отсортированном массиве.

Lm 5.3.6. $Pr[\text{сравнения}(i,j)] = \frac{2}{j-i+1}$ при $i < j$

Доказательство. Сравниться i и j могут только, если при некотором `Partition` выбор пал на один из них. Рассмотрим дерево рекурсии. Посмотрим на самую глубокую вершину, $[l, r)$ всё ещё содержит и i-й, и j-й элементы. Все элементы $i+1, i+2, \ldots, j-1$ также содержатся (т.к. i и j – индексы в отсортированном массиве). i и j разделятся $\Rightarrow$ `Partition` выберет один из $j-i+1$ элементов отрезка $[i,j]$. С вероятностью $\frac{2}{j-i+1}$ он совпадёт с i или j, тогда и только тогда i и j сравнятся. ■

Осталось посчитать сумму $\sum_{i<j}\frac{2}{j-i+1} = 2\sum_i\sum_{j>i}\frac{1}{j-i+1} \leqslant 2(n\ln n + \Theta(n))$ ■

5.3.4. Introsort'97

На основе Quick Sort можно сделать быструю детерминированную сортировку.

1. Делаем Quick Sort от N элементов
2. Если $r - l$ не более 10, переключаемся на Insertion Sort
3. Если глубина более $3\ln N$, переключаемся на Heap Sort

Такая сортировка называется Introsort, в `C++::STL` используется именно она.

5.3.5. Сравнение сортировок

Название	Время	Память	Стабильность
HeapSort	$\mathcal{O}(n \log n)$	$\Theta(1)$	-
MergeSort	$\Theta(n \log n)$	$\Theta(n)$	+
QuickSort	$\Theta(n^2)$ в худшем $\mathcal{O}(n \log n)$ в ср. по сл. битам	$\mathcal{O}(n)$	-
QuickSort с эл. хв. рек.	$\Theta(n^2)$ в худшем $\mathcal{O}(n \log n)$ в ср. по сл. битам	$\mathcal{O}(\log n)$	-
Introsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	-

Интересен вопрос существования стабильной сортировки, работающей за $\mathcal{O}(n \log n)$, не использующей дополнительную память. Среди уже изученных такой нет, но вообще они существуют.

5.4. Порядковые статистики

Задача поиска k-й порядковой статистики формулируется своим простейшим решением

```
int statistic(a, k) {
    sort(a);
    return a[k];
}
```

5.4.1. Одноветочный QuickSort

Вспомним реализацию Quick Sort 5.3.5
Quick Sort = выбрать x + `Partition` + 2 рекурсивных вызова Quick Sort.
Будем делать только 1 рекурсивный вызов:

Код 5.4.1. Порядковая статистика

```
int Statistic( int l, int r, int *a, int k ) { // [l, r]
    if (r <= l) return;
    int i, j, x;
    Partition(l, r, x = a[random [l, r]], i, j);
    if (j < k && k < i) return x;
    return k <= j ? Statistic(l, j, a) : Statistic(i, r, a);
}
```

Действительно, зачем вызываться от второй половины, если ответ находится именно в первой?

Теорема 5.4.2. Время работы 5.4.1 равно $\Theta(n)$

Доказательство. С вероятностью $\frac{1}{3}$ мы попадем в элемент, который лежит во второй трети сортированного массива. Тогда после `Partition` размеры кусков будут не более $\frac{2}{3}n$. Если же не попали, то размеры не более n, вероятность этого $\frac{2}{3}$. Итого:

$$T(n) = n + \tfrac{1}{3}T(\tfrac{2}{3}n) + \tfrac{2}{3}T(n) \Rightarrow T(n) = 3n + T(\tfrac{2}{3}n) \leqslant 9n = \Theta(n)$$ ■

Замечание 5.4.3. Мы могли бы повторить доказательство **??**, тогда нам нужно было бы оценить сумму $\sum T(\max(i, n-i-1))$. Это технически сложнее, зато дало бы константу 4.

5.4.2. (*) Детерминированный алгоритм

Statistic = выбрать x + `Partition` + 1 рекурсивный вызов Statistic.
Чтобы этот алгоритм стал детерминированным, нужно хорошо выбирать x.

• **Идея.** Разобьем n элементов на группы по 5 элементов, в каждой группе выберем медиану, из полученных $\frac{n}{5}$ медиан выберем медиану, это и есть x.

Утверждение 5.4.4. На массиве длины 5 медиану можно выбрать за 6 сравнений.

Поскольку из $\frac{n}{5}$ меньшие $\frac{n}{10}$ не больше x, хотя бы $\frac{3}{10}n$ элементов исходного массива **не более** выбранного x. Аналогично хотя бы $\frac{3}{10}n$ элементов **не менее** выбранного x. Это значит, что после `Partition` размеры кусков не будут превосходить $\frac{7}{10}n$. Теперь оценим время работы алгоритма:

$$T(n) \leqslant 6\tfrac{n}{5} + T(\tfrac{n}{5}) + n + T(\tfrac{7}{10}n) = 2.2(n + \tfrac{9}{10}n + \left(\tfrac{9}{10}\right)^2 n + \dots) = 22n = \Theta(n)$$ ■

5.4.3. C++

В `C++::STL` есть следующие функции

1. `nth_element(a, a + k, a + n)` – k-я статистика на основе одноветочного Quick Sort. После вызова функции k-я статистика стоит на своём месте, слева меньшие, справа большие.
2. `partition(a, a + n, predicate)` – Partition по произвольному предикату.

5.5. Integer sorting

Если мы хотим сортировать вещественные числа, данные с точностью $\pm\varepsilon$, их можно привести к целым: домножить на $\frac{1}{\varepsilon}$ и округлить, после чего сортировать целые.

5.5.1. Наивная CountSort (подсчётом)

Целые числа от 0 до $m-1$ можно отсортировать за $\mathcal{O}(n+m)$.
В частности целые число от 0 до $2n$ можно отсортировать за $\mathcal{O}(n)$.

```
1 int n, a[n];
2 for (int i = 0; i < n; i++) // Θ(n)
3   count[x]++; // насчитали, сколько раз x встречается в a
4 for (int x = 0; x < m; x++) // Θ(m), перебрали x в порядке возрастания
5   while (count[x]--)
6     out(x);
```

Мы уже доказали, что сортировки, основанные на сравнениях не могут работать за $\mathcal{O}(n)$. В данном случае мы пользовались операцией `count[x]++`, ячейки массива `count` упорядочены также, как и числа. Именно это даёт ускорение.

5.5.2. Стабильная CountSort

Давайте используем уже известный нам CountSort, чтобы стабильно отсортировать пары $\langle a_i, b_i \rangle$

```
1 void CountSort(int n, int *a, int *b) { // 0 <= a[i] < m
2     for (int i = 0; i < n; i++)
3         count[a[i]]++; // сколько раз встречается
4     // pos[i] - ‘позиция начала куска ответа, состоящего из пар <i, ?>‘
```

```
5       for (int i = 0; i + 1 < m; i++)
6           pos[i + 1] = pos[i] + count[i];
7       for (int i = 0; i < n; i++)
8           result[pos[a[i]]++] = {a[i], b[i]}; // нужна доппамять!
9   }
```

Важно то, что сортировка **стабильна**, из этого следует наш следующий алгоритм:

5.5.3. Radix sort (цифровая, поразрядная)

Отсортируем n строк длины L. Символ строки – целое число из $[0, k)$.

• **Алгоритм:** отсортируем сперва по последнему символу, затем по предпоследнему и т.д.

• **Корректность:** мы сортируем стабильной сортировкой строки по символу номер i, строки уже отсортированы по символам $(i, L]$. Из стабильности имеем, что строки равные по i-му символу будут отсортированы как раз по $(i, L] \Rightarrow$ теперь строки отсортированы по $[i, L]$.

• **Время работы:** мы L раз вызвали сортировку подсчётом $\Rightarrow \mathcal{O}(L(n + k))$.
Теперь заметим, что $\forall k$ число из $[0, m)$ – строка длины $\log_k m$ над алфавитом $[0, k)$.
При $k = n$ получаем время работы $n\lceil \log_n m \rceil$.

Лекция #6: Динамическое программирование

Октябрь 2021

6.1. Базовые понятия

"Метод динамического программирования" будем кратко называть "динамикой". Познакомимся с этим методом через простой пример.

6.1.1. Условие задачи

У нас есть число 1, за ход можно заменить его на любое из $x + 1$, $x + 7$, $2x$, $3x$. За какое минимальное число ходов мы можем получить n?

6.1.2. Динамика назад

$f[x]$ – минимальное число ходов, чтобы получить число x.
Тогда $f[x] = \min(f[x-1], f[x-7], f[\frac{x}{2}], f[\frac{x}{3}])$, причём запрещены переходы в не натуральные числа. При этом мы знаем, что $f[1] = 0$, получается решение:

```
1 vector<int> f(n + 1, 0);
2 f[1] = 0; // бесполезная строчка, просто подчеркнём факт
3 for (int i = 2; i <= n; i++) {
4   f[i] = f[i - 1] + 1;
5   if (i - 7 >= 1) f[i] = min(f[i], f[i - 7] + 1);
6   if (i % 2 == 0) f[i] = min(f[i], f[i / 2] + 1);
7   if (i % 3 == 0) f[i] = min(f[i], f[i / 3] + 1);
8 }
```

Когда мы считаем значение $f[x]$, для всех $y < x$ уже посчитано $f[y]$, поэтому $f[x]$ посчитается верно. Важно, что мы не пытаемся думать, что выгоднее сделать "вычесть 7" или "поделить на 2", мы честно перебираем все возможные ходы и выбираем оптимум. Введём операцию `relax` – улучшение ответа. Далее мы будем использовать во всех "динамиках".

```
1 void relax( int &a, int b ) { a = min(a, b); }
2 vector<int> f(n + 1, 0);
3 for (int i = 2; i <= n; i++) {
4   int r = f[i - 1];
5   if (i - 7 >= 1) relax(r, f[i - 7]);
6   if (i % 2 == 0) relax(r, f[i / 2]);
7   if (i % 3 == 0) relax(r, f[i / 3]);
8   f[i] = r + 1;
9 }
```

Операция `relax` именно улучшает ответ,
в зависимости от задачи или минимизирует его, или максимизирует.

Введём основные понятия

1. $f[x]$ – функция динамики
2. x – состояние динамики
3. $f[1] = 0$ – база динамики
4. $x \to x + 1, x + 7, 2x, 3x$ – переходы динамики

Исходная задача – посчитать $f[n]$.
Чтобы её решить, мы сводим её к подзадачам такого же вида меньшего размера – посчитать для всех $1 \leqslant i < n$, тогда сможем посчитать и $f[n]$. Важно, что для каждой подзадачи (для каждого x) мы считаем значение $f[x]$ ровно 1 раз. Время работы $\Theta(n)$.

6.1.3. Динамика вперёд

Решим ту же самую задачу тем же самым методом, но пойдём в другую сторону.

```
void relax( int &a, int b ) { a = min(a, b); }
vector<int> f(3 * n, INT_MAX); // 3 * n - чтобы меньше if-ов писать
f[1] = 0;
for (int i = 1; i < n; i++) {
  int F = f[i] + 1;
  relax(f[i + 1], F);
  relax(f[i + 7], F);
  relax(f[2 * i], F);
  relax(f[3 * i], F);
}
```

Для данной задачи код получился немного проще (убрали if-ы).
В общем случае нужно помнить про оба способа, выбирать более удобный.
Суть не поменялась: для каждого x будет верно $f[x] = \min(f[x-1], f[x-7], f[\frac{x}{2}], f[\frac{x}{3}])$.

• Интуиция для динамики вперёд и назад.

Назад: посчитали $f[x]$ через уже посчитанные подзадачи.
Вперёд: если $f[x]$ верно посчитано, мы можем обновить ответы для $f[x+1], f[x+7], \ldots$

6.1.4. Ленивая динамика

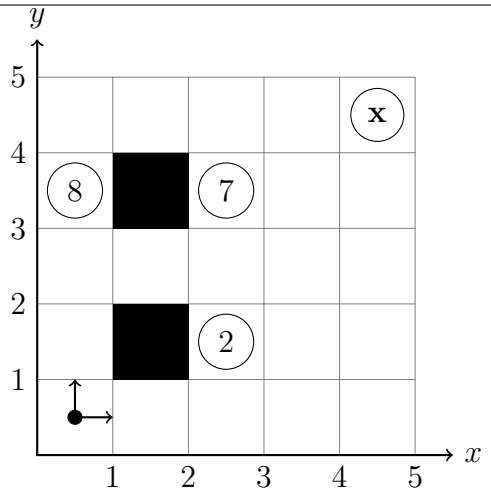
Это рекурсивный способ писать динамику назад, вычисляя значение только для тех состояний, которые действительно нужно посчитать.

```
vector<int> f(n + 1, -1);
int calc(int x) {
  int &r = f[x]; // результат вычисления f[x]
  if (r != -1) return r; // функция уже посчитана
  if (r == 1) return r = 0; // база динамики
  r = calc(x - 1);
  if (i - 7 >= 1) relax(r, calc(x - 7)); // стандартная ошибка: написать f[x-7]
  if (i % 2 == 0) relax(r, calc(x / 2));
  if (i % 3 == 0) relax(r, calc(x / 3));
  // теперь r=f[x] верно посчитан, в следующий раз для x сразу вернём уже посчитанный f[x]
  return ++r;
}
```

Для данной задачи этот код будет работать дольше, чем обычная “динамика назад циклом for”, так как переберёт те же состояния с бо́льшей константой (рекурсия хуже цикла).

Тем не менее представим, что переходы были бы $x \to 2x+1, 2x+7, 3x+2, 3x+10$. Тогда, например, ленивая динамика точно не зайдёт в состояния $[\frac{n}{2}..n)$, а если посчитать точно будет вообще работать за $\mathcal{O}(\log n)$. Чтобы она корректно работала для n порядка 10^{18} нужно лишь `vector<int> f(n + 1, -1);` заменить на `map<long long, int> f;`.

6.2. Ещё один пример



Вам дана матрица с непроходимыми клетками. В некоторых клетках лежат монетки разной ценности. За один ход можно сместиться вверх или вправо. Рассмотрим все пути из левой-нижней клетки в верхнюю-правую.

(a) Нужно найти число таких путей.
(b) Нужно найти путь, сумма ценностей монет на котором максимальна/минимальна.

Решим задачу динамикой назад:

$$cnt[x,y] = \begin{cases} cnt[x-1,y] + cnt[x,y-1] & \text{если клетка проходима} \\ 0 & \text{если клетка не проходима} \end{cases}$$

$$f[x,y] = \begin{cases} max(f[x-1,y], f[x,y-1]) + value[x,y] & \text{если клетка проходима} \\ -\infty & \text{если клетка не проходима} \end{cases}$$

Где $cnt[x,y]$ – количество путей из $(0,0)$ в (x,y),
$f[x,y]$ – вес максимального пути из $(0,0)$ в (x,y),
$value[x,y]$ – ценность монеты в клетке (x,y).

Решим задачу динамикой вперёд:

```
cnt <-- 0, f <-- -∞; // нейтральные значения
cnt[0,0] = 1, f[0,0] = 0; // база
for (int x = 0; x < width; x++)
  for (int y = 0; y < height; y++) {
    if (клетка не проходима) continue;
    cnt[x+1,y] += cnt[x,y];
    cnt[x,y+1] += cnt[x,y];
    f[x,y] += value[x,y];
    relax(f[x+1,y], f[x,y]);
    relax(f[x,y+1], f[x,y]);
  }
```

Ещё больше способов писать динамику.
Можно считать $cnt[x,y]$ – число путей из $(0,0)$ в (x,y). Это мы сейчас и делаем.
А можно считать $cnt'[x,y]$ – число путей из (x,y) в $(width-1, height-1)$.

6.3. Восстановление ответа

Посмотрим на задачу про матрицу и максимальный путь. Нас могут попросить найти только вес пути, а могут попросить найти и сам путь, то есть, “восстановить ответ”.

• Первый способ. Обратные ссылки.

Будем хранить $p[x,y]$ – из какого направления мы пришли в клетку (x,y). 0 – слева, 1 – снизу.
Функцию релаксации ответа нужно теперь переписать следующим образом:

```
void relax(int x, int y, int F, int P) {
  if (f[x,y] < F)
    f[x,y] = F, p[x,y] = P;
}
```

Чтобы восстановить путь, пройдём по обратным ссылкам от конца пути до его начала:

```
void outputPath() {
  for (int x = width-1, y = height-1; !(x == 0 && y == 0); p[x,y] ? y-- : x--)
    print(x, y);
}
```

• Второй способ. Не хранить обратные ссылки.

Заметим, что чтобы понять, куда нам идти назад из клетки (x, y), достаточно повторить то, что делает динамика назад, понять, как получилось значение $f[x, y]$:

```
if (x > 0 && f[x,y] == f[x-1,y] + \fix{value[x,y]}) // f[x,y] получилось из f[x-1,y]
  x--;
else
  y--;
```

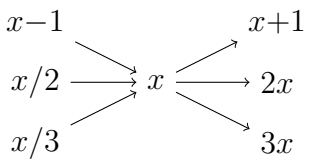
Второму способу нужно меньше памяти, но обычно он требует больше строк кода.

• Оптимизации по памяти

Если нам не нужно восстанавливать путь, заметим, что достаточно хранить только две строки динамики — $f[x], f[x+1]$, где $f[x]$ уже посчитана, а $f[x+1]$ мы сейчас вычисляем. Напомним, решение за $\Theta(n^2)$ времени и $\Theta(n)$ памяти (в отличии от $\Theta(n^2)$ памяти) попадёт в кеш и будет работать значительно быстрее.

6.4. (*) Графовая интерпретация

Рассмотрим граф, в котором вершины – состояния динамики, ориентированные рёбра – переходы динамики ($a \to b$ обозначает переход из a в b). Тогда мы только что решали задачи поиска пути из s (начальное состояние) в t (конечное состояние), минимального/максимального веса пути, а так же научились считать количество путей из s в t.



Утверждение 6.4.1. Любой задаче динамики соответствует ацикличный граф.

При этому динамика вперёд перебирала исходящие из v рёбра, а динамика назад перебирала входящие в v рёбра. Верно и обратное:

Утверждение 6.4.2. Для любого ацикличного графа и выделенных вершин s, t мы умеем искать min/max путь из s в t и считать количество путей из s в t, используя ленивую динамику.

Почему именно ленивую?

В произвольном графе мы не знаем, в каком порядке вычислять функцию для состояния. Но знаем, чтобы посчитать $f[v]$, достаточно знать значение динамики для начал всех входящих в v рёбер.

Почему только на ацикличном?

Пусть есть ориентированный цикл $a_1 \to a_2 \to \cdots \to a_k \to a_1$. Пусть мы хотим посчитать значение функции в вершине a_1, для этого нужно знать значение в вершине a_k, для этого в a_{k-1}, и так далее до a_1. Получили, чтобы посчитать значение в a_1, нужно его знать заранее.

Для произвольного ацикличного графа из V вершин и E рёбер динамика будет работать за $\mathcal{O}(V + E)$. При этом будут посещены лишь достижимые по обратным рёбрам из t вершины.

6.5. Checklist

Вы придумываете решение задачи, используя метод динамического программирования, или даже собираетесь писать код. Чтобы придумать решение, нужно увидеть некоторый процесс, например “мы идём слева направо, снизу вверх по матрице”. После этого, чтобы получилось корректное решение нужно увидеть

1. Состояние динамики (процесса) – мы стоим в клетке (x, y)
2. Переходы динамики – сделать шаг вправо или вверх
3. Начальное состояние динамики – стоим в клетке $(0, 0)$
4. Как ответ к исходной задаче выражается через посчитанную динамику (конечное состояние). В данном случае всё просто, ответ находится в $f[width-1, height-1]$
5. Порядок перебора состояний: если мы пишем динамику назад,
   то при обработке состояния (x, y) должны быть уже посчитаны $(x-1, y)$ и $(x, y-1)$.
   Всегда можно писать лениво, но цикл `for` быстрее рекурсии.
6. Если нужно восстановить ответ, не забыть подумать, как это делать.

6.6. Рюкзак без стоимостей и без повторений

6.6.1. Формулировка задачи

Нам дано n предметов с натуральными весами $a_0, a_1, \ldots, a_{n-1}$.
Требуется выбрать подмножество предметов суммарного веса ровно S.

1. Задача NP-трудна, если решить её за $\mathcal{O}(poly(n))$, получите 1 000 000\$.
2. Простое переборное решение рекурсией за 2^n.
3. $\exists$ решение за $2^{n/2}$ (meet in middle)

6.6.2. Решение динамикой

Будем рассматривать предметы по одному в порядке $0, 1, 2, \ldots$
Каждый из них будем или брать в подмножество-ответ, или не брать.

Состояние: перебрав первые i предметов, мы набрали вес x
Функция: $is[i, x] \Leftrightarrow$ мы могли выбрать подмножество веса x из первых i предметов
Начальное состояние: $(0, 0)$
Ответ на задачу: содержится в $is[n, S]$
Переходы:
$$\begin{cases} [i, x] \to [i+1, x] & \text{(не брать)} \\ [i, x] \to [i+1, x+a_i] & \text{(берём в ответ)} \end{cases}$$

```
1 bool is[n+1][2S+1] <-- 0; // пусть a[i] ⩽ S, запаса 2S хватит
2 is[0,0] = 1;
3 for (int i = 0; i < n; i++)
4   for (int j = 0; j <= S; j++)
5     if (is[i][j])
6       is[i+1][j] = is[i+1][j+a[i]] = 1;
7 // Answer = is[n][S]
```

Время работы $\Theta(nS)$, память $\Theta(nS)$.

6.7. Квадратичные динамики

Def 6.7.1. *Подпоследовательностью последовательности* $a_1, a_2, \ldots, a_n$ *будем называть* $a_{i_1}, a_{i_2}, \ldots, a_{i_k} : 1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$

Задача НОП (LCS). Поиск наибольшей общей подпоследовательности.
Даны две последовательности a и b, найти c, являющуюся подпоследовательностью и a, и b такую, что $len(c) \to \max$. Например, для последовательностей $\langle 1, 3, 10, 2, 7\rangle$ и $\langle 3, 5, 1, 2, 7, 11, 12\rangle$ возможными ответами являются $\langle 3, 2, 7\rangle$ и $\langle 1, 2, 7\rangle$.
Решение за $\mathcal{O}(nm)$
$f[i, j]$ – длина НОП для префиксов `a[1..i]` и `b[1..j]`.
Ответ содержится в $f[n, m]$, где n и m – длины последовательностей.
Посмотрим на последние элементы префиксов `a[i]` и `b[j]`.
Или мы один из них не возьмём в ответ, или возьмём оба. Делаем переходы:

$$f[i, j] = \max \begin{cases} f[i-1, j] \\ f[i, j-1] \\ f[i-1, j-1] + 1, \text{ если } a_i = b_j \end{cases}$$

Время работы $\Theta(n^2)$, количество памяти $\Theta(n^2)$.

Задача НВП (LIS). Поиск наибольшей возрастающей подпоследовательности.
Дана последовательность a, найти возрастающую подпоследовательность a: длина $\to \max$.
Например, для последовательности $\langle 5, 3, 3, 1, 7, 8, 1\rangle$ возможным ответом является $\langle 3, 7, 8\rangle$.
Решение за $\mathcal{O}(n^2)$
$f[i]$ – длина НВП, заканчивающейся ровно в i-м элементе.
Ответ содержится в $\max(f[1], f[2], \ldots, f[n])$, где n – длина последовательности.
Пересчёт: $f[i] = 1 + \max\limits_{j<i, a_j<a_i} f[j]$, максимум пустого множества равен нулю.
Время работы $\Theta(n^2)$, количество памяти $\Theta(n)$.

Расстояние Левенштейна. Оно же "редакционное расстояние".
Дана строка s и операции `INS`, `DEL`, `REPL` – добавление, удаление, замена одного символа.
Минимальным числом операций нужно получить строку t.
Например, чтобы из строки `STUDENT` получить строку `POSUDA`,
можно добавить зелёное (2), удалить красное (3), заменить синее (1), итого 6 операций.
Решение за $\mathcal{O}(nm)$
При решении задачи вместо добавлений в s будем удалять из t. Нужно сделать s и t равными.
$f[i, j]$ – редакционное расстояние между префиксами `s[1..i]` и `t[1..j]`

$$f[i, j] = \min \begin{cases} f[i-1, j] + 1 & \text{удаление из } s \\ f[i, j-1] + 1 & \text{удаление из } t \\ f[i-1, j-1] + w & \text{если } s_i = t_j, \text{ то } w = 0, \text{ иначе } w = 1 \end{cases}$$

Ответ содержится в $f[n, m]$, где n и m – длины строк.

Восстановление ответа.
Заметим, что пользуясь стандартными методами из раздела "восстановление ответа", мы можем найти не только число, но и восстановить сами общую последовательность, возрастающую последовательность и последовательность операций для редакционного расстояния.

6.8. Оптимизация памяти для НОП

Рассмотрим алгоритм для НОП. Если нам не требуется восстановление ответа, можно хранить только две строки динамики, памяти будет $\Theta(n)$. Восстанавливать ответ мы пока умеем только за $\Theta(n^2)$ памяти.

6.9. (*) НВП за $\mathcal{O}(n \log n)$

Пусть дана последовательность $a_1, a_2, \ldots, a_n$.

Код 6.9.1. Алгоритм поиска НВП за $\mathcal{O}(n \log n)$

```
1 x[] <-- inf
2 x[0] = -inf, answer = 0;
3 for (int i = 0; i < n; i++) {
4   int j = lower_bound(x, x + n, a[i]) - x;
5   x[j] = a[i], answer = max(answer, j);
6 }
```

Попробуем понять не только сам код, но и как его можно придумать, собрать из стандартных низкоуровневых идей оптимизации динамики.

Для начало рассмотрим процесс: идём слева направо, некоторые число берём в ответ (НВП), не которые не берём. Состояние этого процесса можно описать (k, len, i) – мы рассмотрели первые k чисел, из них len взяли в ответ, последнее взятое равно i. У нас есть два перехода $(k, len, i) \to (k{+}1, len, i)$, и, если $a_k > a_i$, можно сделать переход $(k, len, i) \to (k{+}1, len{+}1, k)$

Обычная идея преобразования "процесс $\to$ решение динамикой" – максимизировать $len[k, i]$. Но можно сделать $i[k, len]$ и минимизировать конец выбранной последовательности $x[i]$ ($x[k, len]$). Пойдём вторым путём, поймём, как вычислять $x[k, len]$ быстрее чем $\mathcal{O}(n^2)$.

Lm 6.9.2. $\forall k, len \quad x[k, len] \leqslant x[k, len + 1]$

Доказательство. Если была последовательность длины $len{+}1$, выкинем из неё любой элемент, получим длину len. ■

Обозначим $x_k[len] = x[k, len]$ – т.е. x_k – одномерный массив, строка массива x.

Lm 6.9.3. В реализации 6.9.1 строка 5 преобразует x_i в x_{i+1}

Доказательство. При переходе от x_i к x_{i+1}, мы пытаемся дописать элемент a_i ко всем существующим возрастающим подпоследовательностям, из 6.9.2 получаем, что приписать можно только к $x[0..j)$, где j посчитано в строке 4. Заметим, что $\forall k < j - 1$ к $x[k]$ дописывать бесполезно, так как $x[k + 1] < a[i]$. Допишем к $x[j - 1]$, получим, что $x[j]$ уменьшилось до a_i. ■

Восстановление ответа: кроме значения $x[j]$ будем также помнить позицию $xp[j]$, тогда:

```
1 prev[i] = xp[j - 1], xp[j] = i;
```

Насчитали таким образом ссылки `prev` на предыдущий элемент последовательности.

Лекция #7: Динамическое программирование (часть 2)

Октябрь 2021

7.1. Динамика по подотрезкам

Рассмотрим динамику по подотрезкам (практика, задача 4). Например, задачу о произведении матриц, которую, кстати, в 1981-м Hu & Shing решили за $\mathcal{O}(n \log n)$.
Решение динамикой: насчитать $f_{l,r}$, стоимость произведения отрезка матриц $[l, r]$

$$f_{l,r} = \min_{m \in [l,r)} (f_{l,m} + f_{m+1,r} + a_l a_{m+1} a_r)$$

Порядок перебора состояний: $r \uparrow\, l \downarrow$

7.2. Работа с множествами

Существует биекция между подмножествами n-элементного множества $X = \{0, 1, 2, \dots n-1\}$ и целыми числами от 0 до $2^n - 1$. $f\colon A \to \sum_{x \in A} 2^x$. Пример $\{0, 3\} \to 2^0 + 2^3 = 9$.
Таким образом множество можно использовать, как индекс массива.

Lm 7.2.1. $A \subseteq B \Rightarrow f(A) \leqslant f(B)$

С множествами, закодированными числами, многое можно делать за $\mathcal{O}(1)$:

`(1 << n) - 1`	Всё n-элементное множество X
`(A >> i) & 1`	Проверить наличие i-го элемента в множестве
`A | (1 << i)`	Добавить i-й элемент
`A & ~(1 << i)`	Удалить i-й элемент
`A ^ (1 << i)`	Добавить/удалить i-й элемент, был $\Rightarrow$ удалить, не был $\Rightarrow$ добавить
`A & B`	Пересечение
`A | B`	Объединение
`X & ~B`	Дополнение
`A & ~B`	Разность
`(A & B) = A`	Проверить, является ли A подмножеством B

7.3. Динамика по подмножествам

Теперь решим несколько простых задач.

7.4. Число бит в множестве (размер множества)

```
1 for (int A = 1; A < (1 << n); A++)
2   bit_cnt[A] = bit_cnt[A >> 1] + (A & 1);
```

Заметим, что аналогичный результат можно было получить простым перебором:

```
1 void go(int i, int A, int result) {
2   if (i == n) {
3     bit_cnt[A] = result;
4     return;
5   }
6   go(i + 1, A, result);
7   go(i + 1, A | (1 << i), result + 1);
8 }
```

```
9 go(0, 0, 0);
```

Здесь i – номер элемента, A – набранное множество, $result$ – его размер.

7.5. Гамильтоновы путь и цикл

Def 7.5.1. *Гамильтонов путь – путь, проходящий по всем вершинам ровно по одному разу.*
Будем искать гамильтонов путь динамическим программированием.
Строим путь. Что нам нужно помнить, чтобы продолжить строить путь? Где мы сейчас стоим и через какие вершины мы уже проходили, чтобы не пройти через них второй раз.

`is[A,v]` – можно ли построить путь, который проходит ровно по A, заканчивается в v.

Пусть $g[a, b]$ – есть ли ребро из b в a, n – число вершин в графе, тогда:

```
1 for (int i = 0; i < n; i++)
2   is[1 << i, i] = 1; // База: A = {i}, путь из одной вершины
3 for (int A = 0; A < (1 << n); A++)
4   for (int v = 0; v < n; v++)
5     if (is[A, v])
6       for (int x = 0; x < n; x++) // Переберём следующую вершину
7         if (x ∉ A && g[x, v])
8           is[A | (1 << x), x] = 1;
```

Время работы $\mathcal{O}(2^n n^2)$, память $\mathcal{O}(2^n n)$ машинных слов.

7.6. Перебор подмножеств

```
1 for (B = 0; B < 2^n; B++)
2   for (C = B; C > 0; C--, C &= B) // все непустые подмножества B
3     // ...
```

Операцией "`C--`" мы переходим к предыдущему подмножеству, операцией "`C &= B`" мы гарантируем, что в каждый момент времени C – всё ещё подмножество. (Тут надо аккуратно проверить, что действительно переберём все.) Важная тонкость: мы перебираем все подмножества кроме пустого.

Теорема 7.6.1. Время работы 3^n.

Доказательство. Когда множества B и C зафиксированы, каждый элемент находится в одном из трёх состояний: лежит в C, лежит в $B \setminus C$, лежит в дополнении B. Всего 3^n вариантов. ■

Доказательство. Другой способ доказать теорему.
Мы считаем $\sum_B 2^{|B|} = \sum_k \binom{n}{k} 2^k = (1+2)^n = 3^n$. ■

7.7. Set cover

Задача: дано $U = \{1, 2, \dots, n\}, A_1, A_2, \dots A_m \subseteq U$, выбрать минимальное число множеств, покрывающих U, то есть, $I\colon (\bigcup_{i \in I} A_i = U) \wedge (|I| \to \min)$. Взвешенная версия: у i-го множества есть положительный вес w_i, минимизировать $\sum_{i \in I} w_i$. Задача похожа на "рюкзак на подмножествах", её иногда так и называют. Решать её будем также.

Решение за $\mathcal{O}(2^n m)$.
Динамика $f[B]$ – минимальное число множеств из A_i, дающих в объединении B.

База: $f[0] = 0$. Переход: $relax(f[B \cup A_i], f[B] + 1)$.
Память $\mathcal{O}(2^n)$ (число состояний), времени $2^n m$ – по m переходов из каждого состояния.

Лекция #8: Алгоритм Хаффмана

Октябрь 2021

Алгоритм Хаффмана – наиболее известный алгоритм сжатия текста.
Хотим придумать префиксные коды, минимизирующие величину

$$F = \sum_i (len_i \cdot cnt_i)$$

где len_i – длина кода, cnt_i – частота i-го символа, F – количество бит в закодированном тексте.
На практике нужно ещё сохранить сами коды, и длина закодированного текста будет $F + |\texttt{store\_codes}|$. Префиксные коды удобно представлять в виде дерева. Чтобы раскодировать очередной символ закодированного текста, спустимся по дереву кодов до листа.

• Алгоритм построения префиксных кодов

Будем строить дерево снизу. Изначально каждому символу с ненулевой частотой сопоставлен лист дерева. Пока в алфавите больше одного символа, выберем символы x и y с минимальными cnt, создадим новый символ xy и вершину дерева для него, из которой ведут рёбра в x и y. Сделаем $cnt_{xy} = cnt_x + cnt_y$, удалим из алфавита x, y, добавим xy.

• Реализация

Если для извлечения минимума использовать кучу, мы получим время $\mathcal{O}(\Sigma \log \Sigma)$, где Σ – размер алфавита.
Можно изначально отсортировать символы по частоте и заметить, что новые символы по частоте только возрастают, поэтому для извлечения минимума достаточно поддерживать две очереди – “исходные символы” и “новые символы”. Минимум всегда содержится в начале одной из двух очередей $\Rightarrow \mathcal{O}(sort(\Sigma) + \Sigma)$.

Лекция #9: Графы и поиск в глубину

Октябрь 2021

9.1. Опеределения

Def 9.1.1. *Граф G – пара множеств вершин и рёбер $\langle V, E\rangle$. E – множество пар вершин.*

- Вершины ещё иногда называют *узлами*.
- Если направление ребёр не имеет значение, граф *неориентированный* (неорграф).
- Если направление ребёр имеет значение, граф *ориентированный* (орграф).
- Если ребру дополнительно сопоставлен вес, то граф называют *взвешенным*.
- Рёбра в орграфе ещё называют *дугами* и у ребра вводят понятие *начало* и *конец*.
- Если E – мультимножество, то могут быть равные рёбра, их называют *кратными*.
- Иногда, чтобы подчеркнуть, что E – мультимножество, говорят *мультиграф*.
- Для ребра $e = (a, b)$, говорят, что e *инцидентно* вершине a.
- *Степень* вершины v в неорграфе $deg\, v$ – количество инцидентных ей рёбер.
- В орграфе определяют ещё *входящую* и *исходящую степени*: $deg\, v = deg_{in}\, v + deg_{out}\, v$.
- Два ребра с общей вершиной называют *смежными*.
- Две вершины, соединённых ребром тоже называют *смежными*.
- Вершину степени ноль называют *изолированной*.
- Вершину степени один называют *висячей* или *листом*.
- Ребро (v, v) называют *петлёй*.
- *Простым* будем назвать граф без петель и кратных рёбер.

Def 9.1.2. *Путь – чередующаяся последовательность вершин и рёбер, в которой соседние элементы инцидентны, а крайние – вершины. В орграфе направление всех рёбер от i к i+1.*

- Путь *вершинно простой* или просто *простой*, если все вершины в нём различны.
- Путь *рёберно простой*, если все рёбра в нём различны.
- Пути можно рассматривать и в неорграфах и в орграфах. Если в графе нет кратных рёбер, обычно путь задают только последовательностью вершин.

Замечание 9.1.3. Иногда отдельно вводят понятие *маршрута*, *цепи*, *простой цепи*. Мы, чтобы не захламлять лексикон, ими пользоваться не будем.

- *Цикл* – путь с равными концами.
- Циклы тоже бывают вершинно и рёберно простыми.
- *Ацикличный* граф – граф без циклов.
- *Дерево* – ацикличный неорграф.

9.2. Хранение графа

Будем обозначать $|V| = n$, $|E| = m$. Иногда сами V и E будут обозначать размеры.

• Список рёбер

Можно просто хранить рёбра: `pair<int,int> edges[m];`
Чтобы в будущем удобно обрабатывать и взвешенные графы, и графы с потоком:

```
struct Edge { int from, to, weight; };
Edge edges[m];
```

• Матрица смежности

Можно для каждой пары вершин хранить наличие ребра, или количество рёбер, или вес...
`bool c[n][n];` для простого невзвешенного графа. n^2 бит памяти.
`int c[n][n];` для простого взвешенного графа или незвешенного мультиграфа. $\mathcal{O}(n^2)$ памяти.

`vector<int> c[n][n];` для взвешенного мультиграфа придётся хранить список всех весов всех рёбер между парой вершин.

`vector<vector<bool>> c(n, vector<bool>(n));` – чтобы первый способ правда весил n^2 бит.
Константа времени работы увеличится (нужно достать определённый бит 32-битного числа).

• Списки смежности

Можно для каждой вершины хранить список инцидентных ей рёбер: `vector<Edge> c[n];`
Чтобы списки смежности умели быстро удалять, заменяем `vector` на `set/unordered_set`.

• Сравнение способов хранения

Основных действий, которых нам нужно будет проделывать с графом не так много:

- `adjacent(v)` перебрать все инцидентные v рёбра.
- `get(a,b)` посмотреть на наличие/вес ребра между a и b.
- `all` просмотреть все рёбра графа
- `add(a,b)` добавить ребро в граф
- `del(a,b)` удалить ребро из графа

Ещё важно оценить дополнительную память.

	`adjacent`	`get`	`all`	`add`	`del`	memory
Список рёбер	$\mathcal{O}(E)$	$\mathcal{O}(E)$	$\mathcal{O}(E)$	$\mathcal{O}(1)$	$\mathcal{O}(E)$	$\mathcal{O}(E)$
Матрица смежности	$\mathcal{O}(V)$	$\mathcal{O}(1)$	$\mathcal{O}(V^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(V^2)$
Списки смежности (vector)	$\mathcal{O}(deg)$	$\mathcal{O}(deg)$	$\mathcal{O}(V+E)$	$\mathcal{O}(1)$	$\mathcal{O}(deg)$	$\mathcal{O}(E)$
Списки смежности (hashTable)	$\mathcal{O}(deg)$	$\mathcal{O}(1)$	$\mathcal{O}(V+E)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(E)$

Единственные плюсы первого способа – не нужна доппамять; в таком виде удобно хранить граф в файле (чтобы добавить одно ребро, допишем его в конец файла).
Если матрица смежности уж слишком велика, можно хранить хеш-таблицу $\langle a, b\rangle \to c[a, b]$.
В большинстве задач граф хранят **списками смежности** (иногда с `set` вместо `vector`).

Пример задачи, которую хорошо решает матрица смежности:
- даны граф и последовательность вершин в нём, проверить, что она – простой путь.

Пример задачи, которую хорошо решают списки смежности:
- пометить все вершины, смежные с v.

Пример задачи, где нужна сила обеих структур:
- даны две смежные вершины, найти третью, чтобы получился треугольник.

9.3. Поиск в глубину

Поиск в глубину = depth-first-search = `dfs`

- **Задача:** пометить все вершины, достижимые из a.
- **Решение:** рекурсивно вызываемся от всех соседей a.

```
vector<vector<int>> g(n); // собственно наш граф
void dfs(int v) {
  if (mark[v]) return; // от каждой вершины идём вглубь только один раз
  mark[v] = 1; // пометили
  for (int x : g[v])
    dfs(x);
}
dfs(a);
```

Время работы $\mathcal{O}(E)$, так как по каждому ребру `dfs` проходит не более одного раза.

- **Компоненты связности**

Def 9.3.1. *Вершины неор графа лежат в одной компоненте связности iff существует путь между ними. Иначе говоря, это классы эквивалентности отношения достижимости.*

Чуть модифицируем `dfs`: `mark[v]=cc`. И будем запускать его от всех вершин:

```
for (int a = 0; a < n; a++)
  if (!mark[a])
    cc++, dfs(a);
```

Теперь каждая вершина покрашена в цвет своей компоненты. Время работы $\mathcal{O}(V+E)$ – по каждому ребру `dfs` проходит не более одного раза, и кроме того запускается от каждой вершины.

- **Восстановление пути**

Ищем путь в определённую вершину? на обратном ходу рекурсии можно восстановить путь!

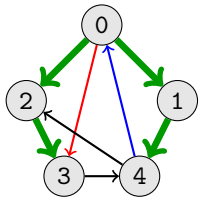
```
bool dfs(int v) {
  if (v == goal) { path.push_back(v); return 1; }
  mark[v] = 1;
  for (int x : g[v])
    if (dfs(x)) { path.push_back(x); return 1; }
  return 0;
}
dfs(start);
```

Вершины пути будут записаны в `path` в порядке от `goal` к `start`.

9.4. Классификация рёбер

После `dfs(v)` остаётся дерево с корнем в v. Отец вершины x – та, из которой мы пришли в x. Пусть все вершины достижимы из v. Рёбра разбились на следующие классы.

- **Древесные:** принадлежат дереву.
- **Прямые:** идут вниз по дереву.
- **Обратные:** идут вверх по дереву.
- **Перекрёстные:** идут между разными поддеревьями.



Рёбра можно классифицировать относительно любого корневого дерева, но именно относительно дерева, полученного `dfs` в неорграфе, нет перекрёстных рёбер.

Lm 9.4.1. Относительно дерева `dfs` **неорграфа** нет перекрёстных рёбер

Доказательство. Если есть перекрёстное ребро $a \to b$, есть и $b \to a$ (граф неориентированный). Пусть $t_{in}[a] < t_{in}[b]$. $a \to b$ перекрёстное $\Rightarrow t_{in}[b] > t_{out}[a]$. Противоречие с тем, что `dfs` пытался пройти по ребру $a \to b$. ■

Времена входа-выхода – стандартные полезные фичи `dfs`-а, считаются так:

```
bool dfs(int v) {
  t_in[v] = T++;
  ...
  t_out[v] = T++;
}
```

9.5. Топологическая сортировка

Def 9.5.1. *Топологической сортировка орграфа* – *сопоставление вершинам номеров* $ind[v]$: $\forall e(a \to b)$ $ind[a] < ind[b]$.

Lm 9.5.2. Топологическая сортировка $\exists$ iff граф ацикличен.

Доказательство. Если есть цикл, рассмотрим неравенства по циклу, получим противоречие. Если циклов нет, $\exists$ вершина v нулевой входящей степени, сопоставим ей $ind[v] = 0$, удалим её из графа, граф остался ацикличным, по индукции пронумеруем оставшиеся вершины. ■

В процессе док-ва получили нерекурсивный алгоритм топологической сортировки за $\mathcal{O}(V+E)$: поддерживаем входящие степени и очередь вершин нулевой входящей степени. Итерация:

```
v = q.pop(); for (int x : g[v]) if (--deg[x] == 0) q.push(x)
```

Алгоритм 9.5.3. *Рекурсивный топсорт*

`dfs` умеет сортировать вершины по времени входа и времени выхода.

```
bool dfs(int v) {
  in_time_order.push_back(v);
  ...
  out_time_order.push_back(v);
}
```

Топологический порядок вершин записан в `reverse(out_time_order)`;
Доказательство: пусть есть ребро $a \to b$, тогда мы сперва выйдем из b и только затем из a.

Лекция #10: Кратчайшие пути

Ноябрь 2021

Def 10.0.1. *Взвешенный граф – каждому ребру соответствует вещественный вес, обычно обозначают w_e (weight) или c_e (cost). Вес пути – сумма весов рёбер.*

Задача SSSP (single-source shortest path problem):
Найти расстояния от выделенной вершины s до всех вершин.

Задача APSP (all pairs shortest path problem):
Найти расстояния между всеми парами вершин, матрицу расстояний.

Обе задачи мы будем решать в ориентированных графах.
Любое решение также будет работать и в неориентированном графе.

10.1. Short description

Приведём результаты, за сколько умеют решать SSSP для разных типов графов:

Задача	Время	Название
ацикличный граф	$V + E$	Динамика
$w_e = 1$	$V + E$	Поиск в ширину
$w_e \geqslant 0$	$V^2 + E$	Dijkstra
$w_e \geqslant 0$	$E \log V$	Dijkstra + bin-heap
$w_e \geqslant 0$	$A^* \leqslant$ Dijkstra	(A^*) A-star
любые веса	VE	Ford-Bellman

Можно решать APSP запуском SSSP-решения от всех вершин. Время получится в V раз больше.
Кроме того, есть несколько алгоритмов специально для APSP:

Задача	Время	Название
любые веса	V^3	Floyd-Warshall
любые веса	$VE + V^2 \log V$	Johnson

К поиску в ширину и алгоритму Флойда применима оптимизация “битовое сжатие”:
$V + E \to \frac{V^2}{w}$, $V^3 \to \frac{V^3}{w}$, где w – размер машинного слова.

Беллман-Форд, Флойд – динамическое программирование.
Дейкстра и A^* – жадные алгоритмы. A^* в худшем случае не лучше алгоритма Дейкстры, но на графах типа “дорожная сеть страны” часто работает за o(размера графа).
Алгоритм и Джонсона основан на не известной нам пока идее потенциалов.

10.2. bfs

Ищем расстояния от s. Расстояние до вершины v обозначим $dist_v$. $A_d = \{v\colon dist_v = d\}$.
Алгоритм: индукция по d.
База: $d = 0, A_0 = \{s\}$.
Переход: мы уже знаем $A_0, \ldots, A_d$, чтобы получить A_{d+1} переберём $N(A_d)$ – соседей A_d, возьмём те из них, что не лежат в $A_0 \cup \cdots \cup A_d$: $A_{d+1} = N(A_d) \setminus (A_0 \cup \cdots \cup A_d)$.
Чтобы за $\mathcal{O}(1)$ проверять, лежит ли вершина v в $A_0 \cup \cdots \cup A_d$, используем $mark_v$.

```
1.  A_0 = {s}
2.  mark ← 0, mark_s = 1
3.  for d = 0..|V|
4.      for v ∈ A_d
5.          for x ∈ neighbors(v)
6.              if mark_x = 0 then
7.                  mark_x = 1, A_{d+1} ← x
```

Время работы $\mathcal{O}(V+E)$, так как в строке 4 каждую v мы переберём ровно один раз для связного графа. Соответсвенно в строке 5 мы по разу переберём каждое ребро.

• Версия с очередью

Обычно никто отдельно не выделяет множества A_d, так как исходная задача всё-таки в том, чтобы найти $dist_v$. Обозначим $q = A_0A_1A_2\ldots$, т.е. выпишем подряд все вершины в том порядке, в котором мы находили до них расстояние. Занумеруем элементы q с нуля. Заодно заметим, что массив `mark` не нужен, так как проверку `mark[x] = 0` можно заменить на `dist[v] = +∞`.

```
1.  q = {s}
2.  dist ← +∞, d_s = 0
3.  for (i = 0; i < |q|; i++)
4.      v = q[i]
5.      for x ∈ neighbors(v)
6.          if dist_x = +∞ then
7.              dist_x = dist_v + 1, q ← x
```

Заметим, что `q` – очередь.

10.3. Модификации bfs

10.3.1. 1-k-bfs

Задача: веса рёбер целые от 1 до k. Найти от одной вершины до всех кратчайшие пути.

Решение раскатерением рёбер: ребро длины k разделим на k рёбер длины 1. В результате число вершин и рёбер увеличилось не более чем в k раз, время работы $\mathcal{O}(k(V+E))$.

Решение несколькими очередями

```
1 for (int d = 0; d < (V-1)k; d++) // (V−1)k = max расстояние
2   for (int x : A[d])
3     if (dist[x] == d)
4       for (Edge e : edges[x])
5         if (dist[e.end] > (D = dist[x] + e.weight))
6           A[D].push_back(e.end), dist[e.end] = D;
```

Этот же код верно работает и для весов 0-k. Правда в таком случае мы можем вставлять новые вершины в список, по которому прямо сейчас итерируемся, поэтому `for (int x : A[d])` некорректно сработает для `vector<int>`. Его следует заменить на `for (size_t i = 0; i < A[d].size(); i++)`.

Lm 10.3.1. Такой bfs работает за $\mathcal{O}(Vk + E)$

Доказательство. Внешний цикл совершает Vk итераций, список смежности каждой вершины смотрим не более одного раза. Суммарное количество вершин во всех списках не более E. ■

Lm 10.3.2. При работе кода выше для 0-k-графа каждая вершина добавляется не более k+1 раза.

Доказательство. Пусть мы впервые добавили вершину v перейдя по ребру из u веса x.
Тогда $d_v = d_u + x$, а значит v лежит в списке d_v.
В какие ещё списки её могут добавить? Только в списки от d_u до $d_u + x$. ■

Замечание 10.3.3. Если убрать из кода выше строчку 3, то он все ещё будет работать верно, разве что список смежности одной вершины может просматриваться k+1 раз, а не 1. Получится время работы $\mathcal{O}(k(V + E))$.

10.3.2. 0-1-bfs

Возьмём обычный `bfs` с одной очередью, заменим очередь на дек. Изменение кода: если перешли по ребру и ответ улучшился, добавим вершину в начало дека.

Алгоритм делает то же самое, что и 0-k `bfs` в случае $k = 1$: мы можем представлять наш дек как две очереди – вершины с расстоянием d, за ними вершины с расстоянием d+1. Значит, и время работы такое же: $\mathcal{O}(k(V+E)) = \mathcal{O}(V+E)$.

10.4. Дейкстра

Алгоритм Дейкстры решает SSSP в графе с неотрицательными весами. Будем от стартовой вершины s идти "вперёд", перебирать вершины в порядке возрастания d_s (кстати, так же делает `bfs`). На каждом шаге берём v: $d_v = \min$ среди всех ещё не рассмотренных v. Прорелаксируем все исходящие из неё ребра, обновим d для всех соседей. Поскольку веса рёбер неотрицательны на любом пути $s = v_1, v_2, v_3, \dots$ величина $d_{v_i}\nearrow$. Алгоритм сперва знает только d_{v_1} и выберет v_1, прорелаксирует d_{v_2}, через некоторое число шагов выберет v_2, прорелаксирует d_{v_3} и т.д.

Формально. Общее состояние алгоритма:
Мы уже нашли расстояния до вершин из множества A, $\forall x \notin A\ d_x = \min_{v \in A}(d_v + w_{vx})$.
Начало алгоритма: $A = \varnothing$, $d_s = 0, \forall v \neq s\ d_v = +\infty$.
Шаг алгоритма: возьмём $v = argmin_{i \notin A} d_i$, прорелаксируем все исходящие из v рёбра.
Утверждение: d_v посчитано верно. Доказательство: рассмотрим кратчайший путь в v, D – длина этого пути, пусть a – последняя вершин из A на пути, пусть b следующая за ней.
Тогда $D \geqslant d_a + w_{ab} \geqslant d_v \Rightarrow d_v$ – длина кратчайшего пути до v.

Время работы Дейкстры = $V \cdot$ `ExtractMin` + $E \cdot$ `DecreaseKey`.

(a) Реализация без куч даст время работы $\Theta(E + V^2)$.
(b) С бинарной кучей даст время работы $\mathcal{O}(E \log V)$. Обычно пишут именно так.

Замечание 10.4.1. Можно запускать Дейкстру и на графах с отрицательными рёбрами. Если разрешить вершину доставать из кучи несколько раз, алгоритм останется корректным, но на некоторых тестах будет работать экспоненциально долго. В среднем, кстати, те же $E \log V$.

10.5. Флойд

Алгоритм Флойда – простое и красивое решение для APSP в графе с отрицательными рёбрами:

```
1 // Изначально d[i,j] = вес ребра между i и j, или ++∞, если ребра нет
2 for (int k = 0; k < n; k++)
3   for (int i = 0; i < n; i++)
4     for (int j = 0; j < n; j++)
5       relax(d[i,j], d[i,k] + d[k,j])
```

На самом деле мы считаем динамику `f[k,i,j]` – минимальный путь из i в j, при условии, что ходить можно только по вершинами $[0, k)$, тогда

`f[k+1,i,j] = min(f[k][i][j], f[k][i][k] + f[k][k][j])`.

Наша реализация использует лишь двумерный массив и чуть другой инвариант: после k шагов внешнего цикла в `d[i,j]` учтены все пути, что и а `f[k,i,j]` и, возможно, какие-то ещё.

Время работы $\mathcal{O}(V^3)$. На современных машинах в секунду получается успеть $V \leqslant 1000$.

10.5.1. Восстановление пути

Также, как в динамике. Если вам понятны эти слова, дальше можно не читать `;-)`

В алгоритмах `bfs`, `Dijkstra`, `A*` при релаксации расстояния `d[v]` достаточно сохранить ссылку на вершину, из которой мы пришли, в `v`. В самом конце, чтобы восстановить путь, нужно от конечной вершины пройти по обратным ссылкам.

Флойд. Способ #1. Можно после релаксации `d[i,j]=d[i,k]+d[k,j]` сохранить ссылку `p[i,j]=k` – промежуточную вершину между i и j. Тогда восстановление ответа – рекурсивная функция: `get(i,j) { k=p[i,j]; if (k != -1) get(i,k), get(k,j); }`

Флойд. Способ #2. А можно хранить `q[i,j]` – первая вершина в пути $i \rightsquigarrow j$. Тогда изначально `q[i,j]=j`. После релаксации `d[i,j]=d[i,k]+d[k,j]` нужно сохранить `q[i,j]=q[i,k]`.

10.5.2. Поиск отрицательного цикла

Что делать Флойду, если в графе есть отрицательный цикл? Хорошо бы хотя бы вернуть информацию о его наличии. Идеально было бы для каждой пары вершин (i, j), если между ними есть кратчайший путь найти его длину, иначе вернуть `IND`.

Lm 10.5.1. ($\exists$ отрицательный цикл, проходящий через i) $\Leftrightarrow$ (по окончании Флойда `d[i,i]<0`).

Lm 10.5.2. ($\nexists$ кратчайшего пути из a в b) $\Leftrightarrow$ ($\exists\, i$ и пути "из a в i" и "из i в b": `d[i,i]<0`).

Замечание 10.5.3. О переполнениях. Веса всех кратчайших путей по модулю не больше $M = (V-1)W$, где W – максимальный модуль веса ребра. Если в графе нет отрицательных циклов, Флойду хватает типа, вмещающего числа из $[-M, M]$. При наличии отрицательных циклов, могут получиться числа меньше $-M$, поэтому будем складывать с корректировкой:

```
1 int sum(int a, int b) {
2   if (a < 0 && b < 0 && a < -M - b)
3     return -M;
4   return a + b;
5 }
```

Как найти хотя бы один отрицательный цикл? Хочется попробовать восстановить путь от i до i (где `d[i,i]<0` также, как любой другой путь. К сожалению, именно так не получится, возможно ссылки зациклятся. Однако если поймать первый момент, когда образуется i, что `d[i,i]<0`, то обычное восстановление ответа сработает.

Лекция #11: Кратчайшие пути

Ноябрь 2021

Мы уже знаем поиск в ширину и алгоритм Дейкстры. Для графов с отрицательными рёбрами у нас пока есть только Флойд, который решает задачу APSP, для SSSP ничего лучше нам не известно. Цель сегодняшней лекции – научиться работать с графами с отрицательными рёбрами. Перед изучением нового попробуем модифицировать старое.

Берём Дейкстру и запускаем её на графах с отрицательными рёбрами. Когда до вершины улучшается расстояние, кладём вершину в кучу. Теперь одна вершина может попасть в кучу несколько раз, но на графах без отрицательных циклов алгоритм всё ещё корректен. В среднем по тестам полученный алгоритм работает $\mathcal{O}(VE)$ и даже быстрее, но $\exists$ тест, на котором время работы экспоненциально. Возможность такой тест придумать будет у вас в дз.

Теперь берём bfs на взвешенном графе с произвольными весами. Вершину кладём в очередь, если до неё улучшилось расстояние. Опять же одна вершина может попасть в очередь несколько раз. На графах без отрицательных циклов алгоритм всё ещё корректен. Оказывается, что мы методом «а попробуем» получили так называем «алгоритм Форд-Беллмана с очередью», который работает за $\mathcal{O}(VE)$, а в среднем по тестам даже линейное время.

Теперь всё по порядку.

11.1. Алгоритм Форд-Беллмана

Решаем SSSP из вершины s.
Насчитаем за $\mathcal{O}(VE)$ динамику `d[k,v]` – минимальный путь из s в v из не более чем k рёбер.
База: `d[0,v] = (v == s ? 0 : +∞)`.
Переход: `d[k+1,v] = min(d[k,v], min`$_x$ `d[k,x] + w[x,v])`, где внутренний минимум перебирает входящие в v рёбра, вес ребра `w[x,v]`. Получили "динамику назад".
Ответ содержится в `d[n-1,v]`, так как кратчайший путь содержит не более $n-1$ ребра.
Запишем псевдокод версии "динамика вперёд".

```
vector<vector<int>> d(n, vector<int>(n, INFINITY));
d[0][s] = 0;
for (int k = 0; k < n - 1; k++)
  d[k+1] = d[k];
  for (Edge e : all_edges_in_graph)
    relax(d[k+1][e.end], d[k][e.start] + e.weight);
```

Соптимизируем память до линейной:

```
vector<int> d(n, INFINITY);
d[s] = 0;
for (int k = 0; k < n - 1; k++) // n−1 итерация
  for (Edge e : all_edges_in_graph)
    relax(d[e.end], d[e.start] + e.weight);
```

После k первых *итераций* внешнего цикла в `d[v]` содержится минимум из некоторого множества путей, в которое входят все пути из не более чем k рёбер $\Rightarrow$ алгоритм всё ещё корректен.
Полученный псевдокод в дальнейшем мы и будем называть алгоритмом Форда-Беллмана.
Его можно воспринимать так "взять массив расстояний d и улучшать, пока улучшается".

11.2. Выделение отрицательного цикла

Изменим алгоритм Форд-Беллмана: сделаем `+1` итерацию внешнего цикла.
$\exists$ отрицательный цикл $\Leftrightarrow$ на последней n-й итерации произошла хотя бы одна релаксация.
Действительно, если нет отрицательного цикла, релаксаций не будет. С другой стороны:

Lm 11.2.1. $\forall$ итерации $\forall$ отрицательного цикла произойдёт релаксация хотя бы по одному ребру.

Доказательство. Обозначим номера вершин $v_1, v_2, \ldots, v_k$ и веса рёбер $w_1, w_2, \ldots, w_k$. Релаксаций не произошло $\Leftrightarrow (d[v_1] + w_1 \geqslant d[v_2]) \wedge (d[v_2] + w_2 \geqslant d[v_3]) \wedge \ldots$. Сложим все неравенства, получим $\sum_i d[v_i] + \sum_i w_i \geqslant \sum_i d[v_i] \Leftrightarrow \sum_i w_i \geqslant 0$. Противоречие с отрицательностью цикла. ■

Осталось этот цикл восстановить. Пусть на n-й итерации произошла релаксация $a \to b$. Для восстановления путей для каждой вершины v мы поддерживаем предка $p[v]$.

• **Алгоритм восстановления:** откатываемся из вершины b по ссылкам p, пока не зациклимся.
Обязательно зациклимся. Полученный цикл обязательно отрицательный.

Lm 11.2.2. $\forall$ момент времени $\forall$ вершины v верно $d[p_v] + w[p_v, v] \leqslant d[v]$.

Доказательство. В момент сразу после релаксации верно верно $d[p_v] + w[p_v, v] = d[v]$.
До следующей релаксации $d[v]$ и p_v не меняются, а $d[p_v]$ может только уменьшаться. ■

Следствие 11.2.3. После релаксации v произошла релаксация $p_v \Rightarrow d[p_v] + w[p_v, v] < d[v]$.

Lm 11.2.4. Откат по ссылкам p из вершины b зациклится.

Доказательство. Пусть не зациклился $\Rightarrow$ остановился в вершине s. При этом $p_s = -1 \Rightarrow d[s]$ не менялось $\Rightarrow d[s] = 0$. Обозначим вершины пути $s = v_1, v_2, \ldots, v_k = b$. Последовательно для всех рёбер пути используем неравенство из леммы 11.2.2, получаем $d[s] + \texttt{(вес пути)} \leqslant d[b] \Leftrightarrow \texttt{(вес пути)} \leqslant d[b]$. Но $d[b]$ уже обновился на n-й итерации, значит строго меньше веса любого пути из не более чем $(n-1)$ ребра. Противоречие. ■

Lm 11.2.5. Вес полученного цикла отрицательный.

Доказательство. Сложим по всем рёбрам цикла неравенство из леммы 11.2.2, получим $\sum d_i + \sum w_i \leqslant \sum d_i \Leftrightarrow \sum w_i \leqslant 0$. Чтобы получить строгую отрицательность рассмотрим y – последнюю вершину цикла, для которой менялось расстояние. Тогда для вершины цикла z: $p_z = y$ неравенство из леммы 11.2.2 строгое. ■

Из доказанных лемм следует:
Теорема 11.2.6. Алгоритм восстановления отрицательного цикла корректен.

11.3. Модификации Форд-Беллмана

Теперь наш алгоритм умеет всё, что должен.
Осталось сделать так, чтобы он работал максимально быстро.

11.3.1. Форд-Беллман с break

Мы уверены, что после $n-1$ итерации массив d меняться перестанет. На случайных тестах это произойдёт гораздо раньше. Оптимизация: делать итерации, пока массив d меняется. Факт: на случайных графах в среднем $\mathcal{O}(\log V)$ итераций $\Rightarrow$ время работы $\mathcal{O}(E \log V)$.

11.3.2. Форд-Беллман с очередью

Сперва заметим, что внутри итерации бесполезно просматривать некоторые рёбра. Ребро $a \to b$ может дать успешную релаксацию, только если на предыдущей итерации поменялось $d[a]$.
Пусть B_k – вершины, расстояние до которых улучшилось на k-й итерации. Псевдокод:

```
1.  d = {+∞,...,+∞}, d[s] = 0
2.  B_0 = {s}
3.  for (k = 0; B_k ≠ ∅; k++)
4.      for v ∈ B_k
5.          for x ∈ neighbors(v)
6.              if d[x] + w[v,x] < d[v] then
7.                  d[v] = d[x] + w[v,x], B_{k+1} ∪= {x}
```

Последние две оптимизации могли только уменьшить число операций в Форд-Беллмане.

• Добавляем очередь

Сделаем примерно то же самое, что с поиском в ширину.
Напомним, bfs мы сперва писали через множества A_d, а затем ввели очередь $q = \{A_0, A_1, \dots\}$.

```
1.  d = {+∞,...,+∞}, d[s] = 0
2.  q = {s}, inQueue[s] = 1
3.  while (!q.empty())
4.      v = q.pop(), inQueue[v] = 0
5.      for x ∈ neighbors(v)
6.          if d[x] + w[v,x] < d[v] then
7.              d[v] = d[x] + w[v,x]
8.              if (!inQueue[v]) inQueue[v] = 1, q.push(v)
```

Алгоритм остался корректным. Докажем, что время работы $\mathcal{O}(VE)$. Пусть в некоторый момент t_k для всех вершин из A_k расстояние посчитано верно. В очереди каждая вершина встречается не более 1 раза $\Rightarrow$ все вершины из очереди мы обработаем за $\mathcal{O}(E)$. После такой обработки корректно посчитаны расстояния до всех вершин из A_{k+1}.

11.4. Потенциалы Джонсона

Чтобы воспользоваться алгоритмом Дейкстры на графах с отрицательными рёбрами, просто сделаем веса неотрицательными...

Def 11.4.1. *Любой массив вещественных чисел p_v можно назвать потенциалами вершин. При этом потенциалы задают новые веса рёбер: $e\colon a \to b, w'_e = w_e + p_a - p_b$.*

Самое важное: любые потенциалы сохраняют кратчайшие пути.

Lm 11.4.2. $\forall s, t$ кратчайший путь на весах w_e перейдёт в кратчайший на весах w'_e

Доказательство. Рассмотрим любой путь $s \rightsquigarrow t\colon s = v_1 \to v_2 \to \dots \to v_k = t$.
Его новый вес равен $w'_{v_1v_2} + w'_{v_2v_3} + \dots = (w_{v_1v_2} + p_{v_1} - p_{v_2}) + (w_{v_2v_3} + p_{v_2} - p_{v_3}) + \dots$
Заметим, что почти все p_v посокращаются, останется $W + p_s - p_{p_t}$, где W – старый вес пути.
То есть, веса всех путей изменились на константу $\Rightarrow$ минимум перешёл в минимум. ■

Осталось дело за малым – подобрать такие p_v, что $\forall e\ w'_e \geqslant 0$.
Для этого внимательно посмотрим на неравенство $w'_e = w_e + p_a - p_b \geqslant 0 \Leftrightarrow p_b \leqslant p_a + w_e$ и поймём, что расстояния d_v, посчитанные от любой вершины s отлично подходят на роль p_v.
Чтобы все v были достижимы из s, введём фиктивную s и из неё во все вершины нулевые рёбра.
Если в исходном графе $\forall e\ w_e \geqslant 0$, получим $\forall v\ d_v = 0$, в любом случае $\forall v\ d_v \leqslant 0$.
Как найти расстояния? Форд-Беллманом.
Можно не добавлять s, а просто начать работу Форд-Беллмана с массива `d = {0, 0, ..., 0}`.

Получается мы свели задачу "поиска расстояний" к "задаче поиска потенциалов", а её обратно к "задаче поиска расстояний". Зачем?

Алгоритм Джонсона решает задачу APSP следующим образом: один раз запускает Форда-Белмана для поиска потенциалов, затем V раз запускает Дейкстру от каждой вершины, получает матрицу расстояний в графе без отрицательных циклов за время $VE + V(E + V \log V) = VE + V^2 \log V$. Заметим, что Форд-Беллман не является узким местом этого алгоритма.

Лекция #12: DSU и MST

Ноябрь 2021

12.1. DSU: Система Непересекающихся Множеств

Цель: придумать структуру данных, поддерживающую операции:

- `init(n)` – всего n элементов, каждый в своём множестве.
- `get(a)` – по элементу узнать уникальный идентификатор множества, в котором лежит a.
- `join(a, b)` – объединить множества элемента a и элемента b.

Для удобства считаем, что элементы занумерованы числами $0 \dots n-1$.

12.1.1. Решения списками

Напишем максимально простую реализацию, получится примерно следующее:

```
1  list<int> sets[n]; // храним множества в явном виде
2  int id[n]; // для каждого элемента храним номер множества
3  void init(int n) {
4    for (int i = 0; i < n; i++)
5      id[i] = i, sets[i] = {i};
6  }
7  int get(int i) {
8    return id[i]; // O(1)
9  }
10 void join(int a, int b) {
11   for (int x : sets[id[b]]) // долгая часть
12     id[x] = id[a];
13   sets[id[a]] += sets[id[b]]; // O(1), += не определён... но было бы удобно =)
14 }
```

Чтобы долгая часть работала быстрее будем "перекрашивать" меньшее множество (у нас же есть выбор!). Для этого перед `for` добавим `if + swap(a, b)`.

Теорема 12.1.1. Тогда суммарное число операций всех `for` во всех `join` не более $n \log n$.

Доказательство. Пусть x сейчас перекрашивается $\Rightarrow$ размер множества, в котором живёт x, увеличился как минимум вдвое. Для каждого x произойдёт не более $\log_2 n$ таких событий. ■

Следствие 12.1.2. Суммарное время работы m `join`-ов $\mathcal{O}(m + n \log n)$.

12.1.2. Решения деревьями

Каждое множество – дерево. Отец корня дерева – он сам. Функция `get` вернёт корень дерева. Функция `join` соединит корни деревьев. Для ускорения используют две эвристики:

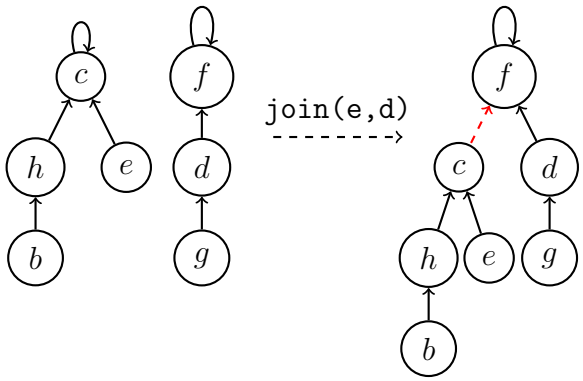
• **Ранговая сжатие путей:** все рёбра, по которым пройдёт `get` перенаправить в корень.

• **Ранговая эвристика:** подвешивать меньшее к большему. Можно выбирать меньшее по глубине, *рангу*, размеру. Эти идеи дают идентичный результат.

Def 12.1.3. *Ранг вершины – глубина её поддерева, если бы не было сжатия путей. Сжатие путей не меняет ранги. Ранг листа равен нулю.*

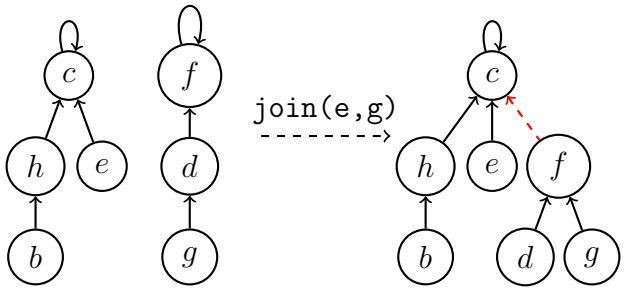
Вот базовая реализация без ранговой эвристики и без сжатия путей:

```
int p[n]; // для каждого элемента храним отца
void init(int n) {
  for (int i = 0; i < n; i++)
    p[i] = i; // каждый элемент - сам себе корень
}
int get(int i) {
  // подняться до корня, вернуть корень
  return p[i] == i ? i : get(p[i]);
}
void join(int a, int b) {
  a = get(a), b = get(b); // перешли к корням
  p[a] = b; // подвесили один корень за другой
}
```



А вот версия и с ранговой эвристикой и со сжатием путей:

```
int p[n], rank[n];
void init(int n) {
  for (int i = 0; i < n; i++)
    p[i] = i, rank[i] = 0;
}
int get(int i) {
  return p[i] == i ? i : (p[i] = get(p[i]));
}
void join(int a, int b) {
  a = get(a), b = get(b);
  if (not (rank[a] <= rank[b])) swap(a, b);
  if (rank[a] == rank[b]) rank[b]++;
  p[a] = b; // a - вершина с меньшим рангом
}
```



Время работы `join` во всех версиях равно `Time(get) + 1`. Теперь оценим время работы `get`.

Теорема 12.1.4. Ранговая эвристика без сжатия путей даст $\mathcal{O}(\log n)$ на запрос.

Для доказательства заметим, что глубина не больше ранга, а ранг не больше $\log_2 n$, так как:

Lm 12.1.5. Размер дерева ранга k хотя бы 2^k.

Доказательство. Индукция. База: для нулевого ранга имеем ровно одну вершину.
Переход: чтобы получить дерево ранга $k+1$ нужно два дерева ранга k. ■

Напомним пару определений:

Def 12.1.6. *Пусть v – вершина, p – её отец, size – размер поддерева.*

Ребро $p \to v$ называется

тяжёлым	*если* $size(v) > \frac{1}{2} size(p)$
лёгким	*если* $size(v) \leqslant \frac{1}{2} size(p)$

Теорема 12.1.7. Сжатие путей без ранговой эвристики даст следующую оценку:
m запросов `get` в сумме пройдут не более $m + (m+n)\log_2 n$ рёбер.

Доказательство. $\forall v$, поднимаясь вверх от v, мы пройдём не более $\log_2 n$ лёгких рёбер, так как при подъёме по лёгкому ребру размер поддерева хотя бы удваивается. Рассмотрим любое тяжёлое ребро $(v \to p)$ кроме самого верхнего. Сжатие путей отрезает v у p, это уменьшает размер поддерева p хотя бы в два раза, что для каждого p случится не более $\log_2 n$ раз. ■

12.1.3. (*)Оценка $\mathcal{O}(\log^* n)$

Теорема 12.1.8. Сжатие путей вместе с ранговой эвристикой дадут следующую оценку: m запросов `get` в сумме пройдут не более $m(1 + 2\log^* n) + \Theta(n)$ рёбер.

Доказательство. Обозначим $x = 1.7$, заметим $x^x \geqslant 2$. Назовём ребро $v \to p$ крутым, если $rank[p] \geqslant x^{rank[v]}$ (при подъёме по этому ребру ранг сильно возрастает). Обозначим время работы i-го `get`, как $t_i = 1 + a_i + b_i =$ одно верхнее ребро + a_i крутых + b_i не крутых.

$\forall v$ имеем $rank[p_v] > rank[v] \Rightarrow a_i \leqslant 2\log^* n$. Теперь исследуем жизненный цикл не крутых рёбер. $\forall v$, если v не корень, $rank[v]$ уже не будет меняться. При сжатии путей у всех рёбер кроме самого верхнего возрастёт разность $(rank[p_v] - rank[v]) \Rightarrow$ каждое некрутое ребро уже через $x^{rank[v]}$ проходов по нему вверх навсегда станет крутым $\Rightarrow \sum_i b_i \leqslant \sum_v x^{rank[v]} = \sum_r count(r)x^r$, где $count(r)$ – число вершин с рангом r. Вершины с рангом r имеют поддеревья размера $\geqslant 2^r$, эти поддеревья не пересекаются $\Rightarrow count(r) \leqslant \frac{n}{2^r} \Rightarrow \sum_i b_i \leqslant \sum_r \frac{n}{2^r}x^r = n\sum_r \left(\frac{x}{2}\right)^r = \Theta(n)$. ■

Теорема 12.1.9. Сжатие путей вместе с ранговой эвристики дадут следующую оценку: m запросов `get` в сумме пройдут не более $\Theta(m + n\log^* n)$ рёбер.

Доказательство. Крутое ребро, которое k раз поучаствовало в сжатии путей в качестве не самого верхнего крутого, назовём *ребром крутизны* k. Крутизна любого ребра не более $\log^* n$. При сжатии пути, крутизны всех крутых кроме самого верхнего растут $\Rightarrow \sum a_i \leqslant n\log^* n$. ■

12.2. (*)Оценка $\mathcal{O}(\alpha^{-1}(n))$

12.2.1. (*)Интуиция и $\log^{**} n$

Теорема 12.2.1. Сжатие путей вместе с ранговой эвристикой дадут следующую оценку: m запросов `get` в сумме пройдут не более $\Theta(m(1 + \log^{**} n) + n)$ рёбер.

Доказательство. Ребро $(v \to p)$ крутизны хотя бы $rank[v]$ назовём *дважды крутым*.
Если обычное ребро $x^{rank[v]}$ раз поучаствует в сжатии, оно станет крутым $\Rightarrow$
проходов по не крутым рёбрам $\sum_v x^{rank[v]} = \Theta(n)$.
Если крутое ребро $rank[v]$ раз поучаствует в сжатии, оно станет дважды крутым $\Rightarrow$
проходов по не дважды крутым рёбрам $\sum_v rank[v] \leqslant 2n + \sum_v x^{rank[v]} = \Theta(n)$.
Осталось оценить число дважды крутых рёбер на пути. Для дважды крутого ребра имеем:

$$rank[p_v] \geqslant x^{x^{x^{\cdots^{rank[v]}}}}$$

Высота степенной башни – $rank[v]$, поэтому $(rank[v] > 2\log^* n \Rightarrow rank[p_v] > n) \Rightarrow$ на любом пути вниз $\mathcal{O}(\log^{**} n)$ дважды крутых рёбер (проход по такому ребру меняет ранг r на $2\log^* r$). ■

Замечание 12.2.2. Аналогично можно рассмотреть *трижды крутые* и даже k-раз крутые рёбра. Тогда суммарное время всех `get`-ов будет $\Theta((m + n)k + m\log^{**\cdots*} n)$.

12.2.2. (*)Введение обратной функции Аккермана

Def 12.2.3. *Функция Аккермана*
$\begin{cases} A_0(n) = n+1 \\ A_k(n) = A_{k-1}^{n+1}(n) = A_{k-1}(A_{k-1}(\dots(n))) \text{ (взять функцию } n+1 \text{ раз)} \end{cases}$

Lm 12.2.4. $A_k(n) > n$

Lm 12.2.5. $f(t) = A_t(1)$ монотонно возрастает

Доказательство. $A_{t+1}(1) = A_t(A_t(1)) = A_t(x) > A_t(1)$, так как $x > 1$. ■

Выпишем несколько первых функций явно:
$A_0(t) = t+1$
$A_1(t) = A_0^{(t+1)}(t) = 2t+1$
$A_2(t) = A_1^{(t+1)}(t) = 2^{t+1}(t+1) - 1 \geqslant 2^t$ (последнее равенство доказывает по индукции)
$A_3(t) = A_2^{(t+1)}(t) \geqslant A_2^t(2^t) \geqslant \dots \geqslant 2^{2^{2^{\dots^t}}}$ (высота башни $t+1$).

Def 12.2.6. *Обратная функция Аккермана* – $\alpha(t) = \min\{k \mid A_k(1) \geqslant t\}$

Посчитаем несколько первых значений функции α
$A_0(1) = 1+1 = 2$
$A_1(1) = 2 \cdot 1 + 1 = 3$
$A_2(1) = 2^2 * 2 - 1 = 7$
$A_3(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2047$
$A_4(1) = A_3(A_3(1)) = A_3(2047) > 2^{2^{2^{\dots^{2047}}}}$ (башня высоты 2048).

Получившаяся оценка снизу на $A_4(1)$ – невероятно большое число, превышающее число атомов в наблюдаемой части Вселенной. Поэтому обычно предполагают, что $\alpha(t) \leqslant 4$.

12.2.3. (*)Доказательство

Для краткости записей ранг вершины e СНМ будем обозначать r_e, а родителя e в СНМ p_e.
У каждого множества в СНМ есть ровно один корень (представитель множества).
Элемент, который не является корнем, будем называть обычным.

Мы уже знаем, что $\forall e\colon e \neq p_e\ r_{p_e} > r_e$. Интересно, на сколько именно больше:

Def 12.2.7. *Порядок элемента.* $\forall$ *обычного элемента* e $\quad level(e) = \max\{k \mid r_{p_e} \geqslant A_k(r_e)\}$.

Def 12.2.8. *Итерация порядка.* $\forall$ *обычного элемента* e $\quad iter(e) = \max\{i \mid r_{p_e} \geqslant A_{level(e)}^{(i)}(r_e)\}$.

Получили пару $\langle level(e), iter(e)\rangle$. Чем больше пара, тем у e больше разница с рангом отца.

Lm 12.2.9. $\forall$ обычного элемента e верно, что $0 \leqslant level(e) < \alpha(n)$.

Доказательство. $level(e) \geqslant 0$, так как $A_0(r_e) = r_e + 1 \leqslant r_{p_e}$.
$n > r_{p_e} \geqslant A_{level(e)}(r_e) \geqslant A_{level(e)}(1) \Rightarrow n > A_{level(e)}(1) \Rightarrow level(e) < \alpha(n)$.

■

Lm 12.2.10. $\forall$ обычного элемента e верно, что $1 \leqslant iter(e) \leqslant r_e$.

Доказательство. $r_{p_e} \geq A_{level(e)}(r_e) \Rightarrow iter(e) \geq 1$.
$r_{p_e} < A_{level(e)+1}(r_e) = A^{(r_e+1)}_{level(e)}(r_e) \Rightarrow iter(e) < r_e + 1 \Leftrightarrow iter(e) \leq r_e$. ■

Будем доказывать время работы СНМ методом амортизационного анализа.

Def 12.2.11. *Потенциал элемента. Определим для любого элемента e величину $\varphi(e)$.*

$$\varphi(e) = \begin{cases} \alpha(n) \cdot r_e & \text{если } e \text{ – представитель своего множества} \\ (\alpha(n) - level(e)) \cdot r_e - iter(e) & \text{если } e \text{ – обычный элемент} \end{cases}$$

Def 12.2.12. *Потенциал. Возъмём $\varphi = \sum_e \varphi(e)$.*

Обозначим через φ_t потенциал после первых t операций.

Lm 12.2.13. $\varphi_0 = 0$, $\forall i\ \varphi_i \geqslant 0$.

Доказательство. Изначально все элементы – представители, и все $r_e = 0$. Далее все $r_e \geqslant 0$, появляются обычные элементы. Используем 12.2.9 и 12.2.10, получаем неотрицательность потенциалов обычных элементов. ■

Lm 12.2.14. Если у обычной вершины v увеличить ранг отца, то $\varphi(v) \searrow$.

Теорема 12.2.15. Для операции типа *join* амортизированное время $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$.

Доказательство. Пусть мы присвоили $p_a = b$. Тогда потенциал мог измениться только у b, a и детей a. Итого $\Delta\varphi = \left(\sum_{c \in children(a)} \Delta\varphi(c)\right) + \Delta\varphi(b) + \Delta\varphi(a)$. Первые два слагаемых неположительны, последнее равно $\alpha(n)$. ■

Теорема 12.2.16. Для операции типа *get* амортизированное время $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$.

Доказательство. После сжатия путей по 12.2.14 потенциалы вершин не могли увеличиться. Пусть i-ая операция *get* проходит $w_i + x_i + 2$ вершин. Здесь w_i – число вершин v: $\Delta\varphi(v) < 0$, x_i – число, вершин v: $\Delta\varphi(v) = 0$, но $r_{p_v} \uparrow$, а 2 – корень и его сын. Если у вершины v не меняется потенциал, не меняются и $level(v), iter(v)$.

Осталось показать, что $x_i \leqslant \alpha(n)$. От противного: $(x_i > \alpha(n)) \wedge$ (по 12.2.9 $\exists$ не более $\alpha(n)$ различных значений $level$) $\Rightarrow \exists$ две вершины a и b на пути: $\Delta\varphi(a) = \Delta\varphi(b) = 0$, $level(a) = level(b)$. После сжатия путей у нижней из $\{a, b\}$ как минимум изменится $iter$. Противоречие. ■

Теорема 12.2.17. Суммарное время работы m произвольных операций с СНМ – $\mathcal{O}(m \cdot \alpha(n))$.

Доказательство. $\sum\limits_i t_i = \sum\limits_i a_i - \varphi_m + \varphi_0$ ($\varphi_0 = 0$, $\varphi_m \geqslant 0$) $\Rightarrow \sum\limits_i t_i \leqslant \sum\limits_i a_i = m \cdot \alpha(n)$. ■

12.3. MST: Минимальное Остовное Дерево

Def 12.3.1. *Остовное дерево связного графа – подмножество его рёбер, являющееся деревом.*

Def 12.3.2. *Вес остова – сумма весов рёбер.*

Задача: дан граф, найти остов минимальной стоимости.

Задача поиска максимального остова равносильна данной (домножение весов на -1).

12.3.1. Алгоритм Краскала

Начнём с пустого остова. Будем перебирать рёбра в порядке возрастания веса.
Если текущее ребро при добавлении в остов не образует циклов, добавим ребро в остов.
Проверку на циклы будем делать СНМ-ом. Время работы алгоритма: $\mathcal{O}(sort(E) + (V+E)\alpha)$.

12.3.2. Алгоритм Прима

Поддерживаем остовное дерево множества вершин A. База: $A = \{0\}$, рёбер в остове нет. Для перехода $\forall v$ поддерживаем $d_v = \min_{A \xrightarrow{e} v} \langle w_e, e \rangle$. Переход: выбрать $v \notin A$ $d_v = \min$, подсоединить v к остову за ребро d_v`.second`. Чтобы быстро выбирать вершину v, поддерживаем кучу всех d_v.

Время работы $= E \cdot DecreaseKey + V \cdot ExtractMin$, то есть, $\mathcal{O}(E + V \log V)$ и $\mathcal{O}(V^2)$.

Замечание 12.3.3. Алгоритм Прима – Дейкстра с операцией “max” вместо “+”.

12.3.3. Сравнение алгоритмов

- Краскал просто пишется, быстро работает.
- На очень плотных графах быстрее V^2 через Прима.

12.3.4. Лемма о разрезе и доказательства

Доказательства трёх алгоритмов очень похожи, все они опираются на *лемму о разрезе*:

Lm 12.3.4. Для любого разбиения множества вершин $V = A \sqcup B$, существует минимальный остов, содержащий e – минимальное по весу ребро, проходящее через разрез $\langle A, B \rangle$.

Доказательство. Возьмём минимальный остов без e. Добавим e, получится цикл.
Два ребра этого цикла проходят через разрез $\langle A, B \rangle$. Старое неменьше e, удалим его. ■

Если же в процессе построения min остова уже известно подмножество рёбер будущего min остова, эти рёбра задают компоненты связности. Мы можем сжать компоненты в вершины и для конденсации применить лемму о разрезе. Итого:

Следствие 12.3.5. X – подмножество рёбер некого минимального остова. Зафиксируем разрез $V = A \sqcup B$ такой, что все рёбра X не пересекают разрез. Тогда $\exists$ минимальный остов, содержащий $X \cup \{e\}$, где e – минимальное по весу ребро, проходящее через разрез $\langle A, B \rangle$.

Для доказательства описанных выше алгоритмов нужно лишь применить правильный разрез.

• Доказательство алгоритма Прима.

Разрез – текущее множество A и дополнение $V \setminus A$.

• Доказательство алгоритма Краскала.

Добавляем ребро (a, b). Разрез – любой такой, что (a, b) через него проходит.

Лекция #13: Базовые алгоритмы на строках

Ноябрь 2021

13.1. Обозначения, определения

s, t – строки, $|s|$ – длина строки, $\overline{s}$ – перевёрнутая s,
`s[l:r)` и `s[l:r]` – подстроки,
`s[0:i)` – префикс, `s[i:|s|-1]` = `s[i:]` – суффикс.
Σ – алфавит, $|\Sigma|$ – размер алфавита.
Говорят, что s – подстрока t, если $\exists\, l, r\colon s = t[l{:}r)$.

13.2. Поиск подстроки в строке

Даны текст t и строка s. Ещё иногда говорят "*строка (string) t и образец (pattern) s*".
Вхождением s в t назовём позицию i: `s` = `t[i:i+|s|)`.
Возможны различные формулировки задачи поиска подстроки в строке:

(a) Проверить, есть ли хотя бы одно вхождение s в t.
(b) Найти количество вхождений s в t.
(c) Найти позицию любого вхождения s в t, или вернуть -1, если таких нет.
(d) Вернуть множества всех вхождений s в t.

13.2.1. C++

В языке `C++` у строк типа `string` есть стандартный метод `find`. Работает за $\mathcal{O}(|s| \cdot |t|)$, возвращает целое число – номер позиции в исходной строке, начиная с которого начинается первое вхождение подстроки или `string::npos`.

Функция из `<cstring>` `strstr(t, s)` ищет s в t. Работает за линию в `Unix`, за квадрат в `Windows`.

В обоих случаях квадрат имеет очень маленькую константу (AVX-регистры).

Все вхождения можно перечислить таким циклом:

```
for (size_t pos = t.find(s); pos != string::npos; pos = st.find(s, pos + 1))
  ; // pos - позиция вхождения
```

или таким

```
for (char *p = t; (p = strstr(p, s)) != 0; p++)
  ; // p - указатель на позицию вхождения в t
```

13.2.2. Префикс функция и алгоритм КМП

Def 13.2.1. $\pi_0(s)$ – *длина* max *собственного префикса s, совпадающего с суффиксом s.*

Def 13.2.2. *Префикс-функция строки* `s` – *массив* $\pi(s)\colon \pi(s)[i] = \pi_0\big(s[0{:}i)\big)$.

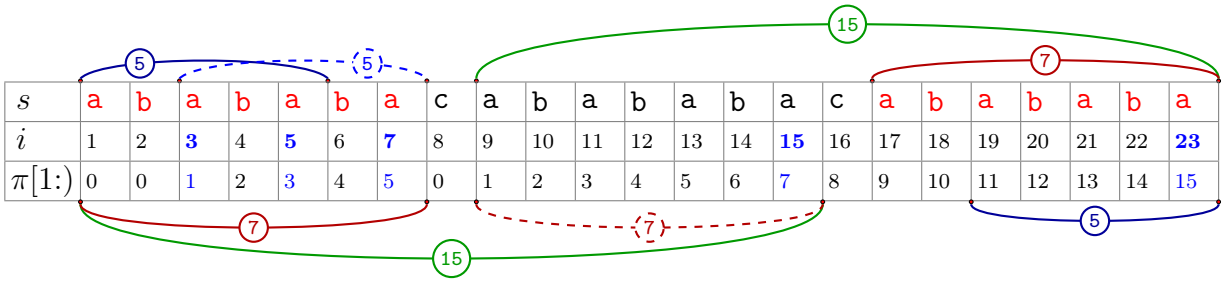
Когда из контекста понятно, о префикс-функции какой строки идёт речь, пишут просто $\pi[i]$.

• Алгоритм Кнута-Мориса-Пратта.

Пусть '#' – любой символ, который не встречается ни в t, ни в s. Создадим новую строку $w = s\#t$ и найдем её префикс-функцию. Благодаря символу '#' $\forall i\ \pi(w)[i] \leqslant |s|$. Такие i, что $\pi(w)[i] = |s|$, задают позиции окончания вхождений s в $w \Rightarrow (j = i - 2|s| - 1)$ – начало вхождения s в t.

• **Вычисление префикс-функции за $\mathcal{O}(n)$.**

Все префиксы равные суффиксам для строки `s[0:i)` это $X = \{i, \pi[i], \pi[\pi[i]], \pi[\pi[\pi[i]]], \dots\}$



s	a	b	a	b	a	b	a	c	a	b	a	b	a	b	a	c	a	b	a	b	a	b	a
i	1	2	**3**	4	**5**	6	**7**	8	9	10	11	12	13	14	**15**	16	17	18	19	20	21	22	**23**
$\pi[1:)$	0	0	1	2	3	4	5	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Пример (см. картинку выше): для всей строки s имеем $X = \{23, 15, 7, 5, 3, 1\}$.

Заметим, что или $\pi[i{+}1] = 0$, или $\pi[i{+}1]$ получается, как $x + 1$ для некоторого $x \in X$.
Будем перебирать $x \in X$ в порядке убывания, получается следующий код:

```
1 p[1] = 0, n = |s|
2 for (i = 2; i <= n; i++)
3   int k = p[i - 1];
4   while (k > 0 && s[k] != s[i - 1])
5     k = p[k];
6   if (s[k] == s[i - 1])
7     k++;
8   p[i] = k;
```

Заметим, что с учётом последней строки цикла первая не нужна, получаем:

```
1 p[1] = k = 0, n = |s|
2 for (i = 2; i <= n; i++)
3   while (k > 0 && s[k] != s[i - 1])
4     k = p[k];
5   if (s[k] == s[i - 1])
6     k++;
7   p[i] = k;
```

Вычисление префикс функции работает за $\mathcal{O}(n)$, так как k увеличится $\leqslant n$ раз $\Rightarrow$ суммарное число шагов цикла `while` не более n.

Собственно КМП работает за $\mathcal{O}(|s| + |t|)$ и требует $\mathcal{O}(|s| + |t|)$ дополнительной памяти.
Упражнение: придумайте, как уменьшить количество доппамяти до $\mathcal{O}(|s|)$.

13.2.3. LCP

Def 13.2.3. *`lcp[i,j]` (largest common prefix) для строки s – длина наибольшего общего префикса суффиксов `s[i:]` и `s[j:]`.*

Вычислить массив `lcp` можно за $\mathcal{O}(n^2)$, так как $\texttt{lcp[i,j]} = \begin{cases} 1 + \texttt{lcp[i+1,j+1]} & s[i] = s[j] \\ 0 & \text{иначе} \end{cases}$

Аналогично можно определить и вычислить массив `lcp` для двух разных строк.

13.2.4. Z-функция

Def 13.2.4. *Z-функция – массив z такой, что $z[0] = 0, \forall i{>}0\ z[i] =$ `lcp[0,i]`.*

Для поиска подстроки снова введем $w = s\#t$ и посчитаем $Z(w)$.
Найдем все позиции i: $Z(w)[i] = |s|$. Это позиции всех вхождений строки s в строку t.
Осталось научиться вычислять Z-функцию за линейное время.

```
1 z[0] = 0;
2 for (i = 1; i < n; i++)
3   int k = 0;
4   while (s[i + k] == s[k]) // s[n] == '\x0' ⇒ нет выхода за пределы!
5     k++;
6   z[i] = k;
```

Приведенный алгоритм работает за $\mathcal{O}(n^2)$. На строке $aaa\ldots a$ оценка n^2 достигается.
Ключом к ускорению является следующая лемма:

Lm 13.2.5. $\forall l < i < l + z[l] = r$ имеем `s[0:z[l])` = `s[l:r)` и `s[i-l:z[l])` = `s[i:r)`.

Следствие леммы: $z[i] \geqslant \min(r - i, z[i - l])$. Логично взять l: $r = l + z[l] = \max$.
Немного модифицируем код, чтобы получить асимптотику $\mathcal{O}(n)$.

```
1 z[0] = 0, l = r = 0;
2 for (i = 1; i < n; i++)
3   int k = max(0, min(r - i, z[i - l]))
4   while (s[i + k] == s[k])
5     k++
6   z[i] = k
7   if (i + z[i] > r) l = i, r = i + z[i]
```

Теорема 13.2.6. Приведенный выше алгоритм работает за $\mathcal{O}(n)$.

Доказательство. `k++` $\Rightarrow$ `r++`, а r может увеличиваться $\leqslant n$ раз. ■

Лекция #14: Хеширование

Ноябрь 2021

14.1. Полиномиальные хеши строк

Основная идея этой секции – научиться с предподсчётом за $\mathcal{O}$(суммарной длины строк) вероятностно сравнивать на равенство любые их подстроки за $\mathcal{O}(1)$.
Например, мы уже умеем считать частичные суммы за $\mathcal{O}(1) \Rightarrow$ можем за $\mathcal{O}(1)$ проверить, равны ли суммы символов в подстроках. Если не равны $\Rightarrow$ строки точно не равны...

Def 14.1.1. *Хеш-функция объектов из мн-ва A в диапазон $[0, m)$ – любая функция $A \to \mathbb{Z}/m\mathbb{Z}$.*

Например, сумма символов строки, посчитанная по модулю 256 – пример хеш-фукнции из множества строк в диапазон $[0, 256)$. Задача – придумать более удачную хеш-функцию.

Зачем нужны хеши (хеш-функции)? Чтобы сравнивать объекты на равенство.

Хеши не совпали $\Rightarrow$ объекты точно не совпали.
Хеши совпали $\Rightarrow$ с некоторой вероятностью объекты всё равно различаются (коллизия хешей) и у нас есть выбор — или остатоновиться и получить RP алгоритм, или после равенства хешей сравнить сами объекты и получить ZPP алгоритм.

• Полиномиальная хеш-функция

Def 14.1.2. *Пусть $s = s_0 s_1 \dots s_{n-1} \Rightarrow h_{p,m}(s) = (s_0 p^{n-1} + s_1 p^{n-2} + \dots + s_{n-1}) \bmod m$*

$h_{p,m}(s)$ – полиномиальный хеш для строки s посчитанный в точке p по модулю m.

По сути мы взяли многочлен (полином) с коэффициентами «символы строки» и посчитали его значение в точке p по модулю m. Можно было бы определить $h_{p,m}(s) = \sum_i s_i p^i$, но при реализации нам будет удобен порядок суммирования из 14.1.2.

```
1  int *h;
2  void initialize(int n, char* s) {
3    h = new int[n + 1]; // h[i] - хеш префикса s[0:i)
4    h[0] = 0; // хеш пустой строки действительно 0...
5    for (int i = 0; i < n; i++)
6      h[i + 1] = ((int64_t)h[i] * p + s[i]) % m; // 0 < m < 2^31
7  }
8  int getHash(int l, int r, char* s) { // [l,r)
9    // deg[r - l] = p^(r-l), никогда не пишите здесь лишний if
10   T res = (h[r] - (int64_t)h[l] * deg[r - l]) % m;
11   return res < 0 ? res + m : res; // остаток мог быть отрицательным
12 }
```

Чтобы вероятность коллизии была мала нужно:
m – простое большое число.
p – заранее фиксированное случайное число.

Стандартные ошибки:

(a) Символы строки должны быть >0, иначе $\texttt{hash}(00) = \texttt{hash}(0)$.
(b) Переполнения! Например, при подсчёте p^k.
(c) Остаток может быть отрицательными: $(h_r - h_l p^{r-l}) \bmod m$

(d) ± 1 во всех формулах. В коде выше полуинтервал $[l, r)$, индексация с нуля.

Если вы хотите делать вычисления по более большому модулю $\approx 10^{18}$, у вас два пути – или считать два хеша по двум модулям 10^9+7, 10^9+9, или использовать тип `__int128`.

Ещё есть вариант – считать по модулю 2^{32} или 2^{64}, тогда можно везде писать тип `uint_32/uint_64`, и не делать лишних взять по модулю. Константа станет меньше, а вот работать будет не всегда (ниже подробно разберёмся).

Def 14.1.3. *Коллизия хешей – ситуация вида* $s \neq t,\ h_{p,m}(s) = h_{p,m}(t)$.

Займёмся точными оценками чуть позже, пока предположим, что $\forall$ простого m, если мы выбираем p равновероятно в $[0, m)$, вероятность коллизии при одном сравнении равна $\frac{1}{m}$.
Из умения сравнивать строки на равенство за $\mathcal{O}(1)$ следует алгоритм поиска строки в тексте:

14.1.1. Алгоритм Рабина-Карпа

Можно искать s в t, предподсчитав полиномиальные хеши для t и для каждого потенциального вхождения $[i, i+|s|)$ сравнить за $\mathcal{O}(1)$ хеш подстроки $t[i, i+|s|)$ с хешом s.
Если хеши совпали, то возможно два развития событий: мы можем
или проверить за линию равенство строк, или, не проверяя, выдать вхождение i.

Для задачи поиска одного вхождения в первом случае мы получили `RP`, во втором `ZPP`.
На практике используют оба подхода.
Для задачи поиска всех вхождений проверять каждое вхождение за линию – слишком долго.

• **Оптимизируем память.**

Преимущество Рабина-Карпа над π-функцией и Z-функцией – реализация с $\mathcal{O}(1)$ доппамяти.
Обозначим $n = |s|$ и $ht_i = h_{p,m}(t[i : i+n))$. Посчитаем $h_{p,m}(s)$, ht_0 и p^n.
Осталось, зная ht_i, научиться считать $ht_{i+1} = ht_i \cdot p - t_i \cdot p^n$.

14.1.2. Наибольшая общая подстрока за $\mathcal{O}(n \log n)$

Задача: даны строки s и t, найти w: w – подстрока s, подстрока t, $|w| \to \max$.
Заметим, что если w – общая подстрока s и t, то любой её префикс тоже $\Rightarrow$
функция $f(k) =$ "есть ли у s и t общая подстрока длины k" – монотонный предикат $\Rightarrow$
максимальное k: $f(k) = 1$ можно найти бинпоиском ($\mathcal{O}(\log \min(|s|, |t|)$ итераций).

Осталось для фиксированного k за $\mathcal{O}(|s| + |t|)$ проверить, есть ли у s и t общая подстрока длины k. Сложим хеши всех подстрок s длины k в хеш-таблицу. Для каждой подстроки t длины k поищем её хеш в хеш-таблице. Если нашли совпадение, как и в Рабине-Карпе есть два пути – проверить за линию или сразу поверить в совпадение. Оба метода работают.

14.2. Хеш-таблица

Задача: изначально есть пустое множество целых чисел хотим уметь быстро делать много операций вида добавить элемент, удалить элемент, проверить наличие элемента.

Медленное решение: храним множество в векторе,
`add` = `push_back` = $\mathcal{O}(1)$, `find` = $\mathcal{O}(n)$, `del` = `find` + $\mathcal{O}(1)$ (swap с последним и `pop_back`).

Простое решение: если элементы множества от 0 до 10^6, заведём массив `is`$[10^6+1]$.

`is[x]` = есть ли элемент `x` в множестве. Все операции за $\mathcal{O}(1)$.

Решение: хеш-таблица – структура данных, умеющая делать операции `add`, `del`, `find` за рандомизированное $\mathcal{O}(1)$.

14.2.1. Хеш-таблица на списках

```
list<int> h[N]; // собственно хеш-таблица
void add(int x) { h[x % N].push_back(x); } // O(1) в худшем
auto find(int x) { return find(h[x % N].begin(), h[x % N].end(), x); }
// find работает за длину списка
void erase(int x) { h[x % N].erase(find(x)); } // работает за find + O(1)
```

Вместо `list` можно использовать любую структуру данных, `vector`, или даже хеш-таблицу.

Если в хеш-таблице живёт n элементов и они равномерно распределены по спискам, в каждом списке $\frac{n}{N}$ элементов $\Rightarrow$ при $n \leqslant N$ и равномерном распределении элементов, все операции работают за $\mathcal{O}(1)$. Как сделать распределение равномерным? Подобрать хорошую хеш-функцию!

Утверждение 14.2.1. Если N простое, то хеш-функция `x` $\to$ `x % N` достаточно хорошая.

Без доказательства.

Если добавлять в хеш-таблицу новые элементы, со временем n станет больше N.
В этот момент нужно перевыделить память $N \to 2N$ и передобавить все элементы на новое место. Возьмём $\varphi = -N \Rightarrow$ амортизированное время удвоения $\mathcal{O}(1)$.

14.2.2. Хеш-таблица с открытой адресацией

Реализуется на одном циклическом массиве. Хеш-функция используется, чтобы получить начальное значение ячейки. Далее двигаемся вправо, пока не найдём ячейку, в которой живёт наш элемент или свободную ячейку, куда можно его поселить.

```
int h[N]; // собственно хеш-таблица
// h[i] = 0 : пустая ячейка
// h[i] = -1 : удалённый элемент
// h[i] > 0 : лежит что-то полезное
int getIndex(int x): // поиск индекса по элементу, требуем x > 0
  int i = x % N; // используем хеш-функцию
  while (h[i] && h[i] != x)
    if (++i == N) // массив циклический
      i = 0;
  return i;
```

1. **Добавление:** `h[getIndex(x)] = x;`
2. **Удаление:** `h[getIndex(x)] = -1;`, нужно потребовать `x != -1`, ячейка не становится свободной.
3. **Поиск:** `return h[getIndex(x)] != 0;`

Lm 14.2.2. Если в хеш-таблице с открытой адресацией размера N занято αN ячеек, $\alpha < 1$, матожидание время работы `getIndex` не более $\frac{1}{1-\alpha}$.

Доказательство. Худший случай – `x` отсутствует в хеш-таблице. Без доказательства предположим, что свободные ячейки при хорошей хеш-функции расположены равномерно.
Тогда на каждой итерации цикла `while` вероятность «**не** остановки» равна α.

Вероятность того, что мы не остановимся и после k шагов равна α^k, то есть, сделаем k-й шаг (не

ровно k шагов, а именно k-й!). Время работы = матожидание числа шагов = $1+\sum_{k=1}^{\infty}$ (вероятность того, что мы сделали k-й шаг) = $1+\alpha+\alpha^2+\alpha^3+\dots=\frac{1}{1-\alpha}$. ■

• Переполнение хеш-таблицы

При слишком большом α операции с хеш-таблицей начинают работать долго.
При $\alpha=1$ (нет свободных ячеек), `getIndex` будет бесконечно искать свободную. Что делать?
При $\alpha>\frac{2}{3}$ удваивать размер и за линейное время передобавлять все элементы в новую таблицу.

При копировании, конечно, пропустим все -1 (уже удалённые ячейки) $\Rightarrow$ удалённые ячейки занимают лишнюю память ровно до ближайшего перевыделения памяти.

14.2.3. Сравнение

У нас есть два варианта хеш-таблицы. Давайте сравним.

Пусть мы храним x байт на объект и 8 на указатель, тогда хеш-таблицы используют:

- Списки (если ровно n списков): $8n+n(x+8)=n(x+16)$ байт.
- Открытая адресация (если запас в 1.5 раз): $1.5n{\cdot}x$ байт.

Время работы: открытася адресация делает 1 просмотр, списки 2 (к списку, затем к первому элементу) $\Rightarrow$ списки в два раза дольше.

14.2.4. C++

В плюсах зачем-то реализовали на списках... напишите свою, будет быстрее.

`unordered_set<int> h;` – хеш-таблица, хранящая множество `int`-ов.
Использование:

1. `unordered_set<int> h(N);`	выделить заранее память под N ячеек
2. `h.count(x);`	проверить наличие x
3. `h.insert(x);`	добавить x, если уже был, ничего не происходит
4. `h.erase(x);`	удалить x, если его не было, ничего не происходит

`unordered_map<int, int> h;` – хеш-таблица, хранящая `pair<int, int>`, пары `int`-ов.
Использование:

1. `unordered_map<int, int> h(N);`	выделить заранее память под N ячеек
2. `h[i] = x;`	i-й ячейкой можно пользовать, как обычным массивом
3. `h.count(i);`	есть ли пара с первой половиной i (ключ i)
4. `h.erase(i);`	удалить пару с первой половиной i (ключ i)

Относиться к `unordered_map` можно, как к обычному массиву с произвольными индексами.
В теории эта структура называется «ассоциативный массив»: каждому ключу i в соответствие ставится его значение $h[i]$.

Замечание 14.2.3. Чтобы работало всегда, придётся выделять память со случайным запасом:
`unordered_map<int, int> h(N + randomTime() % N)`,
чтобы ушлые люди не могли подобрать к вашей программ анти-хеш теста.

14.3. Универсальное семейство хеш функций

[wiki] [итмо-конспект] [Carter,Wegman'1977]

Def 14.3.1. *Хеш-функция. Сжимающее отображение* $h : U \to M$, $|U| > |M|$, $|M| = m$.

Def 14.3.2. *Универсальная система хеш-функций (1-я версия определения).*
Множество хеш-функций $\mathcal{H}$ *– универсальная система, если*

$$\forall x, y \; x \neq y \colon \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \tfrac{1}{m}$$

Def 14.3.3. *Универсальная система хеш-функций (2-я версия определения ≡ 1-й).*

$$\forall x, y \; x \neq y \colon \sum_{h \in \mathcal{H}}[h(x) = h(y)] \leq \tfrac{|\mathcal{H}|}{m}$$

14.4. Оценка для хеш-таблицы с закрытой адресацией

[wiki] [wiki-probability] [2-choice-hashing]

Пусть у нас есть универсальное семейство хеш-функций $\mathcal{H}$.
Хеш-таблица с закрытой адресацией (на списках) – вероятностный алгоритм.
Вся вероятностная часть заключается в том, что мы выбираем случайную $h \in \mathcal{H}$.

Операции `Find(x)`, `Del(x)` работают за длину списка.
Оценим для них матожидание времени работы.
$E[time(find(z))] = E[\text{длины списка}] = \sum\limits_{x \in table} Pr[h(x) = h(z)] = 1 + (n-1) \cdot \frac{1}{m} = \mathcal{O}(1 + \frac{n}{m})$, где
n – количество элементов в таблице, а m – размер таблицы (и диапазон хеш-функции).

Замечание 14.4.1. Мы оценили именно матожидание времени работы. Матожидание средней длины списка всегда $\frac{m}{n}$, даже если все элементы всегда класть в один список.

14.5. Оценки вероятностей коллизий

• Многочлены

Пусть m – простое число.

Lm 14.5.1. Число корней многочлена степени n над $\mathbb{Z}/m\mathbb{Z}$ не более n.

Lm 14.5.2. Пусть $t \in \mathbb{Z}/m\mathbb{Z} \Rightarrow Pr[P(t) = 0] = \frac{1}{m}$, где P – случайный многочлен.

Первую лемму вы знаете из курса алгебры,
вторая получается из того, что $P(x) = Q(x)(x-t)+r$: у случайного $P(x)$ все r равновероятны.

Нам важна простота модуля m, для непростого оценки неверны.
Например, у многочлена x^{64} при $m = 2^{64}$ любое чётное число является корнем.

• Связь со строками

$h_{p,m}(S) = S(p) \bmod m$, где S – и строка, и многочлен с коэффициентами S_i.
Тогда $h_{p,m}(S) = h_{p,m}(T) \Leftrightarrow (S - T)(p) \equiv 0 \bmod m$.
Запишем несколько следствий из лемм про многочлены.

Следствие 14.5.3. $\forall$ пары $\langle p, m \rangle$ вероятность совпадения хешей случайных строк s и t – $\frac{1}{m}$.

Доказательство. Разность многочленов s и t – случайный многочлен $(s - t)$. Далее 14.5.2. ■

Следствие 14.5.4. $\forall m, \forall s, t \; Pr_p[h_{p,m}(s) = h_{p,m}(t)] \leqslant \frac{\max(|s|,|t|)}{m}$.
По-русски: даны фиксированные строки, выбираем случайное p, оцениваем Pr коллизии.

Доказательство. Подставим многочлен $(s-t)$ в лемму 14.5.1. ■

Теперь пусть сравнений строк было много.

Теорема 14.5.5. Пусть дано множество **случайных** различных строк, и сделано k сравнений $\langle p, m\rangle$ хешей каких-то из этих строк $\Rightarrow$ вероятность существования коллизии не более $\frac{k}{m}$.

Доказательство. $Pr[\text{коллизии}] \leqslant E[\text{коллизий}] = k \cdot Pr[\text{коллизии при 1 сравнении}] = k/m$. ■

Теорема 14.5.6. Пусть дано множество **произвольных** различных строк длины $\leqslant n$, выбрано случайное p и сделано k сравнений $\langle p, m\rangle$ хешей каких-то из этих строк $\Rightarrow$ вероятность существования коллизии $\leqslant \frac{nk}{m}$.

Доказательство. Суммарное число корней у k многочленов степени $\leqslant n$ не более nk. ■

Замечание 14.5.7. На самом деле оценка $\frac{nk}{m}$ из 14.5.6 не достигается. В практических рассчётах можно смело пользоваться оценкой $\frac{k}{m}$ из 14.5.5.

14.5.1. Число различных подстрок (на практике)

Рассмотрим два решения, оценим их вероятности ошибок.
Решение #1: сложить хеши всех $\frac{1}{2}n(n-1)$ подстрок в хеш-таблицу.
Решение #2: отдельно решаем для каждой длины, внутри сложим хеши всех $\leqslant n$ подстрок в хеш-таблицу, просуммируем размеры хеш-таблиц.

В первом случае, у нас неявно происходит $\approx n^4/8$ сравнений подстрок $\Rightarrow$ вероятность наличия коллизии $\approx \frac{n^4/8}{m} \Rightarrow$ при $n = 1000$ нам точно не хватит 32-битного модуля,
при $n = 10\,000$ для $m \approx 10^{18}$ вероятность коллизии $\approx \frac{1}{800}$.

Во втором случае $\approx \sum_{i=1..n} i^2/2 \approx n^3/6$ сравнений подстрок $\Rightarrow$ вероятность наличия коллизии $\approx \frac{n^3/6}{m} \Rightarrow$ при $n = 10\,000$ для $m \approx 10^{18}$ вероятность коллизии $\approx \frac{1}{6\cdot 10^6}$.

Лекция #15: Ахо-Корасик

Декабрь 2021

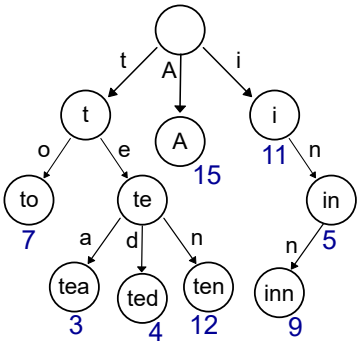
15.1. Бор

Бор – корневое дерево. Рёбра направлены от корня и подписаны буквами. Некоторые вершины бора подписаны, как конечные.

Базовое применение бора – хранение словаря `map<string, T>`.

Пример из wiki бора, содержащего словарь
$\{\text{A}:15,\ \text{to}:7,\ \text{tea}:3,\ \text{ted}:4,\ \text{ten}:12,\ \text{i}:11,\ \text{in}:5,\ \text{inn}:9\}$.

Для строки `s` операции `add(s)`, `delete(s)`, `getValue(s)` работают, как спуск вниз от корня.



Самый простой способ хранить бор: `vector<Vertex> t;`, где `struct Vertex { int id[|Σ|]; };`
Сейчас рёбра из вершины `t` хранятся в массиве `t.id[]`. Есть другие структуры данных:

Способ хранения	Время спуска по строке	Память на ребро
`array`	$\mathcal{O}(\|s\|)$	$\mathcal{O}(\|\Sigma\|)$
`list`	$\mathcal{O}(\|s\| \cdot \|\Sigma\|)$	$\mathcal{O}(1)$
`map (TreeMap)`	$\mathcal{O}(\|s\| \cdot \log \|\Sigma\|)$	$\mathcal{O}(1)$
`HashMap`	$\mathcal{O}(\|s\|)$ с большой const	$\mathcal{O}(1)$
`SplayMap`	$\mathcal{O}(\|s\| + \log S)$	$\mathcal{O}(1)$

Иногда для краткости мы будем хранить бор массивом `int next[N][|Σ|];`. `next[v][c] == 0` $\Leftrightarrow$ ребра нет.

• Сортировка строк

Если мы храним рёбра в структуре, способной перебирать рёбра в лексикографическом порядке (не хеш-таблица, не список), можно легко отсортировать массив строк:
(1) добавить их все в бор, (2) обойти бор слева направо.
Для `SplayMap` и n и строк суммарной длины S, получаем время $\mathcal{O}(S + n \log S)$.
Для `TreeMap` получаем $\mathcal{O}(S \log |\Sigma|)$.

Замечание 15.1.1. Если бы мы научились сортировать строки над произвольным алфавитом за $\mathcal{O}(|S|)$, то для $\Sigma = \mathbb{Z}$, получилась бы сортировка целых чисел за $\mathcal{O}(|S|)$.

Часто размер алфавита считают $\mathcal{O}(1)$.
Например строчные латинские буквы – 26, или любимый для биологов $|\{A, C, G, T\}| = 4$.

15.2. Алгоритм Ахо-Корасик

Даны текст t и словарь $s_1, s_2, \ldots, s_m$, нужно научиться искать словарные слова в тексте.
Простейший алгоритм, отлично работающий для коротких слов, – сложить словарные слова в бор и от каждой позиции текста i попытаться пройти вперёд, откладывая суффикс t_i вниз по бору, и отмечая все концы слов, которые мы проходим. Время работы – $\mathcal{O}(|t| \cdot \max |s_i|)$.

Ту же асимптотику можно получить, сложив все хеши всех словарных слов в хеш-таблицу, и

проверив, есть ли в хеш-таблице какие-нибудь подстроки t длины не более $\max |s_i|$.

Давайте теперь соптимизируем первое решение также, как префикс-функция, позволяет простейший алгоритм поиска подстроки в строке улучшить до линейного времени. Обобщение префикс-функции на бор – суффиксные ссылки:

Def 15.2.1. $\forall$ *вершины бора* v*:*
$str[v]$ *– строка, написанная на пути от корня бора до* v*.*
$suf[v]$ *– вершина бора, соответствующая самому длинному суффиксу* $str[v]$ *в боре.*

$\forall$ позиции текста i насчитаем вершину бора v_i: $str[v_i]$ – суффикс $t[0{:}i)$, $|str[v_i]| \to \max$.
Пересчёт v_i**:**

```
v[0] = root, p = root
for (i = 0; i < |t|; i++)
  while (next[p][t[i]] == 0) // ‘нет ребра‘
    p = suf[p]
  v[i + 1] = p = next[p][t[i]]
```

Чтобы цикл `while` всегда останавливался введём фиктивную вершину `f` и сделаем `suf[root] = f`, $\forall c$ `next[f][c] = root`.

Поиск словарных слов. Пометим все вершины бора, посещённые в процессе: `used[`v_i`] = 1`. В конце алгоритма поднимем пометки вверх по суффиксным ссылкам: `used[v]` $\Rightarrow$ `used[suf[v]]`. Для i-го словарного слова при добавлении мы запомнили вершину `end[i]`, тогда наличии этого слова в тексте лежит в `used[end[i]]`. Также можно насчитывать число вхождений.

Суффссылки. Чтобы всё это счастье работало осталось насчитать суффссылки.

Способ #1: полный автомат.

```
suf[v] = go[suf[parent[v]]][parent_char[v]];
go[v][c] = (next[v][c] ? next[v][c] : go[suf[v]][c]);
```

Естественно, чтобы от `parent[v]` и `suf[v]` всё было уже посчитано, поэтому нужно или перебирать вершины в порядке bfs от корня, или считать эту динамику рекурсивно-лениво.

Способ #2: пишем bfs от корня и пытаемся продолжить какой-нибудь суффикс отца.

```
q <-- root
while q --> v:
  z = suf[parent[v]]
  while next[z][parent_char[v]] == 0:
    z = suf[z]
  suf[v] = next[z][parent_char[v]]
```

Этот способ экономнее по памяти, если `next` – не массив, а, например, `map<int,int>`.

Теорема 15.2.2. Время построения линейно от длины суммарной строк, но не от размера бора.

Доказательство. Линейность от размера бора ломается на примере «бамбук длины n из букв a, из листа которого торчат рёбра по n разным символам». Линейность от суммарной длины строк следует из того, что если рассмотреть путь, соответствующий $\forall$ словарному слову s_i, то при вычислении суффссылок от вершин именно этого пути, указатель z в `while` всё время поднимался, а затем опускался не более чем на 1 $\Rightarrow$ сделал не более $2|s_i|$ шагов. ■

Лекция #16: Теория чисел

Декабрь 2021

16.1. Решето Эратосфена

Задача: найти все простые от 1 до n.
Решето Эратосфена предлагает вычёркивать числа, кратные уже найденным простым:

```
vector<bool> is_prime(n + 1, 1); // ‘Хорошо по памяти даже при n ≈ 10^9!‘
is_prime[0] = is_prime[1] = 0;
for (int i = 2; i <= n; i++)
  if (is_prime[i]) // ‘Нашли новое простое!‘
    for (int j = i + i; j <= n; j += i)
      is_prime[j] = 0; // ‘cnt++, чтобы определить константу‘
```

Замечание 16.1.1. Сейчас для каждого числа мы находим лишь один бит. Код легко модифицировать, чтобы для каждого числа находить наименьший простой делитель.

Данную версию кода можно соптимизировать в константу раз, пользуясь тем, что у любого не простого числа есть делитель не более корня.

```
for (int i = 2; i * i <= n; i++) // ‘cnt++, чтобы определить константу‘
  if (is_prime[i])
    for (int j = i * i; j <= n; j += i)
      is_prime[j] = 0; // ‘cnt++, чтобы определить константу‘
```

Можно ещё соптимизить: мы ищем только нечётные простые $\Rightarrow$
внешний цикл можно вести только по нечётным i, а во внутреннем прибавлять $2i$.
Теперь у нас три версии решета, отличающиеся не большими оптимизациями.
Эмпирический запуск при $n = 10^6$ даёт значения `cnt`: $3.7752\,n, 2.1230\,n, 0.8116\,n$ соответственно.

Теорема 16.1.2. Обе версии работают за $\Theta(n \log\log n)$.

- **(*) Более быстрое решение.**

Чтобы найти все простые от 1 до n за $\mathcal{O}(n)$, достаточно модифицировать алгоритм так, чтобы каждое составное x помечать лишь один раз, например, наименьшим простым делителем x.

Пусть $d[x]$ – номер наименьшего простого делителя x ($primes[d[x]]$ – собственно делитель).
Пусть $x = primes[d[x]] \cdot y \Rightarrow (d[y] \geqslant d[x] \vee y = 1)$
Алгоритм: перебирать y, а для него потенциальные $d[x]$ (простые не большие $d[y]$).

```
vector<int> primes, d(n + 1, -1);
for (int y = 2; y <= n; y++)
  if (d[y] == -1)
    d[y] = primes.size(), primes.push_back(\red{y});
  for (int i = 0; i <= d[y] && y * primes[i] <= n; i++)
    d[y * primes[i]] = i; // ‘x=y*primes[i], i=d[x]‘
```

16.2. Определение

Def 16.2.1. $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$ *поле остатков по модулю* p.

Def 16.2.2. $(\mathbb{Z}/m\mathbb{Z})^*$ *Группа по умножению* $\{a\colon (a,m)=1 \wedge 1 \le a < m\}$.

Def 16.2.3. *Линейное диофантово уравнение* $(a,b,c$ *даны, нужно найти* $x,y)$.

$$ax + by + c = 0;\ x, y \in \mathbb{Z}$$

Деление: $\frac{a}{b} = a \cdot b^{-1}$

• Алгоритм Евклида

Используется школьниками для подсчёта gcd:
$gcd(a,b) = gcd(a-b,b)$, повторяя вычитание много раз, получаем $gcd(a,b) = gcd(a \bmod b, b)$
Итого: `int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }`

16.3. Расширенный алгоритм Евклида

Теперь задача – найти ещё и $x,y\colon ax+by=gcd(a,b)$

Действуя опять же рекурсивно, имея базу, $a \cdot 1 + b \cdot 0 = a$
$a \cdot 0 + b \cdot 1 = b$
мы можем добавить переход из двух строк $a \cdot x_i + b \cdot y_i = r_i$
$a \cdot x_{i+1} + b \cdot y_{i+1} = r_{i+1}$
в новую $r_{i+2} = r_{i+1} \bmod r_i = r_{i+1} - kr_i$ (k – частное)
$x_{i+2} = x_{i+1} - kx_i$
$y_{i+2} = y_{i+1} - ky_i$

Заметим, что процесс на числах r_i – обычный алгоритм Евклида.

```
1 def euclid(a, b): # returns (x,y): ax + by = gcd(a, b)
2   if b == 0:
3     return 1, 0
4   x, y = euclid(b, a % b);
5   return y, x - (a // b) * y # ‘целочисленное деление‘
```

• Решение диофантового уравнения:

Если $c \bmod gcd(a,b) \neq 0$, то решений нет.
Иначе найдём $x,y\colon ax+by=gcd(a,b)$ и домножим уравнение на $c/gcd(a,b)$.

16.4. Обратные в $(\mathbb{Z}/m\mathbb{Z})^*$ и $\mathbb{Z}/p\mathbb{Z}$

Задача: a и m даны, хотим найти x: $a \cdot x \equiv 1 \bmod m$
Первый способ – решить диофантово уравнение $ax + my = 1 = gcd(a,m)$
Другой способ – воспользоваться малой теоремой Ферма или теоремой Эйлера:

$$a^{p-1} = 1 \bmod p \Rightarrow x = a^{p-2} \text{ (для простого)}$$
$$a^{\varphi(m)} = 1 \bmod m \Rightarrow x = a^{\varphi(m)-1} \text{ (для произвольного)}$$

Замечание 16.4.1. Функцию Эйлера считать долго!

$$\text{Пусть } n = \prod p_i^{\alpha_i} \Rightarrow \varphi(n) = n \prod \frac{p_i - 1}{p_i}, \text{ для вычисления нужна факторизация } n$$

16.5. Возведение в степень за $\mathcal{O}(\log n)$

Сводим к $\mathcal{O}(\log n)$ умножениям. Считаем, что одно умножение работает за $\mathcal{O}(1)$.

```
def pow(x, n):
  if n == 0: return 1
  return pow(x**2, n // 2)**2 * (x if n % 2 == 1 else 1)
```

16.6. Криптография. RSA.

Два типа шифрования:

Симметричная криптография. Один и тот же ключ позволяет и зашифровать, и расшифровать сообщение. Примеры шифрования: xor с ключом; циклический сдвиг алфавита.

Криптография с открытым ключем. Боб хочет послать сообщение Алисе и шифрует его *открытым ключом* Алисы (e), ключ (e) знают все. Для расшифровки Алисе понадобится ее закрытый ключ (d), который знает только она. Сами функции для шифровки и расшифровки открыты, их знают все.

• **RSA.**

Выберем два больших простых числа p, q. Посчитаем $n = pq, \varphi(n) = (p-1)(q-1)$.
Выберем случайное $1 \leqslant e < \varphi(n)$, посчитаем d: $ed \equiv 1 \bmod \varphi(n)$.
Итого: генерим случайно p, q, e; вычисляем n, $\varphi(n)$, d.
Тогда открытым ключем будет пара $\langle e, n\rangle$, а закрытым – $\langle d, n\rangle$.
Действия Боба для шифрования: $m \to \mu = m^e \bmod n$
Действия Алисы для дешифровки: $\mu \to m = \mu^d \bmod n$
Проверим корректность: $(m^e)^d = m^{ed} = m^{\varphi(n)\cdot k+1} \equiv 1^k \cdot m^1 = m$.

Алгоритм надежен настолько, насколько сложна задача факторизации чисел.
Числа умеют факторизовать так (более полный список на wiki):

(a) $\mathcal{O}(n^{1/2})$ – тривиальный перебор всех делителей до корня
(b) $\mathcal{O}(n^{1/4} \cdot gcd)$ – эвристика Полларда
(c) $L_n(1/2, 2\sqrt{2})$ – алгоритм Диксона-Крайчика
(d) $L_n(1/2, 2)$ – метод эллиптических кривых (алгоритм Ленстры)
(e) $L_n(1/3, (32/9)^3)$ – SNFS

Здесь $L_n(\alpha, c) = \mathcal{O}(e^{(c+o(1))(\log n)^\alpha})$.
При $0 < \alpha < 1$ получаем L_n между полиномом и экспонентой.
При $\alpha = 1$ получаем L_n – ровно полином $n^{c+o(1)}$.

Обычно в RSA используют ключ длины $k = 2048$.
При шифровке/расшифровке используют $\mathcal{O}(k)$ операций деления по модулю,
её мы скоро научимся реализовывать за $\mathcal{O}(k^2/w^2)$ и $\mathcal{O}(k \log^2 k)$, оптимальное время – $\mathcal{O}(k \log k)$.
Итого: простейшая реализация RSA даёт время $\mathcal{O}(k^3)$, оптимальная – $\mathcal{O}(k^2 \log k)$ и $\mathcal{O}(k^3/w^2)$.

Лекция #17: BST и AVL

Декабрь 2021

17.1. BST, базовые операции

Хеш-таблицы идеально умеют хранить неупорядоченные множества: `find`, `add`, `del` за $\mathcal{O}(1)$.
Для хранения упорядоченных по *ключу* x_i пар $\langle x_i, data_i \rangle$ будем использовать BST:

Def 17.1.1. *BST – binary search tree (бинарное дерево поиска). В каждой вершине пара* $\langle x, data \rangle$.
Левое поддерево содержит пары $\langle x, data \rangle$ *со строго меньшими* x.
Правое поддерево содержит пары $\langle x, data \rangle$ *со строго большими* x.

Как видно из определения, мы пока предполагаем, что все x_i различны.
$data_i$ – просто дополнительные данные. Например $\langle x_i, data_i \rangle$ – студент, x_i – имя студента.
Хранить вершины дерева будем, как `struct Node { int x; Data data; Node *p, *l, *r; };`

- `v->l` (`left`) – левый сын
- `v->r` (`right`) – правый сын
- `v->p` (`parent`) – отец

Отсутствие сына/отца будем обозначать нулевым указателем. Если мы не пользуемся отцом, то для экономии памяти и времени, хранить и поддерживать его не будем.

Find(x). Основная операция в дереве поиска – поиск. Чтобы проверить, присутствует ли ключ в дереве, спускаемся от корня вниз: если искомый меньше, идём налево, иначе направо.

Add(x). Операция добавления – делаем то же, что и `find`, в итоге или найдём x, или выйдем за пределы дерева. В то место, где мы вышли за пределы дерева, и вставим новое значение.

По BST можно за линию построить сортированный массив значений. Для этого нужно сделать так называемый «*симметричный обход дерева*» – рекурсивно обойти левое поддерево, выписать x, рекурсивно обойти правое поддерево. Через полученный массив можно определить:

- `next/prev(v)` – следующая/предыдущая в отсортированном порядке вершина дерева

Next(v). Если есть правый сын, спуститься в него и до упора влево, иначе идти вверх, пока не найдём больший ключ.

Del(x). Сперва сделаем `find`. Если у вершины не более одного ребёнка, её очень просто удалить — переподвесим своего ребёнка отцу. Если у вершины `v` два ребёнка, то перейдём в `next(v)` (идём вправо и до упора влево), сделаем `swap(v->x, next(v)->x)` и удалим `next(v)`. Заметим, что у `next(v)` точно нет левого ребёнка, так как, чтобы её найти, мы спустились до упора влево.

Пока преимуществом над хеш-таблицей является только `next`.
Вскоре появятся и другие операции, например, на практике сделаем `Node* lower_bound(x)`.
Все описанные операции: `add`, `del`, `find`, `next`, `lower_bound` работают за глубину дерева.
Чтобы в этом был толк, скоро изучим способы поддерживать глубину $\mathcal{O}(\log n)$.

- `(*)` **Ускорение некоторых операций**

По сути BST можно спарить с хеш-таблицей и двусвязным списком.

- `find` за $\mathcal{O}(1)$: хеш-таблица, для каждого x хранящая `Node* v`, содержащую x.
- `next/prev` за $\mathcal{O}(1)$: добавляем на вершинах двусвязный список, это называют *прошивкой*.

∘ `del` за $\mathcal{O}(1)$: он у нас выражался, как `find` + `next` + $\mathcal{O}(1)$.
∘ А ещё удалять можно лениво: `find` + $\mathcal{O}(1)$.

Итого все операции кроме `add` и `lower_bound` можно сделать за $\mathcal{O}(1)$.

• std::set<int>

`set<int>` – какое-то BST (на самом деле RB-tree).
`set<int>::iterator` – указатель на `Node` этого BST.

Заметим, что при модификации BST (add, del) меняются только указатели между вершинами ⇒ если у нас был итератор (`Node* it`), после модификаций он останется корректным итератором. С помощью прошивки мы теперь умеем делать быстрые `++` и `-` для рукописного итератора.

• Равные ключи

Как поступать при желании хранить равные ключи? Есть несколько способов.

1. Самый общий – сказать, что равных не бывает: замем x_i на пару $\langle x_i, i\rangle$.
2. Можно в вершине хранить пару $\langle x, count\rangle$ – сколько раз встречается ключ x. Если рядом с ключом есть дополнительная информация (например, мы храним в дереве студентов, а ключ – имя студента), то нужно хранить не число *count*, а список равных по ключу объектов (равных по имени студентов).
3. Можно ослабить условие из определения, хранить в правом поддереве «большие либо равные ключи». К сожалению, это сработает не для всех описанных ниже реализаций.
4. Можно ещё сильней ослабить определение – равные разрешать хранить и слева, и справа.

17.2. Немного кода

```
1  Node* next(Node* v) {
2    if (v->r != 0) { // от правого сына спускаемся до упора влево
3      v = v->r;
4      while (v->l)
5        v = v->l;
6    } else { // поднимаемся вверх, пока не окажемся левым сыном
7      while (v->p->r == v)
8        v = v->p;
9      v = v->p;
10   }
11   return v;
12 }
```

• Добавление

```
1  Node* add(Node* v, int x) { // v -- корень поддерева
2    if (!v)
3      return new Node(x); // единственная новая вершина
4    if (x < v->x)
5      v->l = add(v->l, x);
6    else
7      v->r = add(v->r, x);
8    return v;
9  }
```

Более короткая версия, использующая ссылку на указатель:

```
void add(Node* &v, int x) {
  if (!v)
    v = new Node(x);
  else
    add(x < v->x ? v->l : v->r, x);
}
```

Алгоритм 17.2.1. *Персистентная версия. Версия, которая не портит старые вершины.*

```
Node* add(Node* v, int x) {
  if (!v)
    return new Node(x);
  else if (x < v->x)
    return new Node(v->x, add(v->l, x), v->r);
  else
    return new Node(v->x, v->l, add(v->r, x));
}
```

Эта версия прекрасна тем, что уже созданные `Node`-ы никогда не изменяются (immutable). Поэтому можно писать так: `Node *root2 = add(root1, x)`, после чего у вас есть два корректных дерева – старое с корнем в `root1` и новое с корнем в `root2`. Да, эти деревья пересекаются по вершинам. Но любую операцию `add`/`find`/`lower_bound` мы можем проделать с любым из них.

• Поддержка характеристик поддеревьев

Очень часто мы хотим в каждом узле хранить какие-то характеристики его поддерева: размер, высоту, максимум в поддереве и т. д.. Естественно, мы должны поддерживать эти величины в актуальном состоянии.

Для этого обычно пишут функцию `update` (или `recalc`) в таком духе:

```
int height(Node *v){
  return v ? v->h : 0;
}
Node* update(Node* v){
  if (!v)
    return v;
  v->h = 1 + max(height(v->l), height(v->r));
  v->max = (v->r ? v->r->max : v->x);
  return v; // update(v) всегда возвращает v - просто для удобства использования
}
```

При изменении указателей в дереве нужно не забывать вызывать эту функцию. Например:

```
Node* add(Node* v, int x) {
  if (!v)
    return new Node(x);
  if (x < v->x)
    v->l = add(v->l, x);
  else
    v->r = add(v->r, x);
  return update(v);
}
```

Чтобы не заводить по функции (как `height`) на каждую новую характеристику, можно сделать фиктивную вершину, которая будет играть роль `nullptr` и иметь какие-то нейтральные значения (нулевая глубина и размер, максимум $-\infty$, и т. д.) — но проверки `if (!v)` нужно будет заменить на сравнения с этой вершиной.

17.3. AVL (Адельсон-Вельский, Ландис'1962)

Def 17.3.1. *Дерево будем называть сбалансированным iff глубина дерева* $\mathcal{O}(\log n)$.

Замечание 17.3.2. Тут нужно быть аккуратными. $\mathcal{O}(\log n)$ выше может быть рандомизированным или амортизированным. В таком случае мы всё равно будем называть дерево сбалансированным. В слове BST буква 'B' = binary, а не balanced.

Def 17.3.3. *Высота вершины* v *– максимальное из расстояний от* v *до листьев в поддереве* v.

Def 17.3.4. *BST называют AVL-деревом, если оно удовлетворяет AVL-инварианту: в каждой вершине разность высот детей не более одного:* $\forall v\ |h(v.l) - h(v.r)| \leqslant 1$.

Lm 17.3.5. Глубина AVL-дерева $\mathcal{O}(\log(n))$.

Доказательство. Обозначим S_h – минимальный размер дерева высоты h, тогда $n = S_h \geqslant S_{h-1} + S_{h-2} \Rightarrow S_h \geqslant h$-е число Фибоначчи $\geqslant 1.618^h \Rightarrow h \leqslant \log_{1.618} n$. ■

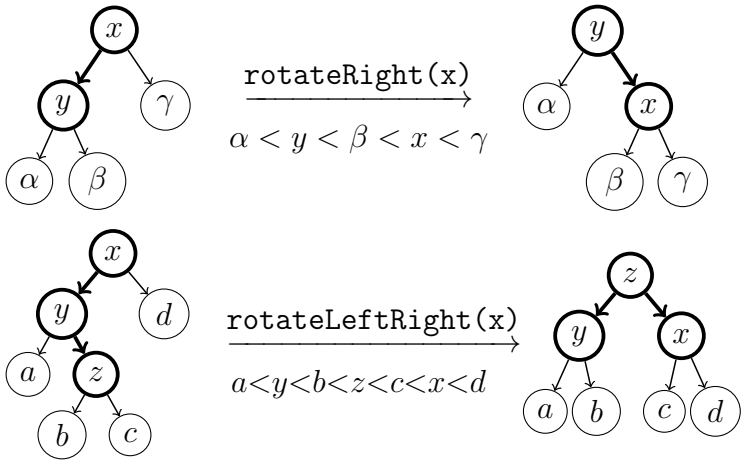
17.3.1. Добавление в AVL-дерево

• Краткое изложение

Добавление в AVL-дерево начинается также, как в обычное BST – спускаемся до упора, вставляем. Что могло пойти не так? Высоты некоторых вершин увеличились на 1, если после этого $|v.l.h - v.r.h| = 2$, нарушено AVL-свойство. Что делать? На обратном ходу рекурсии, поднимаясь снизу вверх, если видим некошерную вершину, «повернём» её поддерево (см. картинки).

(a) Добавление происходило во внука $v.l.l \Rightarrow h(v.l.l) = h(v.l.r) + 1$.
В этом случае достаточно сделать малое вращение по ребру $(v \to v.l)$.

(b) Добавление происходило во внука $v.l.r \Rightarrow h(v.l.l) + 1 = h(v.l.r)$.
В этом случае нужно делать большое вращение по рёбрам $(v \to v.l \to v.l.r)$.



```
1  Node* rebalance(Node* v) {
2    if (!v)
3      return v;
4
5    int hl = height(v->l), hr = height(v->r), d = hl - hr;
6    assert(abs(d) <= 2);
7    if (d == 2)
8      if (height(v->l->l) >= height(v->l->r)) // v->l существует, так как hl ⩾ 2
9        v = rotateRight(v); // Малое правое вращение
10     else
11       v = rotateLeftRight(v); // Большое правое вращение
12   else if (d == -2)
13     ...
14
15   return v;
16 }
17
18 Node* add(Node* v, int x) {
19   if (!v)
20     return new Node(x);
21   if (x < v->x)
22     v->l = add(v->l, x);
23   else
24     v->r = add(v->r, x);
25   return rebalance(v);
26 }
```

• **Более подробное изложение**

Описание перебалансировки в AVL-дереве можно прочитать по ссылке. Там же есть большой кусок реализации AVL-дерева. Доказательство корректности происходящего там не очень аккуратное, ниже доказано более аккуратно.

• **(*) Более аккуратное доказательство**

(Возможно, тут написано что-то переусложнённое; если где-то найдётся что-то более простое, можно будет заменить.)

TODO рисунок (пока рисунки можно смотреть всё по той же ссылке)

Rebalance.

Обозначим за h_1 высоты до вызова `rebalance`, h_2 — после.

Гарантии на вход: $v.l$ и $v.r$ являются корнями корректных AVL-деревьев, и $|h_1(v.l) - h_1(v.r)| \leqslant 2$.

Гарантии на выход. Пусть `u = rebalance(v)`. Тогда u является корнем корректного AVL-дерева (с теми же ключами/данными), и $h_1(v) - 1 \leqslant h_2(u) \leqslant h_1(v)$.

Доказательство гарантий на выход: не умаляя общности, рассматриваем только случаи, когда левое поддерево глубже (то есть $d = 2$) и мы делаем правое вращение — малое или большое.

Пусть $h := h_1(E)$. Тогда $h_1(b) = h + 2$ и $h_1(d) = h + 3$.

Нужно проверить корректность AVL-дерева. Кроме того, мы хотим доказать, что $h_1(v) - 1 \leqslant h_2(u) \leqslant h_1(v)$, то есть что $h + 2 \leqslant h_2(u) \leqslant h + 3$ (*).

◦ Первый случай. Пусть $h_1(A) \geqslant h_1(C)$. Тогда $h_1(A) = h_1(b) - 1 = h + 1$, и $h \leqslant h_1(C) \leqslant h + 1$. Мы выполняем правое малое вращение.

1. Проверяем AVL-условие в вершине d: $|h_2(C) - h_2(E)| = |(h \dots h + 1) - h| \leqslant 1$.

2. Высота d: $h_2(d) = 1 + \max(h_2(C), h_2(E)) = 1 + \max((h \dots h+1), h)$, то есть $h+1 \leqslant h_2(d) \leqslant h+2$.
3. Проверяем AVL-условие в вершине b: $|h_2(A) - h_2(d)| = |(h+1) - (h+1 \dots h+2)| \leqslant 1$.
4. Высота b: $h_2(b) = 1 + \max(h_2(A), h_2(d)) = 1 + \max(h+1, (h+1 \dots h+2))$, то есть $h+2 \leqslant h_2(b) \leqslant h+3$, что и требовалось в (*).

∘ Второй случай. Пусть $h_1(A) = h_1(c) - 1$. Тогда $h_1(c) = h_1(b) - 1 = h+1$, и $h_1(A) = h$. Кроме того, $h-1 \leqslant h_1(C') \leqslant h$ и $h-1 \leqslant h_1(C'') \leqslant h$.
Мы выполняем правое большое вращение.

1. Проверяем AVL-условие в вершине b: $|h_2(A) - h_2(C')| = |h - (h-1 \dots h)| \leqslant 1$.
2. Высота b: $h_2(b) = 1 + \max(h_2(A), h_2(C')) = 1 + \max(h, (h-1 \dots h))$, то есть $h_2(b) = h+1$.
3. Аналогично проверяем AVL-условие в вершине d.
4. Аналогично вычисляем $h_2(d) = h+1$.
5. Проверяем AVL-условие в вершине c: $|h_2(b) - h_2(d)| = |(h+1) - (h+1)| = 0$.
6. Высота c: $h_2(b) = 1 + \max(h_2(b), h_2(d)) = 1 + \max(h+1, h+1) = h+2$, то есть (*) выполнено.

Add.
Обозначим за h_0 высоты до вызова `add`, h_1 — до вызова `rebalance`, h_2 — после.
Гарантии на вход: v является корнем корректного AVL-дерева, и $|h_1(v.l) - h_1(v.r)| \leqslant 2$.
Гарантии на выход. Пусть `u = add(v, x)`. Тогда u является корнем корректного AVL-дерева (с теми же ключами/данными плюс новый ключ), и $h_0(v) \leqslant h_2(u) \leqslant h_0(v) + 1$.

Доказательство гарантий на выход. Не умаляя общности, пусть рекурсивный запуск был вызван от левого поддерева. После рекурсивного запуска левое и правое поддерево v — корректные AVL-деревья: правое не менялось, левое — по предположению индукции. Высота правого не поменялась, а левого увеличилась не больше, чем на единицу, поэтому $h_0(v) \leqslant h_1(v) \leqslant h_0(v) + 1$.
Теперь хотим запустить `rebalance(v)`. Нужно проверить гарантию на вход. Было: $|h_0(v.l) - h_0(v.r)| \leqslant 1$. Рекурсивный запуск увеличил одну из глубин не больше, чем на 1 (по предположению индукции), поэтому теперь разница высот не больше двух и мы имеем право запустить `u = rebalance(v)`.
Если в вершине v уже выполнено AVL-условие, то `rebalance(v)` ничего не сделает, и всё доказано. Если же оно перестало выполняться, это значит, что высота $v.l$ стала на два больше, чем высота $v.r$. Отметим, что в этом случае $h_1(v) = h_0(v) + 1$.
По гарантии `rebalance` на выход, u — корень корректного AVL-дерева, и, кроме того, $h_1(v) - 1 \leqslant h_2(u) \leqslant h_1(v)$. Поэтому $(h_0(v) + 1) - 1 \leqslant h_2(u) \leqslant h_0(v) + 1$, что и требовалось доказать.

• (*) При добавлении вращений мало

Можно показать, если в `add(v)` выполнялось вращение, то высота вершины v до добавления ($h_0(v)$), и после *добавления и вращения* ($h_2(u)$) равны ⇒
сразу же после первого вращения операцию перебалансировки можно прервать.
Можно сформулировать это следующим образом:

Lm 17.3.6. При добавлении в AVL-дереве происходит $\mathcal{O}(1)$ присваиваний указателей.

К сожалению, высоты могут поменяться у $\mathcal{O}(\log n)$ вершин.
Тем не менее, можно показать, что амортизированное число изменений высот $\mathcal{O}(1)$.

- **Удаление**

Удаление из AVL-дерева происходит так же, как удаление из обычного BST.
На обратном ходе рекурсии от удалённой вершины происходит перебалансировка.
В отличие от добавления, повороты могут происходить много раз.

- **(*) Персистентная версия**

Замечание 17.3.7. И в добавлении, и в удалении при подъёме вверх и перебалансировке можно пользоваться ссылками на родителя. Но удобнее всю перебалансировку делать именно на обратном ходу рекурсии.

Замечание 17.3.8. Раз отцы не нужны, дерево можно сделать персистентным (см. **??**).
Получается очень простой и короткий код вращения.
Большое вращение выражается через два малых.
Недостаток: тратим $\mathcal{O}(\log n)$ памяти на каждую операцию изменения дерева.

```
1 // Node = x, l, r
2 Node* rotateRight(Node* v) {
3   return new Node {v->l->x, v->l->l, new Node {v->x, v->l->r, v->r}};
4 }
5 Node* rotateLeftRight(Node* v) {
6   return rotateRight(new Node {v->x, rotateLeft(v->l), v->r});
7 }
```