SPb HSE, 1 курс ПМИ, осень 2024/25
Конспект лекций по алгоритмам

Собрано 10 июня 2025 г. в 13:42

Содержание

Лекция #1: Асимптотика

1-я пара, 2024/25

1.1. О курсе. Хорошие алгоритмы.

Что такое алгоритм, вы представляете. А что такое хороший алгоритм?

1. Алгоритм, который работает на всех тестах. Очень важное свойство. Нам не интересны решения, для которых есть тесты, на которых они не работают.

2. Алгоритм, который работает быстро. Что такое быстро? Время работы программы зависит от размера входных данных. Размер данных часто обозначают за n. Алгоритм, находящий минимум в массиве длины n делает $\approx 4n$ операций. Нам прежде всего важна зависимость от n (пропорционально n), и только во-вторых константа (≈ 4). На самом деле разные операции выполняются разное время, об этом в следующей главе.

Насколько быстро должны работать наши программы? Обычные процессоры для ноутов и телефонов имеют несколько ядер, каждое частотой $\sim$2GHz. Параллельные алгоритмы мы изучать не будем, всё, что изучим, заточено под работу на одном ядре. $\sim$2GHz это 2 000 000 000 элементарных операций в секунду. Если мы пишем на языке `C++` (а мы будем), то это $\approx 10^9$ команд в секунду. Если, например, на `python`, то реально мы успеем выполнить 10^6, в $\approx$1000 раз меньше команд в секунду. За эталон мы берём именно одну секунду — минута по человеческим ощущениям очень медленно, а сотую секунды человек не почувствует.

3. Алгоритм, который использует мало оперативной памяти. Вообще память более дорогой ресурс, чем время, об этом будет в следующей главе, в части про кеш.

4. Простые и понятные алгоритмы. Если алгоритм сложно понять, пересказать (выше порог вхождения), если он содержит много крайних случаев $\Rightarrow$ его сложно корректно реализовать, в нём вероятны ошибки, которые однажды выстрелят.

• **Асимптотика.**

Ближайшие две главы мы будем говорить преимущественно про скорость работы.
Рассмотрим простейший алгоритм, который перебирает все пары i, j: $i \leqslant j \leqslant n$.

```
1 int ans = 0;
2 for (int i = 1; i <= n; i++) // нам дали n
3     for (int j = 1; j <= i; j++)
4         ans++;
5 cout << ans << endl;
```

Мы можем посчитать точное число всех операций (сравнение, присваивание, сложение, ...) в зависимости от n: $1 + 3(1{+}2{+}3{+}\ldots{+}n) + 1$ и получить $f(n) = \frac{3}{2}n(n{+}1) + 2$.

$f(n)$ — время работы программы в зависимости от n, а n — параметр задачи, зачастую «размер входных данных». Ниже мы будем предполагать, что $n \in \mathbb{N}$, $f(n) > 0$. Интересно, насколько быстро растёт время программы в зависимости от n (размера данных). Наша $f(n) \sim n^2$, мы будем говорить «асимптотически работает за n^2» или «за n^2 с точностью до константы», это и есть *асимптотическая часть времени работы*, *асимптотика времени работы*.

Выше мы считали, что все операции работают одно и то же время, просто считали их количество. А потом ещё и забили на константу $\frac{3}{2}$ при n^2. Дальше мы разберёмся с константами и с тем,

какие операции медленнее, какие быстрее. А сейчас сосредоточимся **только** на асимптотике.

1.2. Асимптотика, $\mathcal{O}$-обозначния

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

Def 1.2.1. $f = \Theta(g) \quad \exists N > 0,\, C_1 > 0, C_2 > 0\colon \forall n \geqslant N, C_1 \cdot g(n) \leqslant f(n) \leqslant C_2 \cdot g(n)$

Def 1.2.2. $f = \mathcal{O}(g) \quad \exists N > 0,\, C > 0\colon \forall n \geqslant N, f(n) \leqslant C \cdot g(n)$

Def 1.2.3. $f = \Omega(g) \quad \exists N > 0,\, C > 0\colon \forall n \geqslant N, f(n) \geqslant C \cdot g(n)$

Def 1.2.4. $f = o(g) \quad \forall C > 0\ \exists N > 0\colon \forall n \geqslant N, f(n) \leqslant C \cdot g(n)$

Def 1.2.5. $f = \omega(g) \quad \forall C > 0\ \exists N > 0\colon \forall n \geqslant N, f(n) \geqslant C \cdot g(n)$

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 1.2.6. $f = \Theta(g) \Leftrightarrow g = \Theta(f)$

Замечание 1.2.7. $f = \mathcal{O}(g), g = \mathcal{O}(f) \Leftrightarrow f = \Theta(g)$

Замечание 1.2.8. $f = \Omega(g) \Leftrightarrow g = \mathcal{O}(f)$

Замечание 1.2.9. $f = \omega(g) \Leftrightarrow g = o(f)$

Замечание 1.2.10. $f = \mathcal{O}(g), g = \mathcal{O}(h) \Rightarrow f = \mathcal{O}(h)$

Замечание 1.2.11. Обобщение: $\forall \beta \in \{\mathcal{O}, o, \Theta, \Omega, \omega\}\colon f = \beta(g), g = \beta(h) \Rightarrow \boxed{f = \beta(h)}$

Замечание 1.2.12. $\forall C > 0 \quad C \cdot f = \Theta(f)$

Докажем для примера Rem 1.2.6.

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

Упражнение 1.2.13. $f = \mathcal{O}(\Theta(\mathcal{O}(g))) \Rightarrow f = \mathcal{O}(g)$

Упражнение 1.2.14. $f = \Theta(o(\Theta(\mathcal{O}(g)))) \Rightarrow f = o(g)$

Упражнение 1.2.15. $f = \Omega(\omega(\Theta(g))) \Rightarrow f = \omega(g)$

Упражнение 1.2.16. $f = \Omega(\Theta(\mathcal{O}(g))) \Rightarrow f$ может быть любой функцией

Lm 1.2.17. $g = o(f) \Rightarrow f \pm g = \Theta(f)$

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

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

Lm 1.2.19. $P(x)$ – 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 Rem 1.2.12, Lm 1.2.18 имеем, что все слагаемые кроме $a_k x^k$ являются $o(x^{\deg P})$. Поэтому по лемме Lm 1.2.17 вся сумма является $\Theta(x^k)$. ■

1.3. Рекуррентности и Карацуба

• Алгоритм умножения чисел в столбик

Рассмотрим два многочлена $A(x) = 5 + 4x + 3x^2 + 2x^3 + x^4$ и $B(x) = 9 + 8x + 7x^2 + 6x^3$.
Запишем массивы `a[] = {5, 4, 3, 2, 1}`, `b[] = {9, 8, 7, 6}`.

```
for (i = 0; i < an; i++) // an = 5
    for (j = 0; j < bn; j++) // bn = 4
        c[i + j] += a[i] * b[j];
```

Мы получили в точности коэффициенты многочлена $C(x) = A(x)B(x)$.

Теперь рассмотрим два числа $A = 12345$ и $B = 6789$, запишем те же массивы и сделаем:

```
// Перемножаем числа без переносов, как многочлены
for (i = 0; i < an; i++) // an = 5
    for (j = 0; j < bn; j++) // bn = 4
        c[i + j] += a[i] * b[j];
// Делаем переносы, массив c = [45, 76, 94, 100, 70, 40, 19, 6, 0]
for (i = 0; i < an + bn; i++)
    if (c[i] >= 10)
        c[i + 1] += c[i] / 10, c[i] %= 10;
// Массив c = [5, 0, 2, 0, 1, 8, 3, 8, 0], ответ = 83810205
```

Данное умножение работает за $\Theta(nm)$, или $\Theta(n^2)$ в случае $n = m$.

Следствие 1.3.1. Чтобы умножать длинные числа достаточно уметь умножать многочлены.

Многочлены мы храним, как массив коэффициентов. При программировании умножения, нам важно знать не степень многочлена d, а длину этого массива $n = d + 1$.

• Алгоритм Карацубы

Чтобы перемножить два многочлена (или два длинных целых числа) $A(x)$ и $B(x)$ из n коэффициентов каждый, разделим их на части по $k = \frac{n}{2}$ коэффициентов – A_1, A_2, B_1, B_2.
Заметим, что $A \cdot B = (A_1 + x^k A_2)(B_1 + x^k B_2) = A_1B_1 + x^k(A_1B_2 + A_2B_1) + x^{2k}A_2B_2$.
Если написать рекурсивную функцию умножения, то получим время работы:

$$T_1(n) = 4T_1(\tfrac{n}{2}) + \Theta(n)$$

Из последующей теоремы мы сделаем вывод, что $T_1(n) = \Theta(n^2)$. Алгоритм можно улучшить, заметив, что $A_1B_2 + A_2B_1 = (A_1 + A_2)(B_1 + B_2) - A_1B_1 - A_2B_2$, где вычитаемые величины уже посчитаны. Итого три умножения вместо четырёх:

$$T_2(n) = 3T_2(\tfrac{n}{2}) + \Theta(n)$$

Из последующей теоремы мы сделаем вывод, что $T_2(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585\ldots})$.
Данный алгоритм применим и для умножения многочленов, и для умножения чисел.

Псевдокод алгоритма Карацубы для умножения многочленов:

```
Mul(n, a, b): // n = 2^k, c(w) = a(w)*b(w)
    if n == 1: return {a[0] * b[0]}
    a --> a1, a2
    b --> b1, b2
    x = Mul(n / 2, a1, b1)
    y = Mul(n / 2, a2, b2)
    z = Mul(n / 2, a1 + a2, b1 + b2)
    // Умножение на w^i - сдвиг массива на i вправо
    return x + y * w^n + (z - x - y) * w^(n/2);
```

Чтобы умножить числа, сперва умножим их как многочлены, затем сделаем переносы.

1.4. Мастер Теорема

Теорема 1.4.1. *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 1.4.2. *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. ■

1.5. (*) Экспоненциальные рекуррентные соотношения

Теорема 1.5.1. *Об экспоненциальном рекуррентном соотношении*
Пусть $T(n) = \sum b_i T(n - a_i)$. При этом $a_i > 0, b_i > 0, \sum b_i > 1$.
Тогда $T(n) = \Theta(\alpha^n)$, при этом $\alpha > 1$ и является корнем уравнения $1 = \sum b_i \alpha^{-a_i}$, его можно найти бинарным поиском.

Доказательство. Предположим, что $T(n) = \alpha^n$, тогда $\alpha^n = \sum b_i \alpha^{n-a_i} \Leftrightarrow 1 = \sum b_i \alpha^{-a_i} = f(\alpha)$.
Теперь нам нужно решить уравнение $f(\alpha) = 1$ для $\alpha \in [1, +\infty)$.
Если $\alpha = 1$, то $f(\alpha) = \sum b_i > 1$, если $\alpha = +\infty$, то $f(\alpha) = 0 < 1$. Кроме того $f(\alpha) \searrow [1, +\infty)$.
Получаем, что на $[1, +\infty)$ есть единственный корень уравнения $1 = f(\alpha)$ и его можно найти бинарным поиском.

Мы показали, откуда возникает уравнение $1 = \sum b_i \alpha^{-a_i}$. Доказали, что у него $\exists!$ корень α. Теперь докажем по индукции, что $T(n) = \mathcal{O}(\alpha^n)$ (оценку сверху) и $T(n) = \Omega(\alpha^n)$ (оценку снизу). Доказательства идентичны, покажем $T(n) = \mathcal{O}(\alpha^n)$. База индукции:

$$\exists C \colon \forall n \in B = [1 - \max_i a_i, 1] \;\; T(n) \leqslant C\alpha^n$$

Переход индукции:

$$T(n) = \sum b_i T(n - a_i) \overset{\text{по индукции}}{\leqslant} C \sum b_i \alpha^{n-a_i} \overset{(*)}{=} C\alpha^n$$

(*) Верно, так как α – корень уравнения. ■

1.6. (*) Доказательства по индукции

Lm 1.6.1. *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: $T(n) \leqslant \max\limits_{x=1..n-1}\big(x^2 + (n-x)^2 + x(n-x)\big) \leqslant \max\limits_{x=1..n-1}\big(x^2 + (n-x)^2 + 2x(n-x)\big) = n^2$

• Примеры по теме рекуррентные соотношения

1. $T(n) = T(n-1) + T(n-1) + T(n-2)$.
   Угадаем ответ 2^n, проверим по индукции: $2^n = 2^{n-1} + 2^{n-1} + 2^{n-2}$.
2. $T(n) = T(n-3) + T(n-3) \Rightarrow T(n) = 2T(n-3) = 4T(n-6) = \cdots = 2^{n/3}$
3. $T(n) = T(n-1) + T(n-3)$. Применяем Thm 1.5.1, получаем $1 = \alpha^{-1} + \alpha^{-3}$, находим α бинпоиском, получаем $\alpha = 1.4655\ldots$.

1.7. Числа Фибоначчи

Def 1.7.1. $f_1 = f_0 = 1, f_i = f_{i-1} + f_{i-2}$. f_n – *n-е число Фибоначчи.*

• Оценки снизу и сверху

$f_n = f_{n-1} + f_{n-2}$, рассмотрим $g_n = g_{n-1} + g_{n-1}$, $2^n = g_n \geqslant f_n$.
$f_n = f_{n-1} + f_{n-2}$, рассмотрим $g_n = g_{n-2} + g_{n-2}$, $2^{n/2} = g_n \leqslant f_n$.
Воспользуемся Thm 1.5.1, получим $1 = \alpha^{-1} + \alpha^{-2} \Leftrightarrow \alpha^2 - \alpha - 1 = 0$, получаем $\alpha = \frac{\sqrt{5}+1}{2} \approx 1.618$.
$f_n = \Theta(\alpha^n)$.

1.8. (*) $\mathcal{O}$-обозначения через пределы

Def 1.8.1. $f = o(g)$ *Определение через предел:* $\lim\limits_{n\to+\infty} \frac{f(n)}{g(n)} = 0$

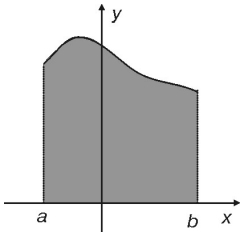
Def 1.8.2. $f = \mathcal{O}(g)$ *Определение через предел:* $\overline{\lim\limits_{n\to+\infty}} \frac{f(n)}{g(n)} < \infty$

Здесь необходимо пояснение: $\overline{\lim\limits_{n\to+\infty}} f(n) = \lim\limits_{n\to+\infty}(\sup\limits_{x\in[n..+\infty]} f(x))$, где sup – верхняя грань.

Lm 1.8.3. Определения o эквивалентны

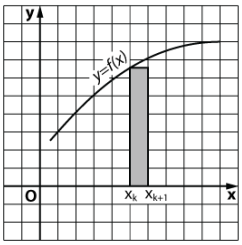
Доказательство. Вспомним, что речь о положительных функциях f и g.
Распишем предел по определению: $\forall C > 0 \quad \exists N \quad \forall n \geqslant N \quad \frac{f(n)}{g(n)} \leqslant C \Leftrightarrow f(n) \leqslant Cg(n)$. ■

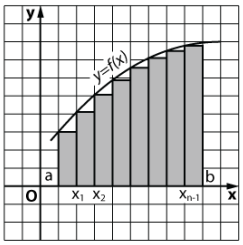
1.9. (*) Замена сумм на интегралы



Def 1.9.1. *Определённый интеграл* $\int_a^b f(x)dx$ *положительной функции* $f(x)$ *– площадь под графиком* f *на отрезке* $[a..b]$.

Lm 1.9.2. $\forall f(x) \nearrow [a..a{+}1] \Rightarrow f(a) \leqslant \int_a^{a+1} f(x)dx \leqslant f(a{+}1)$





Lm 1.9.3. $\forall f(x) \nearrow [a..b{+}1] \Rightarrow \sum_{i=a}^{b} f(i) \leqslant \int_a^{b+1} f(x)dx$

Доказательство. Сложили неравенства из Lm 1.9.2 ■

Lm 1.9.4. $\forall f(x) \nearrow [a..b], f > 0 \Rightarrow \int_a^b f(x)dx \leqslant \sum_{i=a}^{b} f(i)$

Доказательство. Сложили неравенства из Lm 1.9.2, выкинули $[a{-}1, a]$ из интеграла. ■

Теорема 1.9.5. *Замена суммы на интеграл #1*

$\forall f(x) \nearrow [1..\infty), f > 0, S(n) = \sum_{i=1}^{n} f(i), I_1(n) = \int_1^n, I_2(n) = \int_1^{n+1}, I_1(n) = \Theta(I_2(n)) \Rightarrow \boxed{S(n) = \Theta(I_1(n))}$

Доказательство. Из лемм Lm 1.9.3 и Lm 1.9.4 имеем $I_1(n) \leqslant S(n) \leqslant I_2(n)$.
$C_1 I_1(n) \leqslant I_2(n) \leqslant C_2 I_1(n) \Rightarrow I_1(n) \leqslant S(n) \leqslant I_2(n) \leqslant C_2 I_1(n)$ ■

Теорема 1.9.6. *Замена суммы на интеграл #2*

$\forall f(x) \nearrow [a..b], f > 0 \quad \int_a^b f(x)dx \leqslant \sum_{i=a}^{b} f(i) \leqslant f(b) + \int_a^b f(x)dx$

Доказательство. Первое неравенство – лемма Lm 1.9.3. Второе – Lm 1.9.4, применённая к $\sum_{i=a}^{b-1}$ ■

Следствие 1.9.7. Для убывающих функций два последних факта тоже верны.
Во втором ошибкой будет не $f(b)$, а $f(a)$, которое теперь больше.

• Как считать интегралы?

Формула Ньютона-Лейбница: $\int_a^b f'(x)dx = f(b) - f(a)$

Пример: $\ln'(n) = \frac{1}{n} \Rightarrow \int_1^n \frac{1}{x}dx = \ln n - \ln 1 = \ln n$

1.10. Примеры по теме асимптотики

• Вложенные циклы for

```
#define forn(i, n) for (int i = 0; i < n; i++)
int counter = 0, n = 100;
forn(i, n)
  forn(j, i)
    forn(k, j)
      forn(l, k)
        forn(m, l)
          counter++;
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.

• За сколько вычисляется n-е число Фибоначчи?

```
f[0] = f[1] = 1;
for (int i = 2; i < n; i++)
    f[i] = f[i - 1] + f[i - 2];
```

Казалось бы за $\mathcal{O}(n)$. Но это в предположении, что «+» выполняется за $\mathcal{O}(1)$. На самом деле мы знаем, что $\log f_n = \Theta(n)$, т.е. складывать нужно числа длины $n \Rightarrow$ «+» выполняется за $\Theta(i)$, а n-е число Фибоначчи считается за $\Theta(n^2)$.

• Задача из теста про $a^2 + b^2 = N$

```
int b = sqrt(N);
for (int a = 1; a * a <= N; a++)
    while (a * a + b * b >= N; b--)
        ;
    if (a * a + b * b == N)
        cnt++;
```

Время работы $\Theta(N^{1/2})$, так как в сумме b уменьшится лишь $N^{1/2}$ раз. Здесь мы первый раз использовали так называемый «метод двух указателей».

• Число делителей числа

```
vector<int> divisors[n + 1]; // все делители числа
for (int a = 1; a <= n; a++)
    for (int b = a; b <= n; b += a)
        divisors[b].push_back(a);
```

За сколько работает программа?

$$\sum_{a=1}^{n} \lceil \tfrac{n}{a} \rceil = \mathcal{O}(n) + \sum_{a=1}^{n} \tfrac{n}{a} = \mathcal{O}(n) + n \sum_{a=1}^{n} \tfrac{1}{a} \overset{Thm\ 1.9.5}{=} \mathcal{O}(n) + n \cdot \Theta(\int_{1}^{n} \tfrac{1}{x} dx) = \Theta(n \log n)$$

• **Сумма гармонического ряда**

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

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

1.11. Сравнение асимптотик

Def 1.11.1. *Линейная сложность* $\mathcal{O}(n)$

Def 1.11.2. *Квадратичная сложность* $\mathcal{O}(n^2)$

Def 1.11.3. *Полиномиальная сложность* $\exists\, k > 0\colon \mathcal{O}(n^k)$

Def 1.11.4. *Полилогарифм* $\exists\, k > 0\colon \mathcal{O}(\log^k n)$

Def 1.11.5. *Экспоненциальная сложность* $\exists\, c > 0\colon \mathcal{O}(2^{cn})$

Теорема 1.11.6. $\forall x, y > 0, z > 1\ \exists N\ \forall n > N\colon\ \log^x n < n^y < z^n$

Доказательство. Сперва докажем первую часть неравенства через вторую.
Пусть $\log n = k$, тогда $\log^x n < n^y \Leftrightarrow k^x < 2^{ky} = (2^y)^k = z^k \Leftarrow n^y < z^n$ ■
Докажем вторую часть исходного неравенства $n^y < z^n \Leftrightarrow n < 2^{\frac{1}{y} n \log z}$
Пусть $n' = \frac{1}{y} n \log z$, обозначим $C = 1/(\frac{1}{y}\log z)$, пусть $C \leqslant n'$ (возьмём достаточно большое n),
тогда $n^y < z^n \Leftrightarrow n < 2^{\frac{1}{y} n \log z} \Leftrightarrow C \cdot n' < 2^{n'} \Leftarrow (n')^2 < 2^{n'}$

Осталось доказать $n^2 < 2^n$. Докажем по индукции.
База: для любого значения из интервала [10..20) верно,
так как $n^2 \in [100..400) < 2^n \in [1024..1048576)$.
Если n увеличить в два раза, то $n^2 \to 4 \cdot n^2$, а $2^n \to 2^{2n} = 2^n \cdot 2^n \geqslant 4 \cdot 2^n$ при $n \geqslant 2$.
Значит $\forall n \geqslant 2$ если для n верно, то и для $2n$ верно.
Переход: $[10..20) \to [20..40) \to [40..80) \to \ldots$ ■

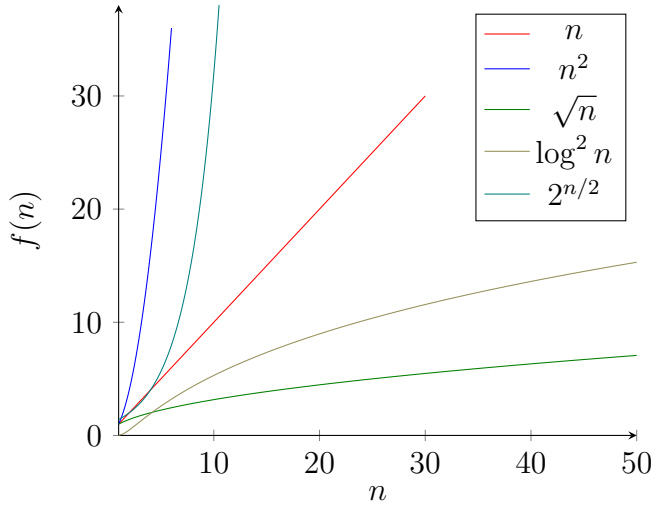
Следствие 1.11.7. $\forall x, y > 0, z > 1\colon\ \log^x n = \mathcal{O}(n^y), n^y = \mathcal{O}(z^n)$

Доказательство. Возьмём константу 1. ■

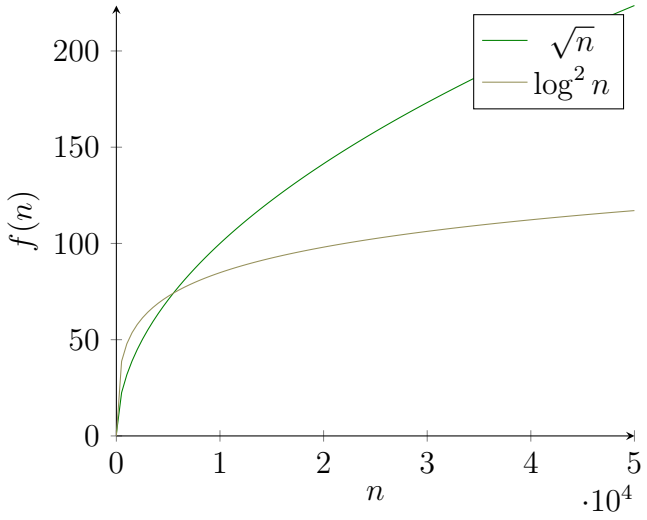
Следствие 1.11.8. $\forall x, y > 0, z > 1\colon\ \log^x n = o(n^y), n^y = o(z^n)$

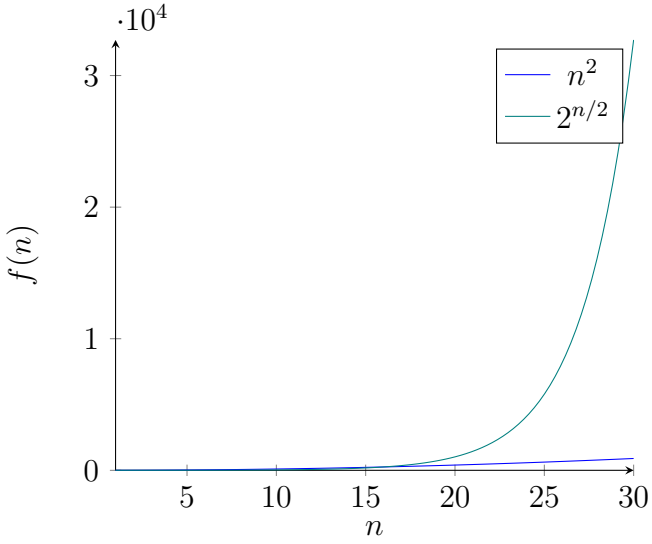
Доказательство. Достаточно перейти к чуть меньшим y, z и воспользоваться теоремой.
$\exists N\, \forall n \geqslant N\ \log^x n < n^{y-\varepsilon} = \frac{1}{n^\varepsilon} n^y,\ \frac{1}{n^\varepsilon} \underset{n\to\infty}{\to} 0 \Rightarrow \log^x = o(n^y)$.
$\exists N\, \forall n \geqslant N\ n^y < (z-\varepsilon)^n = \frac{1}{(z/(z-\varepsilon))^n} z^n,\ \frac{1}{(z/(z-\varepsilon))^n} \underset{n\to\infty}{\to} 0 \Rightarrow n^y = o(z^n)$. ■

• **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

2-я пара, 2024/25

2.1. 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.

```
1 #define _GLIBCXX_DEBUG // должно быть до всех #include
2 // #define _LIBCPP_DEBUG 1 (аналог для компилятора clang)
3 #include <vector> // должно быть после
4 vector<int> a(3);
5 a[3] = 7; // Runtime Error!
```

Для пользователей linux есть более профессиональное решение: valgrind.

UB (undefined behavior) – моменты, когда заранее неизвестно, как поведёт себя программа (из-за ошибок в коде). Его очень сложно найти (т.к. оно может то проявляться, то нет, у вас локально работает, на сервере нет и т.д.). Типы:

1. забыл вернуть ответ из функции (ловится через `-W...`)
2. забыл инициализировать переменную (ловится через `-W...`)
3. вышел за пределы вектора (ловится `#define ...`)
4. криво используем итераторы сета/вектора (ловится `#define ...`)

Типов ещё много, эти самые распространённые. Вам **не** нужно искать эти ошибки, вам нужно прописать один раз в жизни в настройки компилятора `-W...` и в шаблон кода `#define`, и всё будет искаться само.

Пожалуйста, берегите своё время и нервы, не ходите по уже хорошо изученным граблям.

- **Struct (структуры)**

```
struct Point {
  int x, y;
};
Point p, q = {2, 3}, *t = new Point {2, 3};
p.x = 3;
```

- **Pointers (указатели)**

Рассмотрим указатель `int *a;`

`a` – указатель на адрес в памяти (по сути целое число, номер ячейки).

`*a` – значение, которое лежит по адресу.

```
int b = 3;
int *a = &b; // сохранили адрес b в пременную a типа int*
int c[10];
a = c; // указатель на первый элемент массива
*a = 7; // теперь c[0] == 7
Point *p = new Point {0, 0}; // выделили память под новый Point, указтель записали в p
(*p).x = 3; // записали значение в x
p->x = 3; // запись, эквивалентная предыдущей
```

2.2. 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. Ввод и вывод данных. `cin/cout`, `scanf/printf`...
   Используйте буфферизированный ввод/вывод через `fread/fwrite`.
2. Операции библиотеки `<math.h>`: `sqrt, cos, sin, atan` и т.д.
   Эти операции раскладывают переданный аргумент в ряд, что происходит не за $\mathcal{O}(1)$.
3. Взятие числа по модулю, деление с остатком: `a / b, a % b`.
4. Доступ к памяти. Существует два способа прохода по массиву:
   Random access: `for (i = 0; i < n; i++) sum += a[p[i]];` где p – случайная перестановка
   Sequential access: `for (i = 0; i < n; i++) sum += a[i];`
5. Функции работы с памятью: `new`, `delete`. Тоже работают не за $\mathcal{O}(1)$.
6. Вызов функций. Пример, который при $n = 10^7$ работает секунду и использует $\geqslant 320$ `mb`.

```
1 void go(int n) {
2   if (n <= 0) return;
3   go(n - 1); // компилируйте с -O0, чтобы оптимизатор не раскрыл рекурсию в цикл
4 }
```

   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!

2.3. 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 }
```

2.4. 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`.

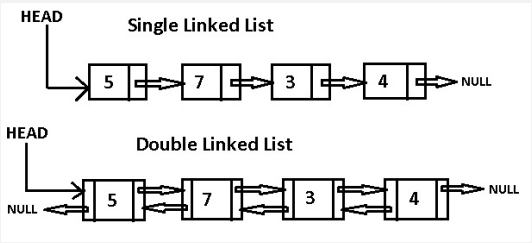
2.5. Двусвязный список

```
1 struct Node {
2   Node *prev, *next; // указатели на следующий и предыдущий элементы списка
3   int x;
4 };
5 struct List {
6   Node *head, *tail; // head, tail - фиктивные элементы
7 };
```

`get(i)`, `set(i,x)`	$\mathcal{O}(i)$
`find(x)`	$\mathcal{O}(n)$
`add_begin(x)`, `add_end(x)`	$\Theta(1)$
`del_begin()`, `del_end()`	$\Theta(1)$
`delete(Node*)`	$\Theta(1)$

Указатель `tail` нужен, чтобы иметь возможность добавлять в конец, удалять из конца за $\mathcal{O}(1)$.
Ссылки `prev`, чтобы ходить по списку в обратном направлении, удалять из середины за $\mathcal{O}(1)$.

```
Node *find(List l, int x) { // найти в списке за линию
    for (Node *p = l.head->next; p != l.tail; p = p->next)
        if (p->x == x)
            return p;
    return 0;
}
Node *erase(Node *v) {
    v->prev->next = v->next;
    v->next->prev = v->prev;
}
Node *push_back(List &l, Node *v) {
    Node *p = new Node();
    p->x = x, p->prev = l.tail->prev, p->next = l.tail;
    p->prev->next = p, p->next->prev = p;
}
void makeEmpty(List &l) { // создать новый пустой список
    l.head = new Node(), l.tail = new Node();
    l.head->next = l.tail, l.tail->prev = l.head;
}
```



2.6. Односвязный список

```
struct Node {
    Node *next; // не храним ссылку назад, нельзя удалять из середины за O(1)
    int x;
};
// 0 - пустой список
Node *head = 0; // не храним tail, нельзя добавлять в конец за O(1)
void push_front(Node* &head, int x) {
    Node *p = new Node();
    p->x = x, p->next = head, head = p;
}
```

2.7. Список на массиве

```
vector<Node> a; // массив всех Node-ов списка
struct {
    int next, x;
};
int head = -1;
void push_front(int &head, int x) {
    a.push_back(Node {head, x});
    head = a.size() - 1;
}
```

Можно сделать свои указатели.
Тогда `next` – номер ячейки массива (указатель на ячейку массива).

2.8. 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 2.8.1. 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 $\mathcal{O}(1)$ ■

2.9. 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; }`

2.10. 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()); }
```

• **Очередь с минимумом через дек минимумов**

Будет разобрано на практике. См. разбор третьей практики.

• **Дек с минимумом через два стека**

Будет решено на практике. См. разбор третьей практики.

Лекция #3: Структуры данных

3-я пара, 2024/25

3.1. Амортизационный анализ

Мы уже два раза оценивали время в среднем – для вектора и очереди с минимумом. Для более сложных случаев есть специальная система оценки «времени работы в среднем», которую называют «амортизационным анализом».

Пусть наша программа состоит из m элементарных операций, i-ая из которых работает t_i.

Def 3.1.1. t_i – *real time (реальное время одной операции)*

Def 3.1.2. $t_{ave} = \frac{\sum_i t_i}{m}$ – *average time (среднее время)*

Def 3.1.3. $a_i = t_i + \Delta\varphi_i$ – *amortized time (амортизированное время одной операции)*

Здесь $\Delta\varphi_i = \varphi_{i+1} - \varphi_i$ = изменение функции φ, вызванное i-й операцией.
a_i – *время, амортизированное функцией* φ. Что за φ?
Можно рассматривать $\forall\varphi$! Интересно подобрать такую, чтобы a_i всегда было небольшим.

• Пример: вектор.

Рассмотрим $\varphi = -size$ (размер вектора, взяли такой потенциал из головы).
1. Нет удвоения: $a_i = t_i + \Delta\varphi_i = 1 + 0 = \mathcal{O}(1)$
2. Есть удвоение: $a_i = t_i + \Delta\varphi_i = size + (\varphi_{i+1} - \varphi_i) = size + (-2size + size) = 0 = \mathcal{O}(1)$

Получили $a_i = \mathcal{O}(1)$, хочется сделать из этого вывод, что $t_{ave} = \mathcal{O}(1)$

• Строгие рассуждения.

Lm 3.1.4. $\sum t_i = \sum a_i - (\varphi_{end} - \varphi_0)$
Доказательство. Сложили равенства $a_i = t_i + (\varphi_{i+1} - \varphi_i)$ ■

Теорема 3.1.5. $t_{ave} = \mathcal{O}(\max a_i) + \frac{\varphi_0 - \varphi_{end}}{m}$
Доказательство. В лемме делим равенство на m, $\sum a_i/m \leqslant \max a_i$, заменяем $\frac{\sum_i t_i}{m}$ на t_{ave} ■

Следствие 3.1.6. Если $\varphi_0 = 0$, $\forall i\ \varphi_i \geqslant 0$, то $t_{ave} = \mathcal{O}(\max a_i)$

• Пример: push, pop(k)

Пусть есть операции `push` за $\mathcal{O}(1)$ и `pop(k)` – достать сразу k элементов за $\Theta(k)$.
Докажем, что в среднем время любой операции $\mathcal{O}(1)$. Возьмём $\varphi = size$
`push`: $a_i = t_i + \Delta\varphi = 1 + 1 = \mathcal{O}(1)$
`pop`: $a_i = t_i + \Delta\varphi = k - k = \mathcal{O}(1)$
Также заметим, что $\varphi_0 = 0$, $\varphi_{end} \geqslant 0$.

• Пример: $a^2 + b^2 = N$

```
1 int y = sqrt(n), cnt = 0;
2 for (int x = 0; x * x <= n; x++)
3     while (x * x + y * y > n) y--;
4     if (x * x + y * y == n) cnt++;
```

Одной операцией назовём итерацию внешнего цикла for.
Рассмотрим сперва корректный потенциал $\varphi = y$.

$a_i = t_i + \Delta\varphi = (y_{old} - y_{new} + 1) + (y_{new} - y_{old}) = \mathcal{O}(1)$
Также заметим, что $\varphi_0 - \varphi_{end} \leqslant \sqrt{n} \overset{Thm\ 3.1.5}{\Rightarrow} t_{ave} = \mathcal{O}(1)$.

Теперь рассмотрим плохой потенциал $\overline{\varphi} = y^2$.
$a_i = t_i + \Delta\overline{\varphi} = (y_{old} - y_{new} + 1) + (y_{new}^2 - y_{old}^2) = \mathcal{O}(1)$
Но, при этом $\varphi_0 = n$, $\varphi_{end} = 0 \overset{Thm\ 3.1.5}{\Rightarrow} t_{ave} = \mathcal{O}(\sqrt{n})$ =(

Теперь рассмотрим другой плохой потенциал $\tilde{\varphi} = 0$.
$a_i = t_i + \Delta\tilde{\varphi} = (y_{old} - y_{new} + 1) = \mathcal{O}(\sqrt{n})$ =(

• **Монетки**

Докажем ещё одним способом, что вектор работает в среднем за $\mathcal{O}(1)$.
Когда мы делаем `push_back` без удвоения памяти, накопим 2 монетки.
Когда мы делаем `push_back` с удвоением $size \to 2size$, это занимает $size$ времени, но мы можем заплатить за это, потратив $size$ накопленных монеток. Число денег никогда не будет меньше нуля, так как до удвоения было хотя бы $\frac{size}{2}$ операций «`push_back` без удвоения».

Эта идея равносильна идее про потенциалы. Мы неявно определяем функцию φ через её $\Delta\varphi$. φ – количество накопленных и ещё не потраченных монеток. $\Delta\varphi =$ соответственно +2 и $-size$.

3.2. Разбор арифметических выражений

Разбор выражений с числами, скобками, операциями.
Предположим, все операции левоассоциативны (вычисляются слева направо).
Решение: идти слева направо, поддерживать два — необработанные операции и аргументы.
Приоритеты: `map<char,int> priority = {{'+':1},{'-':1},{'*':2},{'/':2},{'(':-1}};`
Почему у «(» такой маленький? Чтобы, пока «(» лежит на стеке, она точно не выполнилась.

```
stack<int> value; // уже посчитанные значения
stack<char> op; // ещё не выполненные операции
void make(): // выполнить последнюю невыполненную операцию
    int b = value.top(); value.pop();
    int a = value.top(); value.pop();
    char o = op.top(); op.pop();
    value.push(a o b); // да, не скомпилится, но смысл такой
int eval(string s): // пусть s без пробелов
    s = '(' + s + ')' // при выполнении последней ')', выражение вычислится
    for (char c : s)
        if ('0' <= c && c <= '9') value.push(c - '0'); // просто добавили
        else if (c == '(') op.push(c); // просто добавили, её приоритет меньше всех
        else if (c == ')') { // закрылась? ищем парную открывающую на стеке
            while (op.top() != '(') make();
            op.pop();
        } else { // пришла операция? можно выполнить все предыдущие большего приоритета
            while (op.size() && prioriоty[op.top()] >= prioriоty[c]) make();
            op.push(c);
        }
    return value.top();
```

Теорема 3.2.1. Время разбора выражения s со стеком равно $\Theta(|s|)$

Доказательство. В функции `eval` число вызовов `push` не больше $|s|$. Операция `make` уменьшает размер стеков, поэтому число вызовов `make` не больше числа операций `push` в функции `eval`. ■

3.3. Бинпоиск

3.3.1. 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 $\mathcal{O}(\log n)$ искать в этом массиве элемент x

```
int find(int l, int r, int x): // [l,r]
    while (l <= r) {
        int m = (l + r) / 2;
        if (a[m] == x) return m;
        if (a[m] < x) l = m + 1;
        else r = m - 1;
    return -1;
```

Lm 3.3.1. Время работы $\mathcal{O}(\log n)$

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

• Задача про нули и единицы.

Решим похожую задачу: есть монотонный массив `a[0..n-1]` из нулей и единиц (сперва идут нули, затем единицы). Пример: `0000111111111`. *Задача:* найти позицию последнего нуля и первой единицы.

Решение бинпоиском: чтобы в массиве точно был хотя бы один ноль и хотя бы одна единица, мысленно припишем `a[-1] = 0, a[n] = 1`, поставим указатели `L = -1, R = n` и будем следить, чтобы всегда было `a[L] = 0, a[R] = 1`. Как и выше `L` и `R` сближаются, на каждом шаге отрезок сужается в два раза.

```
while (R - L > 1) {
    int m = (L + R) / 2; // 0 ⩽ m < n ⇒ нет выхода за пределы a[]
    if (a[m] == 0) // можно просто (!a[m] ? L : R) = m;
        L = m;
    else
        R = m;
} // после бинпоиска R-L=1,a[L]=0,a[R]=1
```

3.3.2. Lowerbound и Upperbound

Задача: дан сортированный массив, найти $\min i\colon a_i \geqslant x$.

Например `a`: `1 2 2 2 3 3 7`, `lower_bound(3)`: `1 2 2 2 3 3 7`, $i = 4$.
Сведём задачу к предыдущей (нули и единицы): $a_i < x$ нули, $a_i \geqslant x$ единицы.
Пишем ровно такой же бинпоиск, как выше, но условие «`a[m] == 0`» меняется на `a[m] < x`.

```
int lower_bound(int l, int r, int x): // [l,r)
    while (R - L > 1) {
        int m = (L + R) / 2;
        if (a[m] < x)
            L = m;
        else
            R = m;
    } // после бинпоиска R-L=1, a[L]<x, a[R]⩾x
```

Заметим, что этот бинпоиск строго мощнее чем наш первый `find`:
`find(l, r, x): return a[lower_bound(l, r, x)] == x;`

В языке C++ есть стандартные функции

```
int a[n]; // массив из n элементов
i = lower_bound(a, a + n, x) - a; // min i: a[i] >= x
i = upper_bound(a, a + n, x) - a; // min i: a[i] > x
vector<int> a(n);
i = lower_bound(a.begin(), a.end(), x) - a.begin(); // начало и конец вектора
```

Зачем нужно две функции, что они делают? 1 1 2 2 2 3 3 7 → 1 1 2 2 2 3 3 7, находят первое и последнее вхождение числа в сортированный массив, а их разность – число вхождений.

Ещё можно найти $\boxed{\max i\colon a_i \leqslant x}$ = `upper_bound` − 1 и $\boxed{\max i\colon a_i < x}$ = `lower_bound` − 1.

3.3.3. Бинпоиск по предикату

Предикат – функция, которая возвращает только 0 и 1.
Наш бинпоиск на самом деле умеет искать по любому монотонному предикату.
Мы можем найти такие $l + 1 = r$, что $f(l) = 0, f(r) = 1$.
Например, Выше мы искали по предикату $f(i) =$ (`a[i] <= x ? 0 : 1`).

```
void find_predicate(int &l, int &r): // изначально 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;
```

Пример, как с помощью `find_predicate` сделать `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
```

3.3.4. Вещественный, корни многочлена

Дан многочлен P нечётной степени со старшим коэффициентом 1. У него есть вещественный корень и мы можем его найти бинарным поиском с любой наперёд заданной точностью ε.

Сперва нужно найти точки l, r: $P(l) < 0,\ P(r) > 0$.

```
for (l = -1; P(l) >= 0; l *= 2) ;
for (r = +1; P(r) <= 0; r *= 2) ;
```

Теперь собственно поиск корня:

```
while (r - l > ε)
    double m = (l + r) / 2;
    (P(m) < 0 ? l : r) = m;
```

Внешний цикл может быть бесконечным из-за погрешности ($l{=}10^9, r{=}10^9{+}10^{-6}, \varepsilon{=}10^{-9}$)
Чтобы он точно завершился, посчитаем, сколько мы хотим итераций: $k = \log_2 \frac{r-l}{\varepsilon}$, и сделаем ровно k итераций: `for (int i = 0; i < k; i++)`.
Поиск всех вещественных корней многочлена степени n будет в 6-й практике (см. разбор).

3.4. Два указателя и операции над множествами

Множества можно хранить в виде отсортированных массивов. Наличие элемента в множестве можно проверять бинпоиском за $\mathcal{O}(\log n)$, а элементы перебирать за линейное время.

Также, зная A и B, за линейное время методом «двух указателей» можно найти $A \cap B, A \cup B, A \setminus B$, объединение мультимножеств.

В языке C++ это операции `set_intersection`, `set_union`, `set_difference`, `merge`.
Все они имеют синтаксис `k = merge(a, a+n, b, b+m, c)- c`, где k – количество элементов в ответе, c – указатель «куда сохранить результат». Память под результат должны выделить вы сами.

Пример применение «двух указателей» для поиска пересечения.

Вариант #1, for:

```
B[|B|] = +∞; // барьерный элемент
for (int k = 0, j = 0, i = 0; i < |A|; i++)
    while (B[j] < A[i]) j++;
    if (B[j] == A[i]) C[k++] = A[i];
```

Вариант #2, while:

```
int i = 0, j = 0;
while (i < |A| && j < |B|)
    if (A[i] == B[j]) C[k++] = A[i++], j++;
    else (A[i] < B[j] ? i : j)++;
```

3.5. 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 $\mathcal{O}(1)$.

3.5.1. 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 3.5.1. N — случайное простое $\Rightarrow$ хеш-функция `x` $\to$ `x % N` достаточно хорошая.

Без доказательства. Мы утверждаем хорошесть только для списочной хеш-таблицы.

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 $\mathcal{O}(1)$.

3.5.2. 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 3.5.2. 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) $= 1 + \alpha + \alpha^2 + \alpha^3 + \dots = \frac{1}{1-\alpha}$. ■

Lm 3.5.3. $\exists$ тест такой, что $\forall N$ хеш-функция `x` $\to$ `x % N` **плоха.**

Доказательство. Тест: добавляем числа $0, 2, \dots n$ и $r, r{+}1, r{+}2, \dots r{+}n$, где $r = random$. ■

Утверждение 3.5.4. Пусть N — любое фиксированное простое, а $r = random[1, N{-}1]$, фиксированное число, тогда: хеш-функция `x` $\to$ `(x·r) % N` достаточно хорошая.

Без доказательства. Докажем в следующем семестре в теме «универсальное семейство».

• 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.

3.5.3. 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.

3.5.4. 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 3.5.5. 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 программе анти-хеш теста.

Лекция #4: Структуры данных

4-я пара, 2024/25

4.1. Избавляемся от амортизации

Серьёзный минус вектора – амортизированное время работы. Сейчас мы модифицируем структуру данных, она начнёт чуть дольше работать, использовать чуть больше памяти, но время одной операции в худшем будет $\mathcal{O}(1)$.

4.1.1. Вектор (решаем проблему, когда случится)

В тот `push_back`, когда старый вектор `a` переполнился, выделим память под новый вектор `b`, новый элемент положим в `b`, копировать `a` пока не будем. Сохраним `pos = |a|`.

Инвариант: первые `pos` элементов лежат в `a`, все следующие в `b`. Каждый `push_back` будем копировать по одному элементу.

```
int *a, *b; // выделенные области памяти
int pos = -1; // разделитель скопированной и не скопированной частей
int n, size; // количество элементов; выделенная память
void push_back(int x):
    if (pos >= 0) b[pos] = a[pos], pos--;
    if (n == size):
        delete [] a; // мы его уже скопировали, он больше не нужен
        a = b;
        size *= 2;
        pos = n - 1, b = new int[size];
    b[n++] = x;
```

Как мы знаем, `new` работает за $\mathcal{O}(\log n)$, это нас устроит.
Тем не менее в этом месте тоже можно получить $\mathcal{O}(1)$.

Lm 4.1.1. К моменту `n == size` вектор `a` целиком скопирован в `b`.

Доказательство. У нас было как минимум n операций `push_back`, каждая уменьшала `pos`. ■

Операция обращения к i-му элементу обращается теперь к `(i <= pos ? a : b)`.

Время на копировани не увеличилось. Время обращения к i-му элементу чуть увеличилось (лишний `if`). Памяти в среднем теперь нужно в 1.5 раз больше, т.к. мы в каждый момент храним и старую, и новую версию вектора.

4.1.2. Вектор (решаем проблему заранее)

Сделаем так, чтобы время обращения к i-му элементу не изменилось.
Мы начнём копировать заранее, в момент `size = 2n`, когда вектор находится в нормальном состоянии. Нужно к моменту очередного переполнения получить копию вектора в память большего размера. За `n` `push_back`-ов должны успеть скопировать все `size = 2n` элементов. Поэтому будем копировать по 2 элемента. Когда в такой вектор мы записываем новые значения (`a[i]=x`), нам нужно записывать в обе версии – и старую, и новую.

4.1.3. Сравнение способов

Чтение `a[i]` во 2-м способе быстрее:
во 2-м способе всегда обратимся к старой версии,
в 1-м способе `if (i < pos) a[i] else b[i]`

Запись `a[i]=x` в 1-м способе быстрее:
в 1-м способе записать в одну из двух версий,
во 2-м способе нужно писать в обе версии.

Память: в 1-м способе меньше пустых ячеек.

Как мы увидим на примере очереди с минимумом, 2-й способ более универсальный.

4.1.4. Хеш-таблица

Хеш-таблица – ещё одна структура данных, которая при переполнении удваивается. К ней можно применить оба описанных подхода. Применим первый. Чтобы это сделать, достаточно научиться перебирать все элементы хеш-таблицы и добавлять их по одному в новую хеш-таблицу.

(а) Можно кроме хеш-таблицы дополнительно хранить «список добавленных элементов».
(б) Можно пользоваться тем, что число ячеек не более чем в два раза больше числа элементов, поэтому будем перебирать ячейки, а из них выбирать не пустые.

Новые элементы, конечно, мы будет добавлять только в новую хеш-таблицу.

4.1.5. Очередь с минимумом через два стека

Напомним, что есть очередь с минимумом.

```
Stack a, b;
void push(int x) { b.push(x); }
int pop():
    if (a.empty()) // стек a закончился, пора перенести элементы b в a
        while (b.size())
            a.push(b.pop());
    return a.pop();
```

Воспользуемся вторым подходом «решаем проблему заранее». К моменту, когда стек a опустеет, у нас должна быть уже готова перевёрнутая версия b. Вот общий шаблон кода.

```
Stack a, b, a1, b1;
void push(int x):
    b1.push(x); // кидаем не в b, а в b1, копию b
    STEP; // сделать несколько шагов копирования
int pop():
    if (копирование завершено)
        a = a1, b = b1, начать новое копирование;
    STEP; // сделать несколько шагов копирования
    return a.pop();
```

Почему нам вообще нужно копировать внутри `push`? Если мы делаем сперва 10^6 `push`, затем 10^6 `pop`, к моменту всех этих `pop` у нас уже должен быть подготовлен длинный стек `a`. Если в течение `push` мы его не подготовили, его взять неоткуда.

При копировании мы хотим построить новый стек `a1` по старым `a` и `b` следующим образом (`STEP` сделает несколько шагов как раз этого кода):

```
1 while (b.size()) a1.push(b.pop());
2 for (int i = 0; i < a.size(); i++) a1.push(a[i]);
```

Заметим, что `a.size()` будет меняться при вызовах `a.pop_back()`. `for` проходит элементы `a` снизу вверх. Так можно делать, если стек `a` реализован через вектор без амортизации. Из кода видно, что копирование состоит из $|a| + |b|$ шагов. Будем поддерживаем инвариант, что до начала копирования $|a| \geqslant |b|$. В каждом `pop` будем делать 1 шаг копирования, в каждом `push` также 1 шаг. Проверка инварианта после серии `push`: за k пушей мы сделали $\geqslant k$ копирований, поэтому $|a_1| \geqslant |b_1|$. Проверка корректности `pop`: после первых $|b|$ операций поп все элементы b уже скопировались, далее мы докопируем часть `a`, которая не подверглась `pop_back`-ам.

4.2. 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 4.2.1. *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 4.2.2. 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()`. Извлеяение (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.

4.2.1. 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 // ExtractMin 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 void siftDown(int i):
5     while (1):
6         int l = 2 * i;
7         if (l + 1 <= n && a[l + 1] < a[l]) l++; // выбрать меньшего из детей
8         if (!(l <= n && a[l] < a[i])) break; // если все дети не меньше нас, это конец
9         swap(a[l], a[i]), i = l; // перейти в ребёнка
```

Lm 4.2.3. Обе процедуры корректны

Доказательство. 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 4.2.4. Обе процедуры работают за $\mathcal{O}(\log n)$

Доказательство. Они работают за высоту кучи, которая по Lm 4.2.2 равна $\mathcal{O}(\log n)$. ■

4.2.2. Обратные ссылки и DecreaseKey

Давайте предположим, что у нас есть массив значений: `vector<int> value`.
В куче будем хранить индексы этого массива. Тогда все сравнения `a[i] < a[j]` следует заменить на сравнения через `value`: `value[a[i]] < value[a[j]]`. Чтобы добавить элемент, теперь нужно сперва добавить его в конец `value`: `value.push_back(x)`, а затем сделать добавление в кучу `Add(value.size() - 1)`. Хранение индексов позволяет нам для каждого `i` помнить позицию в куче `pos[i]`: `a[pos[i]] == i`. Значения `pos[]` нужно пересчитывать каждый раз, когда мы меняем значения `a[]`. Как теперь удалить произвольный элемент с индексом `i`?

```
1 void Del(int i):
2     i = pos[i];
3     a[i] = a[n--], pos[a[i]] = i; // не забыли обновить pos
4     siftUp(i), siftDown(i); // новый элемет может быть и меньше, и больше
```

Процедура `DecreaseKey(i)` делается похоже: перешли к `pos[i]`, сделали `siftUp`.

Lm 4.2.5. `Del` и `DecreaseKey` корректны и работают за $\mathcal{O}(\log n)$

Доказательство. Следует из корректности и времени работы `siftUp`, `siftDown` ■

Благодаря обратным ссылкам мы получили структуру данных, которая умеет обрабатывать запросы: `value[i]=x`, `getMin(value)`, `value.push_back(x)`, `extractMin`.
Решение: «`push_back`» = `add`, «`a[i]=x`» = `del(i)`, поменять `value[i])`, `add(i)`.

4.2.3. Build, HeapSort

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

Lm 4.2.6. 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 4.2.7. 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 $\sum_{i=1}^{k} 2^{k-i} i = 2^k \sum_{i=1}^{k} \frac{i}{2^i} \overset{(*)}{=} 2^k \cdot \Theta(1) = \Theta(n)$. `(*) доказано на практике.` ■

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

Lm 4.2.8. 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`. ■

4.3. Аллокация памяти

Нам дали много памяти. А конкретно `MAX_MEM` байт: `uint8_t mem[MAX_MEM]`. Мы – менеджер памяти. Мы должны выделять, когда надо, освобождать, когда память больше не нужна.

Задача: реализовать две функции

1. `int new(int x)` выделяет `x` байт, возвращает адрес первой свободной ячейки
2. `void delete(int addr)` освобождает память, по адресу `addr`, которую когда-то вернул `new`

В общем случае задача сложная. Сперва рассмотрим популярное решение более простой задачи.

4.3.1. Стек

Разрешим освобождать не любую область памяти, а **только последнюю выделенную**. Тогда сделаем из массива `mem` стек: первые `pos` ячеек — занятая память, остальное свободно. Выделить n байт: `pos += n;`. Освободить последние n байт: `pos -= n;`. Код:

```
int pos = 0; // указатель на первую свободную ячейку
int new(uint32_t n): // push n bytes
    pos += n;
    assert(pos <= MAX_MEM); // проверить, что памяти всё ещё хватает
    return pos - n;
void delete(uint32_t old_pos): // освободили всю память, выделенную с момента old_pos
    pos = old_pos; // очищать можно только последнюю выделенную
```

В C++ при вызове функции, при создании локальных переменных используется ровно такая же модель аллокации память, называется также — «стек». Иногда имеет смысл реализовать свой стек-аллокатор и перегрузить глобальный `operator new`, так как стандартные STL-контейнеры `vector`, `set` внутри много раз обращаются к медленному `operator new`.

Эффект ощутим: `vector<vector<int>> a(10,000,000)` ускоряется в 4 раза.
[code], [vector-experiment]

4.3.2. Список

Ещё один частный простой случай `x = CONST`, все выделяемые ячейки одного размера.
Идея: разобьём всю память на куски по `x` байт. Свободные куски образуют список (односвязный), мы храним голову этого списка. *Выделить память:* откусить голову списка. *Освобождение памяти:* добавить в начало списка. Подробнее + детали реализации:

Пусть наше адресуемое пространство 32-битное, то есть, `MAX_MEM` $\leqslant 2^{32}$. Тогда давайте исходные `MAX_MEM` байт памяти разобьём на 4 байта `head` и на $k = \lfloor \frac{\texttt{MAX\_MEM}-4}{\max(x,4)} \rfloor$ ячеек по $\max(x, 4)$ байт. Каждая из k ячеек или свободная, тогда она – «указатель на следующую свободную», или занята, тогда она – «x байт полезной информации». `head` – начало списка свободных ячеек, первая свободная. Изначально все ячейки свободны и объединены в список.

```
const uint32_t size; // размер блока, size >= 4
uint32_t head = 0; // указатель на первый свободный блок
uint8_t mem[MAX_MEM-4]; // часть памяти, которой пользуется new
uint32_t* pointer(uint32_t i) { return (uint32_t*)(mem+i); } // magic =)
void init():
    for (uint32_t i = 0; i + size <= MAX_MEM-4; i += size)
        *pointer(i) = i + size; // указываем на следующий блок

uint32_t new(): // вернёт адрес в нашем 32-битном пространстве mem
    uint32_t res = head;
    head = *pointer(head); // следующий свободный блок
    return res;

void delete(uint32_t x):
    *pointer(x) = head; // записали в ячейки [x..x+4) старый head
    head = x;
```

4.3.3. Куча (кратко)

В общем случае (выделяем сколько угодно байт, освобождаем память в любом порядке) массив `mem` разбит на отрезки свободной памяти и отрезки занятой памяти. Какой свободный отрезок памяти использовать, когда просят выделить n байт? Любой длины $\geqslant x \Rightarrow$ максимальный подойдёт $\Rightarrow$ отрезки свободной памяти будем хранить в куче по длине (в корне максимум).

• **Операция** `new(x)`.

Если в корне кучи максимум меньше x, память не выделяется.
Иначе память выделяется за $\mathcal{O}(1) + \langle$`время просеивания вниз в куче`$\rangle = \mathcal{O}(\log n)$.

• **Операция** `delete(addr)`.

Нужно понять про отрезки слева/справа от `addr` — заняты они, или свободны.
Если свободны, узнать их длину, удалить из кучи, добавить новый большой свободный отрезок.

4.3.4. (*) Куча (подробно)

Сделаем 32-битную версию (все указатели по 4 байта).
Будем думать о нашей памяти, как о массиве M: `uint32_t M[SIZE]`.
Храним кучу свободных кусков. Пусть все свободные куски имеют размер *хотя бы* 8 байт.

(a) Память = служебная информация + пользовательская память.
(b) Первые $4 + n \cdot 4$ байт = хранение n + собственно куча.
(c) Изначально заняты только 8 байт, под $n = 1$ и ровно 1 свободный блок.
При росте n откусываем место от свободного куска справа от кучи.
(d) Каждая ячейка кучи — 4 байта, указатель на начало свободного блока.
Где хранить размер? В первых 4 байтах этого блока.
(e) `new(size)`: смотрим корень кучи `M[1]`, если `M[M[1]] >= size`, возвращаем `size` *последних* байт: `M[1]+M[M[1]]-size`, уменьшаем блок `M[1]` на `size`, вызываем `siftDown`.
(f) `delete(addr,size)`. Если соседи — не свободные куски, добавим новый элемент в кучу.
Для этого расширим кучу, откусим 4 байта смежного с кучей свободного куска. Если соседи — пустые куски, объединим нас и их в один большой кусок, сделаем siftUp в куче.
(g) Хотим делать `delete(addr)` (не передавать `size`)? $\Rightarrow$ в каждом занятом блоке нужно резервировать +4 байта под `size`.

Как выделять память под кучу? Можно заранее фиксировать N и выделить $4N$ байт памяти. Можно надеяться, что смежный с кучей кусок свободен, и при `n++` отщеплять от него очередные 4 байта. Можно по образу вектора с удвоением по надобности выделять память, используя себя же, как источник памяти. Выше выбран второй вариант.

Как понять, пусты ли соседи? Пусть каждый свободный кусок хранит: первые 4 байта = размер куска, последние 4 байта = обратная ссылка (индекс куска в куче). Смотрим на соседей, пытаемся их интерпретировать как свободные, за $\mathcal{O}(1)$ проверяем, что место, на которое они указали в куче, хранит именно их. *Например,* нам дали адрес `A`:
`l=M[A-1]; if (1<=l<=n and M[l]+M[M[l]]==A)` то слева от нас свободный кусок.

4.3.5. (*) Дефрагментация

На входе *дефрагментации* используемая память = набор мелких отрезков, на выходе мы хотим, чтобы вся используемая память шла подряд (образовывала один отрезок). Это делают для жёстких дисков. Это же мы можем сделать и при аллокации оперативной памяти.

Стековый аллокатор можно переделать в универсальный.
Идея: освободить памяти = лениво пометить ячейку, как свободную. Когда память кончилась, делаем дефрагментацию: пройдёмся за линию двумя указателями, оставим только реально существующие ячейки. Чтобы это работало нам нужно уметь подменить уже существующие указатели на новые + помечать ячейки, как свободные.

С аллокатором кучей можно иногда делать то же. Там это не столь критично,
но тоже ценно в ситуации, например, `010101...01` (0 — свободная ячейка).

4.4. 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.

4.4.1. 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();
```

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`.

4.4.2. 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.

4.4.3. Add → Merge

Merge (слияние) – операция, получающая на вход две структуры данных, на выход даёт одну, равную их объединению. Старые структуры объявляются невалидными.
Пример #1. Merge двух сортированных массивов.
Пример #2. Merge двух куч. Сейчас мы научимся делать его быстро.

• **Идея.** У нас есть операция добавления одного элемента, переберём все элементы меньшей структуры данных и добавим их в большую.

```
1 Heap Merge(Heap a, Heap b):
2     if (a.size < b.size) swap(a, b);
3     for (int x : b) a.Add(x);
4     return a;
```

Lm 4.4.1. Если мы начинаем с $\varnothing$ и делаем N произвольных операций из множества {`Add`, `Merge`}, функция `Add` вызовется не более $N \log_2 N$ раз.

Доказательство. Посмотрим на код и заметим, что $|a| + |b| \geqslant 2|b|$, поэтому для каждого x, переданного `Add` верно, что «размер структуры, в которой живёт x, хотя бы удвоился» $\Rightarrow \forall x$ количество операций `Add`(x) не более $\log_2 N \Rightarrow$ суммарное число всех `Add` не более $N \log_2 N$. ■

4.4.4. Build → Add

Хотим взять структуру данных, которая умеет только `Build` и `Get`, научить её `Add`.

• **Пример задачи**

Структура данных: сортированный массив.
Построение (`Build`): сортировка за $\mathcal{O}(n \log n)$.
Запрос (`Get`): количество элементов со значением от L до R, два бинпоиска за $\mathcal{O}(\log n)$

• **Решение #1. Корневая.**

Структура данных: храним два сортированных массива –
большой a (старые элементы) и маленький b (новые элементы), поддерживаем $|b| \leqslant \sqrt{|a|}$.

Новый `Get(L,R)`: `return a.Get(L,R) + b.Get(L,R)`

`Add(x)`: кинуть x в b; вызвать `b.Build()`;
если $|b|$ стало больше $\sqrt{|a|}$, перенести все элементы b в a и вызвать `a.Build()`.

Замечание 4.4.2. В данной конкретной задаче можно вызов пересортировки за $\mathcal{O}(n \log n)$ заменить на merge за $\mathcal{O}(n)$. В общем случае у нас есть только `Build`.

Обозначим `Build(m)` – время работы функции от m элементов, $n = |a|$.
Между двумя вызовами `a.Build()` было $\sqrt{n}$ вызовов `Add` $\Rightarrow$ $\sqrt{n}$ операций `Add` отработали за $\mathcal{O}(\texttt{Build}(n) + \sqrt{n} \cdot \texttt{Build}(\sqrt{n})) = \mathcal{O}(\texttt{Build}(n))$ (для выпуклых функций `Build`) $\Rightarrow$ среднее время работы $\texttt{Add} = \mathcal{O}(\texttt{Build}(n)/\sqrt{n}) = \mathcal{O}(\sqrt{n} \log n)$.

• **Решение #2. Пополняемые структуры.**

Пусть у нас есть структура `S` с интерфейсом `S.Build`, `S.Get`, `S.AllElements`. У любого числа N есть единственное представление в двоичной системе счисления $a_1 a_2 \ldots a_k$ Для хранения $N =$

$2^{a_1}+2^{a_2}+\cdots+2^{a_k}$ элементов будем хранить k структур `S` из $2^{a_1}, 2^{a_2}, \ldots, a^{a_k}$ элементов. $k \leqslant log_2 n$. Новый `Get` работает за $k \cdot$ `S.Get`, обращается к каждой из k частей. Сделаем `Add(x)`. Для этого добавим ещё одну структуру из 1 элемента. Теперь сделаем так, чтобы не было структур одинакового размера.

```
for (i = 1; есть две структуры размера i; i *= 2)
    Добавим S.Build(A.AllElements + B.AllElements). // A, B - те самые две структуры
    Удалим две старые структуры
```

Заметим, что по сути мы добавляли к числу N единицу в двоичной системе счисления.

Lm 4.4.3. Пусть мы начали с пустой структуры, было n вызовов `Add`. Пусть эти n `Add` k раз дёрнули `Build`: `Build`(a_1), `Build`(a_2), $\ldots$, `Build`(a_k). Тогда $\sum_{i=1}^{k} a_i \leqslant n \log_2 n$

Доказательство. Когда элемент проходит через `Build` размер структуры, в которой он живёт, удваивается. Поэтому каждый x пройдёт через `Build` не более $\log_2 n$ раз. ■

Lm 4.4.4. Суммарное время обработки n запросов не более `Build`$(n \log_2 n)$

Доказательство. Чтобы получить эту лемму из предыдущей, нужно наложить ограничение «выпуклость» на время работы `Build`. ■

Lm 4.4.5. $\forall k \geqslant 1, a_i > 0$: $(\sum a_i)^k \geqslant \sum a_i^k$ (без доказательства)

Лемма постулирует «полиномы таки выпуклы, поэтому к ним можно применить Lm 4.4.4».

Применение этой идеи для сортированного массива будем называть *«пополняемый массив»*.

4.4.5. Build → Add, Del

Научим «пополняемый массив обрабатывать запросы».

1. `Count(l, r)` – посчитать число x: $l \leqslant x \leqslant r$
2. `Add(x)` – добавить новый элемент
3. `Del(x)` – удалить ранее добавленный элемент

Для этого будем хранить два «пополняемых массива» – добавленные элементы, удалённый элементы. Когда нас просят сделать `Count`, возвращаем разность `Count`-ов за $\mathcal{O}(\log^2 n)$. `Add` и `Del` работают амортизированно за $\mathcal{O}(\log n)$, так как вместо `Build`, который должен делать `sort`, мы вызовем `merge` двух сортированных массивов за $\mathcal{O}(n)$.

Лекция #5: Сортировки

5-я пара, 2024/25

5.1. Два указателя и алгоритм Мо

• **Задача:** дан массив длины n и m запросов вида
«количество различных чисел на отрезке $[l_i, r_i]$».

Если $l_i \leqslant l_{i+1}, r_i \leqslant r_{i+1}$ – это обычный метод двух указателей с хеш-таблицей внутри. Решение работает за $\mathcal{O}(n+m)$ операций с хеш-таблицей. Такую идею можно применить и к другим типам запросов. Для этого достаточно, зная ответ и поддерживая некую структуру данных, для отрезка $[l, r]$ научиться быстро делать операции `l++`, `r++`.

Если же l_i и r_i произвольны, то есть решение за $\mathcal{O}(n\sqrt{m})$, что,
конечно, хуже $\mathcal{O}(n+m)$, но гораздо лучше обычного $\mathcal{O}(nm)$.

• **Алгоритм Мо**

Во-первых, потребуем теперь четыре типа операций: `l++`, `r++`, `l--`, `r--`.

Зная ответ для $[l_i, r_i]$, получить ответ для $[l_{i+1}, r_{i+1}]$ можно за $|l_{i+1}-l_i| + |r_{i+1}-r_i|$ операций.
Осталось перебирать запросы в правильном порядке, чтобы $\sum_i \big(|l_{i+1} - l_i| + |r_{i+1} - r_i|\big) \to \min$.
Чтобы получить правильный порядок, отсортируем отрезки по $\langle \lfloor \frac{l_i}{k} \rfloor, r_i \rangle$, где k – константа, которую ещё предстоит подобрать. После сортировки пары $\langle l_i, r_i \rangle$ разбились на $\frac{n}{k}$ групп (по l_i).

Посмотрим, как меняется l_i. Внутри группы $|l_{i+1} - l_i| \leqslant k$, при переходе между группами (движение только вперёд) $\sum |l_{i+1} - l_i| \leqslant 2n$. Итого $mk{+}2n$ шагов l_i. Посмотрим, как меняется r_i. Внутри группы указатель r сделает в сумме $\leqslant n$ шагов вперёд, при переходе между группами сделает $\leqslant n$ шагов назад. Итого $2n\frac{n}{k}$ шагов r_i. Итого $\Theta(m + (mk{+}2n) + \frac{n}{k}n)$ операций.

Подбираем k: $f + g = \Theta(\max(f, g))$, при этом с ростом k $\quad f = mk\nearrow$, g $= \frac{n}{k}n\searrow$ $\quad \Rightarrow$
оптимально взять k: $mk = \frac{n}{k}n \Rightarrow k = (n^2/m)^{1/2} = n/m^{1/2} \Rightarrow$
время работы $mk + \frac{n}{k}n = n\sqrt{m} + n\sqrt{m} \Rightarrow$ общее время работы $\Theta(m + n\sqrt{m})$.

5.2. Квадратичные сортировки

Def 5.2.1. *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 5.2.2. *Инверсия – пара* $i < j$: $a_i > a_j$

Def 5.2.3. I *– обозначение для числа инверсий в массиве*

Lm 5.2.4. Массив отсортирован $\Leftrightarrow$ `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]);
```

• **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.*Можно ускорить сортировку:* место для вставки искать бинпоиском. Сортировка всё равно останется квадратичной, но число сравнений станет $n \log n$.

• **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 max элемент встаёт на своё место, «всплывает». Модификация: ShakerBubble, чередовать направление внутреннего цикла.

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

• **Ценность сортировок.**

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.

5.3. Оценка снизу на время сортировки

Если сортировке объектов разрешено общаться с этими объектами единственным способом – сравнивать их на больше/меньше, про неё говорят *основана на сравнениях*.

Lm 5.3.1. 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 – «<» (0) или «>» (1). Сортировка получает k бит информации, и результат её работы зависит только от этих k бит $\Rightarrow$ если для двух перестановок она получит одни и те же k бит, одну из этих двух перестановок она отсортирует неправильно $\Rightarrow \forall$ корректной сортировки $2^k \geqslant n! \Leftrightarrow k \geqslant \log(n!) = \Theta(n \log n)$. ■

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.

5.4. Решение задачи по пройденным темам

Задача: даны два массива, содержащие множества, найти размер пересечения.
Решения:

1. Отсортировать первый массив, бинпоиском найти элементы второго. $\mathcal{O}(n \log n)$.
2. Отсортировать оба массива, пройти двумя указателями. $\mathcal{O}(sort)$.
3. Элементы одного массива положить в хеш-таблицу, наличие элементов второго проверить. $\mathcal{O}(n)$, но требует $\Theta(n)$ доппамяти, и имеет большую константу.

5.5. Быстрые сортировки

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

5.5.1. 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         output(x);
```

Мы уже знаем, что сортировки, основанные на сравнениях не могут работать за $\mathcal{O}(n)$. В данном случае мы пользуемся ещё и `count[x]++`, это возможно только при сортировке целых чисел. Именно это даёт ускорение.

5.5.2. MergeSort (сортировка слиянием)

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

```
void MergeSort(int l, int r, int *a, int *buffer): // [l, r)
    if (r - l <= 1) return;
    int m = (l + r) / 2;
    MergeSort(l, m, a, buffer);
    MergeSort(m, r, a, buffer);
    Merge(l, m, r, a, buffer); // слияние за Θ(r−l), используем буффер
```

`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 5.5.1. 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

Представим, что $n = 2^m$. В рекурсивной версии мы обходим дерево рекурсии сверху вниз. Снизу у нас куски массива длины 1, чуть выше 2, 4 и т.д. Давайте перебирать те же самые вершины дерева рекурсии снизу вверх нерекурсивно:

```
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, buffer); // O(1)
return a; // результат содержится именно тут, указатель может отличаться от исходного a
```

Этот код лучше тем, что нет копирования буфера $(-C_1 \cdot n \log n)$ и нет рекурсии $(-C_2 \cdot n)$.

5.5.3. 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 5.5.2. 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 5.5.3. $a[l, j] \leqslant x$, $a(j, i) = x$, $a[i, r] \geqslant x$

Замечание 5.5.4. 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 5.5.5. 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);
    QuickSort(l, j, a); // j < i
    QuickSort(i, r, a); // j < i
```

5.5.4. Сравнение сортировок

Название	Время	`space`	`stable`
HeapSort	$\mathcal{O}(n \log n)$	$\Theta(1)$	-
MergeSort	$\Theta(n \log n)$	$\Theta(n)$	+
QuickSort	$\mathcal{O}(n \log n)$	$\Theta(\log n)$	-

Существует ли стабильная (stable) сортировка, работающая за $\mathcal{O}(n \log n)$, не использующая дополнительную память (inplace). Среди уже изученных такой нет, но вообще такая $\exists$, она получается на основе `MergeSort` и inplace `Merge` за $\mathcal{O}(n)$. На практике сделаем `inplace stable merge` за $\mathcal{O}(n \log n)$.

5.6. (*) Adaptive Heap sort

Цель данного блока, предявить хотя бы одну сортировку, основанную на сравнениях, которая в худшем, конечно, за $\Theta(n \log n)$, но на почти отсортированных массивах работает сильно быстрее.

5.6.1. (*) Модифицированный HeapSort

Работает за $\mathcal{O}(n \log n)$, но бывает быстрее. Сначала построим кучу h за $\mathcal{O}(n)$, а еще создадим кучу кандидатов на минимальность C. Изначально C содержит только корень кучи h. Теперь n раз делаем: `x = C.extractMin`, добавляем `x` в конец отсортированного массива, добавляем в кучу C детей `x` в куче h. Размер кучи C в худшем случае может быть $\frac{n+1}{2}$, но в лучшем (когда минимальные элементы в куче h лежат в порядке обхода dfs-а) он не превышает $\log n \Rightarrow$ на некоторых входах можно добиться времени работы порядка $\mathcal{O}(n \log \log n)$.

5.6.2. (*) Adaptive Heap Sort

Алгоритм создан в 1992 *году.*

Берём массив a и рекурсивно строим бинарное дерево: корень = минимальный элемент, левое поддерево = рекурсивный вызов от левой половины, правое поддерево = рекурсивный вызов от правой половины. Полученный объект обладает свойством кучи. На самом деле, мы построили декартово дерево на парах $(x_i{=}i, y_i{=}a_i)$. $\exists$ простой алгоритм со стеком построения декартова дерева за $\mathcal{O}(n)$, мы его изучим во 2-м семестре.

Используем на декартовом дереве модифицированный HeapSort из предыдущего пункта.
Есть две оценки на скорость работы этого чуда.

Теорема 5.6.1. Если в массив можно разбить на k возрастающих подпоследовательностей, в куче кандидатов никогда не будет более k элементов.
Следствие 5.6.2. Время работы $\mathcal{O}(n \log k)$.

Теорема 5.6.3. Обозначим k_i – количество кандидатов на i-м шаге, тогда $\sum(k_i{-}1) \leqslant I$, где I – количество инверсий.
Следствие 5.6.4. Сортировка работает за $\mathcal{O}(n \log(1{+}\lceil \frac{I}{n} \rceil))$.

Теорема 5.6.5. Блок – отрезок подряд стоящих элементов, которые в сортированном порядке стоят вместе и в таком же порядке. Если наш массив можно представить в виде конкатенации b блоков, то время сортировки $\mathcal{O}(n + b \log b)$.

5.7. (*) Timsort

Сортировка, предложенная в 2002-м для python и с тех пор вытеснившая quick-sort в других языках (java, rust, java-script, swift). В основе лежит merge-sort $\Rightarrow$ сортировка стабильна. Сам merge-sort нас не устраивает по времени на почти отсортированных массивах и по константе памяти.

По времени: самое важное, разбить исходный массив на уже отсортированные $(\nearrow, \searrow)$ отрезки одним проходом, пусть таких отрезков k $\Rightarrow$ нерекурсивно смерджим за $\mathcal{O}(n \log k)$. Ещё оптимизация: когда мерджим два куска, возможно, изменений нужно чуть-чуть, пример: $\{1, 2, 3, 4, 5, 7\} + \{6\}$. Поэтому будем искать позицию вставки k восходящим бинпоиском за $\mathcal{O}(\log k)$, начиная с границы 7 (чтобы при малых k делалось 1 лишнее сравнение).

По памяти: есть

5.8. (*) Ссылки

[AdaptiveHeapSort]. Levcopoulos and Petersson'92. Там же доказаны все теоремы. [TimSort]. wiki [TimSort]. pdf (описание, обоснование времени)

```
PROCEDURE Adaptive Heapsort (X: sequence).
    Construct the Cartesian tree 𝒞(X)
    Insert the root of 𝒞(X) in a heap
    for i := 1 to n do
        Perform ExtractMax on the heap
        if the extracted element has any children in 𝒞(X) then
            Retrieve the children from 𝒞(X)
            Insert the children in the heap
        endif
    endfor
end
```

5.9. (*) 3D Mo

(та же задача, что для Mo, только теперь массив может меняться)

В Offline даны массив длины n и q запросов двух типов:
- `get(l`$_i$`,r`$_i$`)` – запрос на отрезке
- `a[k`$_i$`] = x`$_i$ – поменять значение одного элемента массива.

Пусть мы, зная `get(l,r)` в a, умеем за $\mathcal{O}(1)$ находить `get(l±1,r±1)` в a и `get(l,r)` в $a' = a$ с одним изменённым элементом (например, запрос – число различных чисел на отрезке, а мы храним частоты чисел `unordered_map<int, int> count;`).

Решение в лоб работает $\mathcal{O}(nq)$. Мы с вами решим быстрее.

Пусть i-й запрос имеет тип `get`. Можно рассматривать его над исходным массивом, но от трёх параметров: `get(l,r,i)` – сперва применить первые i изменений, затем сделать `get`. Заметим, что мы за $\mathcal{O}(1)$ пересчитывать `get`, если на 1 поменять l или r или i.

• Алгоритм

Зафиксируем константы x и y. Отсортируем запросы, сравнивая их по $\langle \lfloor \frac{i}{x} \rfloor, \lfloor \frac{l}{y} \rfloor, r \rangle$.
Между запросами будем переходить за $\Delta i + \Delta l + \Delta r$.

• Время работы

$\sum \Delta i = qx + 2q$ (внутри блоков по i + между блоками)
$\sum \Delta l = qy + 2n\frac{q}{x}$ (внутри блоков по l + между блоками по l * число блоков по i)
$\sum \Delta r = 2n\frac{q}{x}\frac{n}{y}$ (движемся по возрастания * на число блоков)
$T = \Theta(\max(\frac{n^2q}{xy}, q(x+y))$, возьмём x, y: $\frac{n^2q}{xy} = q(x+y) \Leftrightarrow n^2 = xy(x+y) \Rightarrow x = y = \Theta(n^{2/3})$, $\boxed{T = qn^{2/3}}$

• Оптимизации

В два раза можно ускорить (убрать все константы 2 из времени работы), чередуя порядки внутри внешних и внутренних блоков – сперва по возрастанию, затем по убыванию и т.д.

5.9.1. (*) Применяем для mex

Def 5.9.1. $mex(A) = \min(\mathbb{N} \cup \{0\} \setminus A)$.

Пример $mex(\{1,2,3\}) = 0$, $mex(\{0,1,1,4\}) = 2$.
Задача – mex на отрезке меняющегося массива в offline.
Идея выше позволяет решить за $\mathcal{O}(qn^{2/3} \log n)$: кроме `unordered_map<int,int> count` нужно ещё поддерживать кучу (`set`) $\{x$: `count[`x`]` $= 0\}$.

Уберём лишний log. Для этого заметим, что к куче у нас будет $qn^{2/3}$ запросов вида `add/del` и лишь `q` запросов `getMin`. Сейчас мы и то, и то обрабатываем за log, получаем $qn^{2/3} \cdot \log + q \cdot \log$.
Используем вместо кучи «корневую» (см.ниже), чтобы получить `add/del` за $\mathcal{O}(1)$, `getMin` за $\mathcal{O}(n^{1/2})$, получим $qn^{2/3} \cdot 1 + q \cdot n^{1/2} = \Theta(qn^{2/3})$.

• Корневая для минимума

Хотим хранить диапазон целых чисел $[0, C)$.
Разобьём диапазон на $\sqrt{C}$ кусков. В каждом хотим хранить количество чисел.
`add/del` числа x – обновление счётчиков `count[`x`]` и `sum_in_block[`$\lfloor \frac{x}{\sqrt{C}} \rfloor$`]`.
`get` – найти перебором блок i: `sum_in_block[`i`]` > 0 и внутри блока x: `count[`x`]` > 0.

Лекция #6: Сортировки (продолжение)

6-я пара, 2024/25

6.1. Quick Sort

• Глубина

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

На практике и в дз мы показали, что при любом детерминированом выборе x или даже как медианы элементов любых трёх фиксированных элементов, $\exists$ тест, на котором время работы сортировки $\Theta(n^2)$. 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.

6.1.1. Оценка времени работы

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

• Доказательство простое

Мы рекурсивно делимся на две подзадачи размера k и $n-k$. За $k \leqslant \frac{n}{2}$ обозначаем меньшую из двух. Худший случай: $k = 1$, лучший случай $k = \frac{n}{2}$. С вероятностью $\frac{1}{2}$ мы попадём x-ом во вторую или третью четверть сортированной версии массива $\Rightarrow$ получим $k \in [\frac{n}{4}, \frac{n}{2}]$. Огрубим до более плохого случая: разбились на $\frac{n}{4}$ и $\frac{3n}{4}$. Иначе $k < \frac{n}{4}$, огрубим до более плохого случая: 0 и n.
Итого время работы $T(n) \leqslant \frac{1}{2}(T(\frac{1}{4}n) + T(\frac{3}{4}n) + n) + \frac{1}{2}(T(n) + n) \Rightarrow T(n) \leqslant T(\frac{1}{4}n) + T(\frac{3}{4}n) + 2n$.
Как и в доказательстве Мастер-Теоремы, сумма подзадач на каждом уровне рекурсии $\leqslant n$, глубина рекурсии $\leqslant \log_{4/3} n \Rightarrow \mathcal{O}(n \log n)$.

• Доказательство #1.

Теорема 6.1.1. $T(n) \leqslant Cn \ln n$, где $C = 2 + \varepsilon$

Доказательство. Докажем по индукции.
Сначала распишем $T(n)$, как среднее арифметическое по всем выборам x.

$$T(n) = n + \frac{1}{n} \sum_{i=0..n-1} (T(i) + T(n-i-1)) = n + \frac{2}{n} \sum_{i=0}^{n-1} T(i)$$

Воспользуемся индукционным предположением (индукция же!)

$$\frac{2}{n} \sum_{i=0}^{n-1} T(i) \leqslant \frac{2}{n} \sum_{i=0}^{n-1} (Ci \ln i) = \frac{2C}{n} \sum_{i=0}^{n-1} (i \ln i)$$

Осталось оценить противную сумму. Проще всего это сделать, перейдя к интегралу.

$$\sum_{i=0}^{n-1} (i \ln i) \leqslant \int_1^n i \ln i \, \mathrm{d}i$$

Такой интеграл берётся по частям. Мы просто угадаем ответ $(\frac{1}{2}x^2 \ln x - \frac{1}{4}x^2)' = x \ln x$.

$T(n) \leqslant n + \frac{2C}{n}((\frac{1}{2}n^2 \ln n - \frac{1}{4}n^2) - (0 - \frac{1}{4})) = n + Cn \ln n - \frac{2C}{4}n + \frac{2C}{4n} = Cn \ln n + n(1 - \frac{C}{2}) + o(1) = F$

Какое же C выбрать? Мы хотим $F \leqslant Cn \ln n$, берём $C > 2$, профит. ■

• **Доказательство #2.**

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 $\leqslant 1$ раз.

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

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

Lm 6.1.2. $Pr[\text{сравнения}(i,j)] = \frac{2}{j-i+1}$ 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 $\sum_{i<j} \frac{2}{j-i+1} = 2\sum_i \sum_{j>i} \frac{1}{j-i+1} \leqslant 2(n \ln n + \Theta(n))$ ■

6.1.2. Introsort'97

На основе Quick Sort можно сделать быструю сортировку, работающую в худшем за $\mathcal{O}(n \log n)$.

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.

6.2. Порядковые статистики

Задача поиска k-й порядковой статистики формулируется своим простейшим решением

```
1 int statistic(a, k)
2     sort(a);
3     return a[k];
```

6.2.1. Одноветочный QuickSort

Вспомним реализацию Quick Sort Code 5.5.5
Quick Sort = выбрать x + `Partition` + 2 рекурсивных вызова Quick Sort.
Будем делать только 1 рекурсивный вызов:

Код 6.2.1. Порядковая статистика

```
1 int Statistic(int l, int r, int *a, int k): // [l, r]
2     if (r <= l) return -1; // так может случиться, только если исходно !(0<=k<=n)
3     int i, j, x = a[random[l,r]];
4     Partition(l, r, x, i, j);
5     if (j < k && k < i) return x;
6     return k <= j ? Statistic(l,j,a,k) : Statistic(i,r,a,k);
```

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

Теорема 6.2.2. Время работы Code 6.2.1 $= \Theta(n)$

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 6.2.3. Мы могли бы повторить доказательство Thm 6.1.1, тогда нам нужно было бы оценить сумму $\sum T(\max(i, n-i-1))$. Это технически сложнее, зато дало бы константу 4.

6.2.2. 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 рекурсивным вызовом себя, это и есть x.

Утверждение 6.2.4. 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:

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

6.2.3. 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.

6.3. Integer sorting

За счёт чего получается целые числа сортировать быстрее чем произвольные объекты?
$\forall k$ операция деления нацело на k: $x \to \lfloor \frac{x}{k} \rfloor$ сохраняет порядок.
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.

6.3.1. 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!
```

Сортировка выше сортирует пары по `a[i]`. Сортировать по `b[i]` аналогично.
Важно то, что сортировка **стабильна**, из этого следует наш следующий алгоритм:

6.3.2. Radix sort

Задача: сортируем 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:** L раз вызвали сортировку подсчётом $\Rightarrow$ сортируем строки $\Theta(L(n + k))$.

Задача: сортируем целые числа из $[0, m)$.

$\forall$ число из $[0, m)$ – строка длины $\log_k m$ над алфавитом $[0, k)$ (цифры в системе счисления k) $\Rightarrow$ умеем сортировать числа из $[0, m)$ за $\Theta((n + k)\lceil \log_k m \rceil)$. При $k = n$ получаем время $n\lceil \log_n m \rceil$.

• **Выбор системы счисления:**
$\texttt{Time} = \Theta((n + k)\lceil \log_k m \rceil) = \Theta(\max(n, k)\frac{\log m}{\log k})$.
При $k \leqslant n$ это $\Theta(n\frac{\log m}{\log k})$, min достигается при $k = n$.
При $k \geqslant n$ это $\Theta(k\frac{\log m}{\log k})$, min достигается при $k = n$.

• **Использование на практике:**
Вы же помните, что деление — операция не быстрая? Значит, выгодно брать $k = 2^{\text{что-то}}$.
Иногда выгодно взять k чуть больше, чтобы $\lceil \log_k m \rceil$ стал на 1 меньше.
Иногда выгодно взять k чуть меньше, чтобы лучше кешировалось.

6.3.3. Bucket sort

Главная идея заключается в том, чтобы числа от `min` до `max` разбить на n бакетов (пакетов, карманов, корзин). Числовая прямая бьётся на n отрезков равной длины, i-й отрезок:

$$\left[\texttt{min} + \tfrac{i}{n}(\texttt{max}-\texttt{min}+1), \texttt{min} + \tfrac{i+1}{n}(\texttt{max}-\texttt{min}+1)\right)$$

Каждое число x_j попадает в отрезок номер $i_j = \lfloor \frac{x_j - \texttt{min}}{\texttt{max}-\texttt{min}+1} n \rfloor$. Бакеты уже упорядочены: все числа в 0-м меньше всех чисел в 1-м и т.д. Осталось упорядочить числа внутри бакетов. Это можно сделать или вызовом Insertion Sort (алгоритм `B.I.`), чтобы минимизировать константу, или рекурсивным вызовом Bucket Sort (алгоритм `B.B.`)

Код 6.3.1. Bucket Sort

```
1  void BB(vector<int> &a) // результат будет записан в a
2      if (a.empty()) return;
3      int n = a.size, min = min_element(a), max = max_element(a);
4      if (min == max) return; // уже отсортирован
5      vector<int> b[n];
6      for (int x : a)
7          int i = (int64_t)n * (x - min) / (max - min + 1); // номер бакета
8          b[i].push_back(x);
9      a.clear();
10     for (int i = 0; i < n; i++)
11         BB(b[i]); // отсортировали каждый бакет рекурсивным вызовом
12         for (int x : b[i]) a.push_back(x); // сложили результат в массив a
```

Lm 6.3.2. $\texttt{max} - \texttt{min} \leqslant n \Rightarrow$ и `BB`, и `BI` работают за $\Theta(n)$

Доказательство. В каждом рекурсивном вызове $\texttt{max} - \texttt{min} \leqslant 1$ ■

Lm 6.3.3. `BB` работает за $\mathcal{O}(n\lceil\log(\texttt{max}-\texttt{min})\rceil)$

Доказательство. Ветвление происходит при $n \geqslant 2 \Rightarrow$ длина диапазона сокращается как минимум в два раза $\Rightarrow$ глубина рекурсии не более log. На каждом уровне рекурсии суммарно не более n элементов. ■

Замечание 6.3.4. На самом деле ещё быстрее, так как уменьшение не в 2 раза, а в n раз.

Lm 6.3.5. На массиве, сгенерированном равномерным распределением, время $\texttt{BI} = \Theta(n)$

Доказательство. Время работы `BI`: $T(n) = \frac{1}{R} \sum\limits_{random} \left(\sum_i k_i^2\right)$, где k_i — размер i-го бакета.

Заметим, что $\sum_i k_i^2$ — число пар элементов, у которых совпал номер бакета.

$$\frac{1}{R}\sum_{random}\left(\sum_i k_i^2\right) = \frac{1}{R}\sum_{random}\left(\sum_{j_1=1}^{n}\sum_{j_2=1}^{n}[i_{j_1} == i_{j_2}]\right) = \sum_{j_1=1}^{n}\sum_{j_2=1}^{n}\left(\frac{1}{R}\sum_{random}[i_{j_1} == i_{j_2}]\right) = \sum_{j_1=1}^{n}\sum_{j_2=1}^{n}\texttt{Pr}[i_{j_1} == i_{j_2}]$$

Осталось посчитать вероятность, при $j_1 = j_2$ получаем 1, при $j_1 \neq j_2$ получаем $\frac{1}{n}$ из равномерности распределения $T(n) = n \cdot 1 + n(n-1) \cdot \frac{1}{n} = 2n - 1 = \Theta(n)$. Получили точное среднее время работы `BI` на случайных данных. ■

6.4. Van Embde Boas'75 trees

Куча над целыми числами из $[0, C)$, умеющая всё, что подобает уметь куче, за $\mathcal{O}(\log\log C)$.
При описании кучи есть четыре принципиально разных случая:

1. Мы храним пустое множество
2. Мы храним ровно одно число
3. $C \leqslant 2$
4. Мы храним хотя бы два числа, $C > 2$.

Первые три вы разберёте самостоятельно, здесь же детально описан **только 4-й случай**.
Пусть $2^{2^{k-1}} < C \leqslant 2^{2^k}$, округлим C вверх до 2^{2^k} ($\log\log C$ увеличился не более чем на 1).
Основная идея — промежуток $[0, C)$ разбить на $\sqrt{C}$ кусков длины $\sqrt{C}$. Также, как и в BucketSort, i-й кусок содержит числа из $[i\sqrt{C}, (i{+}1)\sqrt{C})$. Заметим, $\sqrt{C} = \sqrt{2^{2^k}} = 2^{2^{k-1}}$.
Теперь опишем кучу уровня $k \geqslant 1$, `Heap<k>`, хранящую числа из $[0, 2^{2^k})$.

```
struct Heap<k> {
    int min, size; // отдельно храним минимальный элемент и размер
    Heap<k-1>* notEmpty; // номера непустых кусков
    unoredered_map<int, Heap<k-1>*> parts; // собственно куски
};
```

• Как добавить новый элемент?

Номер куска по числу x: `index(x) = (x >> 2^(k-1));`

```
void Heap<k>::add(int x): // size ⩾ 2, k ⩾ 2, разбираем только интересный случай
    int i = index(x);
    if parts[i] is empty
        notEmpty->add(i); // появился новый непустой кусок, рекурсивный вызов
        parts[i] = {x}; // O(1)
    else
        parts[i]->add(x); // рекурсивный вызов
    size++, min = parts[notEmpty->min]->min; // пересчитать минимум, O(1)
```

Время работы равна глубине рекурсии $= \mathcal{O}(k) = \mathcal{O}(\log\log C)$.

• Как удалить элемент?

```
void Heap<k>::del(int x): // size ⩾ 2, k ⩾ 2, разбираем только интересный случай
    int i = index(x);
    if parts[i]->size == 1
        notEmpty->del(i); // кусок стал пустым, делаем рекурсивный вызов
        parts[i] = empty heap
    else
        parts[i]->del(i)
    min = parts[notEmpty->min]->min; // пересчитать минимум, O(1)
```

Время работы и анализ такие же, как при добавлении. Получается, мы можем удалить не только минимум, а произвольный элемент по значению за $\mathcal{O}(\log\log C)$.

Жаль, но на практике из-за большой константы хеш-таблицы преимущества мы не получим.

6.5. (∗) Inplace merge за $\mathcal{O}(n)$

Мы уже умеем делать inplace `rotate`, что позволяет нам менять нам местами соседние блоки.

• Inplace stable merge за $\mathcal{O}(|a| + |b|^2)$.

Два указателя с конца. Ищем линейным поиском, куда в a вставить последний элемент b.

```
1 // merge: массив x[0..an)[an..an+bn)
2 for (k=an; bn > 0; bn--, k=i) // −1 элемент b, − все пройденные элементы a
3     for (i = k; i > 0 && x[i-1] > x[k+bn-1]; i--)
4         ;
5     rotate(x + i, x + k, x + k+bn); // swap(a[i..|a|), b) → x[0..i) x[k..k+bn) x[i..k)
```

Время работы – сумма длин кусков в `rotate`. Это ровно $|a| + |b|^2$.
Если $|b| = \mathcal{O}(\sqrt{n})$, задача решена за линию.

• Inplace stable merge с внешним буфером.

Нам достаточно буфера длины $\min(|a|, |b|)$. Элементы буфера никуда не потеряются.
Если $|b| < |a|$, мы можем поменять a, b местами (с сохранением стабильности). Пусть $|buf| = |a|$.

```
1 // merge: массив x[0..an)[an..an+bn), буффер buf[0..an)
2 for (int i = 0; i < k; i++)
3     swap(buf[i], x[i]); // именно swap!
4 for (int i = 0, j = 0, p = 0; p < an + bn; p++)
5     swap(x[p], i == an || (j < bn && x[an+j] < buf[i]) ? x[j++] : buf[i++]);
```

Элементы, лежавшие изначально в buf, в конце оказались там также, но перемешались.

• Inplace merge без внешнего буфера.

Наш `merge(a, b)` не будет стабильным.

0. Пусть $k = \sqrt{|a| + |b|}$. Разобьём a и b на куски по k элементов.
Если у a остался хвост, сделаем `swap(tail(a),b)` с помощью `rotate`.

Теперь у нас есть $m \leqslant k$ отсортированных кусков по k элементов и хвост, в котором от 0 до $2k-1$ элементов. Если в хвосте меньше k элементов, добавим в него последний кусок длины k, теперь длина хвоста от k до $2k-1$. Будем использовать хвост, как буфер z.

1. Отсортируем m кусков *сортировкой выбором* за $m^2 + m \cdot k$, сравнивая куски по первому элементу. Время работы: m^2 сравнений и m операций `swap` кусков.

2. Вызываем `merge`$(1, 2, z)$ `merge`$(2, 3, z)$ `merge`$(3, 4, z)\ldots$ для всех $m-1$ пар соседних кусков.

3. Сортируем элементы z за квадрат.

4. Первые $k \cdot m$ элементов и z отсортированы. Осталось смерджить их за $\mathcal{O}(km + |z|^2)$.

Нестабильность появляется в сортировке выбором и перемешивании элементов буфера z.

6.6. (*) Kirkpatrick'84 sort

Научимся сортировать n целых чисел из промежутка $[0, C)$ за $\mathcal{O}(n \log\log C)$.
Пусть $2^{2^{k-1}} < C \leqslant 2^{2^k}$, округлим вверх до 2^{2^k} ($\log\log C$ увеличился не более чем на 1).
Если числа достаточно короткие, отсортируем их подсчётом, иначе каждое 2^k-битное число x_i представим в виде двух 2^{k-1}-битных половин: $x_i = \langle a_i, b_i \rangle$.
Отсортируем отдельно a_i и b_i рекурсивными вызовами.

```
vector<int> Sort(int k, vector<int> &x) {
    int n = x.size()
    if (n == 1 || n >= 2^{2^k}) return CountSort(x) // за O(n)
    vector<int> a(n), b(n), as, result;
    unordered_map<int, vector<int>> BS; // хеш-таблица
    for (int i = 0; i < n; i++) {
        a[i] = старшие 2^{k-1} бит x[i];
        b[i] = младшие 2^{k-1} бит x[i];
        BS[a[i]].push_back(b[i]); // для каждого a[i] храним все парные с ним b[i]
    }
    for (auto &p : A) as.push_back(p.first); // храним все a-шки, каждую 1 раз
    as = Sort(k - 1, as); // отсортировали все a[i]
    for (int a : as) {
        vector<int> &bs = BS[a]; // теперь нужно отсортировать вектор bs
        int i = max_element(bs.begin(), bs.end()) - bs.begin(), max_b = bs[i];
        swap(bs[i], bs.back()), bs.pop_back(); // удалили максимальный элемент
        bs = Sort(k - 1, bs); // отсортировали всё кроме максимума
        for (int b : bs) result.push_back(<a, b>); // выписали результат без максимума
        result.push_back(<a, max_b>); // отдельно добавили максимальный элемент
    }
    return result;
}
```

Оценим время работы. $T(k, n) = n + \sum_i T(k-1, m_i)$. m_i – размеры подзадач, рекурсивных вызовов. Вспомним, что мы из каждого списка bs выкинули 1 элемент, максимум $\Rightarrow$

$$\sum m_i = |as| + \sum_a (|bs_a| - 1) = \sum_a |bs_a| = n$$

Глубина рекурсии не более k, на каждом уровне рекурсии суммарный размер всех подзадач не более $n \Rightarrow$ суммарное время работы $\mathcal{O}(nk) = \mathcal{O}(n \log\log C)$.

Жаль, но на практике из-за большой константы хеш-таблицы преимущества мы не получим.

Лекция #7: Кучи

7-я пара, 2024/25

7.1. Нижняя оценка на построение бинарной кучи

Мы уже умеем давать нижние оценки на число сравнений во многих алгоритмах. Везде это делалось по одной и той же схеме, например, для сортировки «нам нужно различать $n!$ перестановок, поэтому нужно сделать хотя бы $\log(n!) = \Theta(n \log n)$ сравнений».

В случае построения бинарной кучи от перестановки, ситуация сложнее. Есть несколько возможных корректных ответов. Обозначим за $H(n)$ количество перестановок, являющихся корректной бинарной кучей. Процедура построения бинарной кучи переводит любую из $n!$ перестановок в какую-то из $H(n)$ перестановок, являющихся кучей.

Lm 7.1.1. $H(n) = \frac{n!}{\prod_i size_i}$, где $size_i$ – размер поддерева i-й вершины кучи

Доказательство. По индукции. $l + r = n-1$.

$$H(n) = \binom{l+r}{l} H(l)H(r) = \frac{(l+r)!}{l!r!} \frac{l!}{\prod_{i\in L} size_i} \cdot \frac{r!}{\prod_{i\in R} size_i} = \frac{(n-1)!}{\prod_{i\in L\cup R} size_i} = \frac{n!}{\prod_i size_i}$$

В последнем переходе мы добавили в числитель n, а в знаменатель $size_{root} = n$.

■

Теорема 7.1.2. Любой корректный алгоритм построения бинарной кучи делает в худшем случае не менее $1.364n$ сравнений

Доказательство. Пусть алгоритм делает k сравнений, тогда он разбивает $n!$ перестановок на 2^k классов. Класс – те перестановки, на которых наш алгоритм одинаково сработает. Заметим, что алгоритм делающий одно и то же с разными перестановками, на выходе даст разные перестановки. Если x_i – количество элементов в i-м классе, корректный алгоритм переведёт эти x_i перестановок в x_i различных бинарных куч. Поэтому

$$x_i \leqslant H(n)$$

Из $\sum_{i=1..2^k} x_i = n!$ имеем $\max_{i=1..2^k} x_i \geqslant \frac{n!}{2^k}$. Итого:

$$H(n) \geqslant \max x_i \geqslant \frac{n!}{k} \Rightarrow \frac{n!}{\prod_i size_i} \geqslant \frac{n!}{2^k} \Rightarrow 2^k \geqslant \prod size_i \Rightarrow k \geqslant \sum \log size_i$$

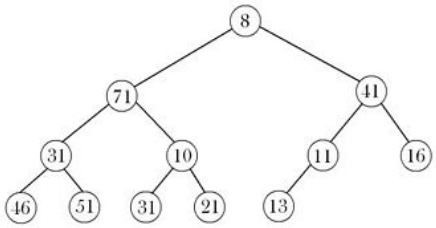
Рассмотрим случай полного бинарного дерева $n = 2^k - 1 \Rightarrow$
$\sum \log size_i = (\log 3)\frac{n+1}{4} + (\log 7)\frac{n+1}{8} + (\log 15)\frac{n+1}{16} + \cdots = (n+1)(\frac{\log 3}{4} + \frac{\log 7}{8} + \frac{\log 15}{16} + \dots)$.
При $n \to +\infty$ величина $\frac{\sum \log size_i}{n+1}$ имеет предел, вычисление первых 20 слагаемых даёт $1.36442\dots$ и ошибку в 5-м знаке после запятой. ■

7.2. Min-Max Heap (Atkison'86)

Min-Max куча – Inplace структура данных, которая строится на исходном массиве и умеет делать `Min`, `Max` за $\mathcal{O}(1)$, а также `Add` и `ExtractMin` за $\mathcal{O}(\log n)$.

Заметим, что мы могли бы просто завести две кучи, одну на минимум, вторую на максимум, а между ними хранить обратные ссылки. Минусы этого решения – в два раза больше памяти, примерно в два раза больше константа времени.

Дети в Min-Max куче такие же, как в бинарной $i \to 2i, 2i+1$. Инвариант: на каждом нечётном уровне хранится минимум в поддереве, на каждом чётном максимум в поддереве. В корне $h[1]$ хранится минимум. Максимум считается как $\max(h[2], h[3])$. Операции `Add` и `ExtractMin` также, как в бинарной куче выражаются через `SiftUp`, `SiftDown`.



• SiftUp

Предположим, что вершина v, которую нам нужно поднять, находится на уровне минимумов (противоположный случай аналогичен). Тогда $\frac{v}{2}$, отец v, находится на уровне максимумов. Возможны следующие ситуации:

1. Значение у v небольше, чем у отца, тогда делаем обычный SiftUp с шагом $i \to \frac{i}{4}$.
2. Иначе меняем местами v и $\frac{v}{2}$, и из $\frac{v}{2}$ делаем обычный SiftUp с шагом $i \to \frac{i}{4}$.

Время работы, очевидно, $\mathcal{O}(\log n)$. Алгоритм сделает не более чем $\frac{\log n}{2} + 1$ сравнение, то есть, примерно в два раза быстрее `SiftUp` от обычной бинарной кучи.

• Корректность SiftUp.

Если нет конфликта с отцом, то вся цепочка от i с шагом два $(i, \frac{i}{4}, \frac{i}{16}, \dots)$ не имеет конфликтов с цепочкой с шагом два от отца. После `swap` внутри `SiftUp` конфликт не появится. Если же с отцом был конфликт, то после `swap(`v`,`$\frac{v}{2}$`)` у $\frac{v}{2}$ и его отца, $\frac{v}{4}$, конфликта нет. ■

• SiftDown

Предположим, что вершина v, которую нам нужно спустить, находится на уровне минимумов (противоположный случай аналогичен). Тогда дети v находятся на уровне максимумов. Возможны следующие ситуации:

1. У v меньше 4 внуков. Обработает случай руками за $\mathcal{O}(1)$.
2. Среди внуков v есть те, что меньше v. Тогда найдём наименьшего внука v и поменяем его местами с v. Осталось проверить, если на новом месте v конфликтует со своим отцом, поменять их местами. Продолжаем SiftDown из места, где первоначально был наименьший внук v.
3. У v все внуки неменьше. Тогда ничего исправлять не нужно.

На каждой итерации выполняется 5 сравнений – за 4 выберем минимум из 5 элементов, ещё за 1 решим возможный конфликт с отцом. После этого глубина уменьшается на 2. Итого $\frac{5}{2}\log_2 n + \mathcal{O}(1)$ сравнений. Что чуть больше, чем у обычной бинарной кучи ($2\log_2 n$ сравнений).

MinMax кучу можно построить за линейное время inplace аналогично двоичной куче.

7.3. Leftist Heap (Clark'72)

Пусть в корневом дереве каждая вершина имеет степень 0, 1 или 2. Введём для каждой вершины

Def 7.3.1. *$d(v)$ – расстояние вниз от v до ближайшего отсутствия вершины.*

Заметим, что $d(\texttt{NULL}) = 0$, $d(v) = \min(d(v.left), d(v.right)) + 1$

Lm 7.3.2. $d(v) \leqslant log_2(size + 1)$

Доказательство. Заметим, что полное бинарное дерево высоты $d(v)-1$ является нашим поддеревом $\Rightarrow size \geqslant 2^{d(v)} - 1 \Leftrightarrow size+1 \geqslant 2^{d(v)}$ ■

Def 7.3.3. *Левацкая куча (leftist heap) – структура данных в виде бинарного дерева, в котором в каждой вершине один элемент. Для этого дерева выполняются условие кучи и условие leftist: $\forall v \quad d(v.left) \geqslant d(v.right)$*

Следствие 7.3.4. В левацкой куче $\log_2 n \geqslant d(v) = d(v.right) + 1$

Главное преимущество левацких куч над предыдущими – возможностью быстрого слияния (`Merge`). Через `Merge` выражаются `Add` и `ExtractMin` (слияние осиротевших детей).

• Merge

Идея. Есть две кучи `a` и `b`. Минимальный из `a->x`, `b->x` будет новым корнем.
Пусть это `a->x`, тогда сделаем `a->r = Merge(a->r, b)` (рекурсия). Конец `=)`

Для удобства реализации `EMPTY` – пустое дерево, у которого `l = r = EMPTY`, `x = +`∞, `d = 0`.

```
1 Node* Merge(Node* a, Node* b):
2     if (!a || !b) return a ? a : b; // если есть пустая, Merge не нужен
3     if (a->x > b->x) swap(a, b); // теперь a - общий корень
4     a->r = Merge(a->r, b);
5     if (a->r->d > a->l->d) // если нарушен инвариант leftist
6         swap(a->r, a->l); // исправили инвариант leftist
7     a->d = a->r->d + 1; // обновили d
8     return a;
```

Время работы: на каждом шаге рекурсии величина `a->d` + `b->d` уменьшается $\Rightarrow$ глубина рекурсии $\leqslant$ `a->d` + `b->d` $\leqslant 2\log_2 n$.

7.4. Skew Heap (Tarjan'86)

Уберём условие $d(v.left) \geqslant d(v.right)$. В функции `Merge` уберём 5 и 7 строки.
То есть, в `Merge` мы теперь не храним d, а просто всегда делаем `swap` детей.

Полученная куча называется «скошенной» (skew heap). В худшем случае один `Merge` теперь может работать $\Theta(n)$, но мы докажем амортизированную сложность $\mathcal{O}(\log_2 n)$. Скошенная куча выгодно отличается короткой реализацией и константой времени работы.

• Доказательство времени работы

Def 7.4.1. *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 $size(v) \leqslant \frac{1}{2} size(p)$

Def 7.4.2. *Ребро parent → v называется правым, если v – правый сын parent.*

Заметим, что из вершины может быть не более 1 тяжёлого ребра вниз.

Lm 7.4.3. На любом вертикальном пути не более $\log_2 n$ лёгких рёбер.

Доказательство. При спуске по лёгкому ребру размер поддерева меняется хотя бы в 2 раза. ■

Как теперь оценить время работы `Merge`? Нужно чем-то самортизировать количество тяжёлых рёбер. Введём потенциал φ = «количество правых тяжёлых рёбер».

Теорема 7.4.4. Время работы `Merge` в среднем $\mathcal{O}(\log n)$

Доказательство. Разделим время работы i-й операции на две части – количество лёгких и тяжёлых рёбер на пройденном в i-й операции пути.

$$t_i = \texttt{Л}_i + \texttt{Т}_i \leqslant log_2 n + \texttt{Т}_i$$

Теперь распишем изменения потенциала φ. Самое важное: когда мы прошли по тяжёлому ребру, после `swap` оно станет лёгким, потенциал уменьшится! А вот когда мы прошли по лёгкому ребру, потенциал мог увеличиться... но лёгких же мало.

$$\Delta\varphi \leqslant \texttt{Л}_i - \texttt{Т}_i \leqslant \log_2 n - \texttt{Т}_i \Rightarrow a_i = t_i + \Delta\varphi \leqslant 2\log n$$

Осталось заметить, что $0 \leqslant \varphi \leqslant n-1$, поэтому среднее время работы $\mathcal{O}(\log n)$. ■

7.5. Quake Heap (потрясная куча)

• **Что у нас уже есть?**

◦ Обычная бинарная куча не умеет Merge, к ней можно прикрутить Merge через Add за $\log^2 n$.
◦ Сегодня изучили Leftist и Skew, которые умеют всё за $\log n$.

• **Наша цель.**

Получить кучу, которая умеет `Add`, `Merge`, `DecreaseKey`, `Min` за $\mathcal{O}(1)$ в худшем и `ExtractMin` за амортизированный $\mathcal{O}(\log n)$. Лучше не получится, т.к. нельзя сортировать быстрее $n \log n$.

Раньше для таких целей использовали Фибоначчиеву или Тонкую (Thin) кучи. Мы изучим более современную и простую quake-heap. Эта куча даёт в теории все нужные $\mathcal{O}(1)$, неплоха на практике, но не является самой быстрой или простой в реализации, главное её достоинство среди подобных — её просто понять.

`[QuakeHeap'2009]`. Полное описание quake-heap от автора (Timothy Chan).
`[Benchmarks'2014]`. Тарьян и ко. замеряют, какие кучи когда быстрее.

• **План.** Нам нужно несколько идей. Пройдём идеи, соединим их в quake-heap.

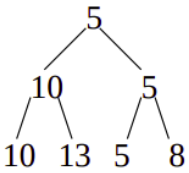
7.5.1. Списко-куча

Список тоже можно использовать как кучу!

Будем хранить элементы в односвязном списке, поддерживаем указатель на текущий минимум. `Heap = {head,tail,min}`. ∀ элемента `x` будем поддерживать обратную ссылку: `x` → `node*`. Операции `Merge`, `Add`, `DecreaseKey`, `Min` работают за $\mathcal{O}(1)$. Операции `ExtractMin` нужно найти новый минимум, для этого она пробежится по всем списку за $\Theta(n)$.

7.5.2. Турнирное дерево

Пусть у нас есть массив `{10,13,5,8}`. Устроим над элементами массива турнир по олимпийской системе, в каждом туре выигрывает минимум. Получим дерево, как на картинке, в корне будет минимум, в листьях элементы исходного массива.



Более опытные из вас уже видели такое под названием «дерево отрезков».

Само по себе турнирное дерево — уже куча.
Можно поменять значение в листе, и за $\mathcal{O}(\log n)$ пересчитать значения на пути до корня.
Чтобы извлечь минимум, заменим значение в соответствующем листе на $+\infty$. Чтобы по корню понять, из какого листа пришло значение, *в вершинах храним не значения, а ссылки на листья.*

Как хранить дерево? Ссылочная структура. У каждой вершины есть *три ссылки: дети, отец.*

7.5.3. Список турнирных деревьев

Если есть два турнирных дерева на массивах длины 2^k и у каждого мы знаем ссылку на корень, за $\mathcal{O}(1)$ их можно соединить в одно дерево: создадим новую вершину, положим туда минимум корней. Далее все деревья имеют размер ровно 2^k, k будем называть *рангом*, нужно по дереву быстро понимать ранг, для этого храним ранг в корне.

Собственно структура: список корней турнирных деревьев.

ExtractMin за $\mathcal{O}(\log n)$. Обозначим длину списка корней за R. Пусть минимум оказался корнем дерева ранга k. После удаления минимума (весь путь от листа до корня), дерево распадётся на k более мелких деревьев. Время работы $R+k$. $k \leqslant \log n$, а вот R может быть большим. Как самортизировать? Уменьшим R до «$\leqslant \log n$». Тогда для потенциала $\varphi = R$ получается

$$a_i = t_i + \Delta\varphi \leqslant (R{+}k) - (R{-}\log n) \leqslant 2\log n$$

Чтобы корней было не больше $\log n$, потребуем «не больше одного корня каждого ранга».
Если видим два дерева одинакового ранга k, их можно объединить в одно дерево ранга $k{+}1$.

```
1 vector<Node*> m(log n, 0); // для каждого ранга k храним или 0, или дерево ранга k
2 for root ∈ ListOfRoots: // просматриваем все имеющиеся корни
3     k = root->rank;
4     while m[k] != 0: // уже есть дерево такого же ранга?
5         root = join(root, m[k]), m[k] = 0, k++; // соединим в k+1!
6     m[k] = root;
```

$R = |\texttt{ListOfRoots}|$. В строке 5 уменьшается общее число корней $\Rightarrow$
строка 5 выполнится не более $R{-}1$ раз $\Rightarrow$ время работы алгоритма $\Theta(R)$.

```
1 ListOfRoots = {} // очистим
2 for root ∈ m:
3     if (root != 0) ListOfRoots.push_back(root) // собрали всё, что лежит в m в список
```

В новом списке $\leqslant \log n$ корней $\Rightarrow \Delta\varphi \leqslant \log n - R \Rightarrow$ амортизированное время работы $\mathcal{O}(\log n)$.

Остальные операции. `Merge` за $\mathcal{O}(1)$ — объединить два списка, `Add` за $\mathcal{O}(1)$ добавить в конец списка, `Min` за $\mathcal{O}(1)$ не забывать везде обновлять указатель на минимум.

DecreaseKey. Эта операция может *только уменьшить* ключ. Важно, что только уменьшить, а в корне минимум. Сейчас понятно, как делать за $\mathcal{O}(\log n)$: поднимемся от листа до корня.

7.5.4. DecreaseKey за $\mathcal{O}(1)$, quake!

Чтобы сделать за $\mathcal{O}(1)$, будем для каждого листа `x` поддерживать ссылку `up[x]` на его самое верхнее вхождение в турнирное дерево (до куда он дошёл в турнире). Промужеточные вершины турнирного дерева хранят ссылку на лист $\Rightarrow$ само значение нужно менять только в листе, $\mathcal{O}(1)$. Если теперь `up[x]` имеет конфликт с отцом `up[x]->p`, решим конфликт радикально за $\mathcal{O}(1)$: отрежем `up[x]` от его отца и добавим в общий список корней.

Сейчас `DecreaseKey` работает за $\mathcal{O}(1)$, а оценка времени `ExtractMin` могла испортиться.

Поймём, как. Количество деревьев после `ExtractMin` = количеству различных рангов $\leqslant$ максимальной высота дерева. Раньше мы знали $size = 2^{heigth} \Rightarrow height \leqslant \log n$, теперь `DecreaseKey` обрезает деревья и нарушает свойства размера.

• Идея.

Выберем $\alpha \in (0.5,1)$, например $\alpha = 0.75$ и будем следить, что $n_{i+1} < \alpha n_i$, где n_i — *суммарное число вершин на уровне i*, здесь $n_0 = n$, листья лежат на уровне 0. Теперь высота $\leqslant \log_{1/\alpha} n$.

• Что делать, когда испортилось?

Испортится в момент, когда `ExtractMin` удаляет корень дерева. Если корень, лежал на уровне i, то n_i уменьшилось, n_{i+1} не изменилось, могло сломаться $n_{i+1} < \alpha n_i$. *Устром землятресение!* Удалим все вершины на слоях $\geqslant i{+}1$ (для этого $\forall i$ поддерживаем список вершин i-го слоя).

• Время работы.

Время работы землятресения $t_{quake} = n_{i+1} + n_{i+2} + n_{i+3} + \dots = \Theta(n_{i+1})$ (на уровнях выше условие не нарушено). Назовём вершину больной, если у неё отрезали детей $\Rightarrow$ степень не 2, а 1. Обобзначим B число больных вершин. Если больных вершин нет, то $n_{i+1} \leqslant \frac{1}{2} n_i$, если все больные, то $n_{i+1} = n_i$. Сейчас $n_{i+1} = \frac{3}{4} n_i \Rightarrow$ «число больных вершин в слое $i{+}1$» $= \frac{1}{2} n_i \Rightarrow$ $\Delta B \leqslant -\frac{1}{2} n_i$ (ещё вершины в верхних слоях) и $t_{quake} = \Theta(|\Delta B|) < 9|\Delta B|$, $\Delta R = \Theta(|\Delta B|) < 9|\Delta B|$.

Возьмём потенциал $\varphi = R + 20 \cdot B$, получим $a_{quake} = t_{quake} + \Delta B \leqslant 9|\Delta B| + 9|\Delta B| + 20\Delta B < 0$.

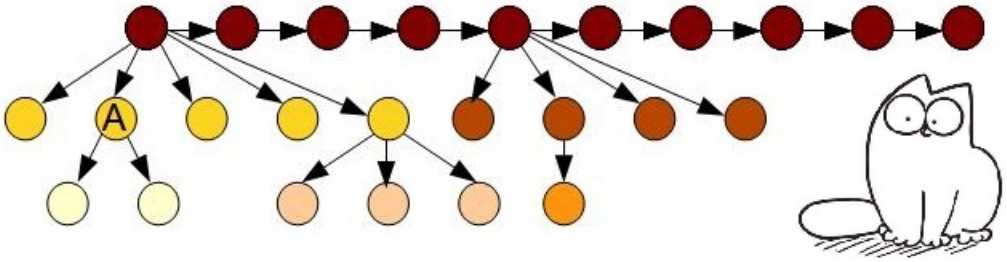
7.6. (*) Pairing Heap

Сейчас мы из списко-кучи получим чудо-структуру, которая умеет амортизированно
`Add`, `Merge` за $\mathcal{O}(1)$, `ExtractMin` за $\mathcal{O}(\log n)$.
`DecreaseKey` за $o(\log n)$, сколько точно, никто не знает.

Пусть ExtractMin проходится по списку длины k. Чтобы с амортизировать $\Theta(k)$, разобьём элементы списка на пары... Формально получится длинная история:

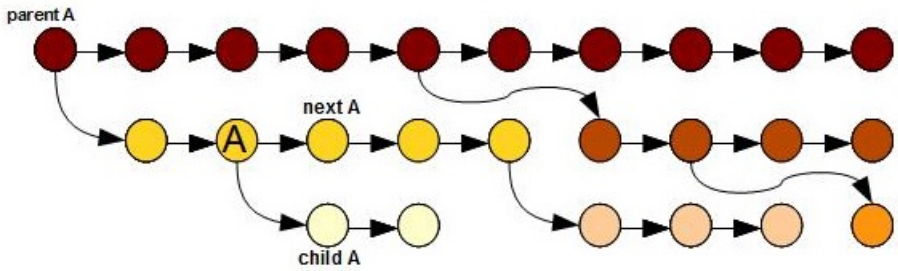
- `PairingHeap = minElement + список детей вида PairingHeap`

Если главный минимум не рисовать, то PairingHeap представляет из себя список корней нескольких деревьев, для каждого дерева выполняется свойство кучи, то есть, значение в любой вершине – минимум в поддереве.



Детей вершины мы храним двусвязным списком. Будем хранить указатель на первого ребенка, для каждого ребенка указатели на соседей в списке и отца. Итого:

```
struct PairingHeap {
    int x;
    PairingHeap *child, *next, *prev, *parent;
};
PairingHeap *root = new PairingHeap {minElement, otherElements, 0, 0, 0};
```



Далее в коде есть куча крайних случаев – списки длины 0, 1, 2. Цель этого конспекта – показать общую идею, на частности не отвлекаемся. Для начала вспомним, что мы умеем со списками:

```
// связали два узла списка
Link(a,b) { a->next = b, b->prev = a; }
// удалить a из списка (a перестанет быть ребёнком a->parent).
ListDelete(a) { Link(a->left, a->right); }
// в список детей a добавить b
ListInsert(a,b) { Link(b, a->child), a->child = b; }
```

Основная операция `Pair` — создать из двух куч одну. Выбирает кучу с меньшим ключом, к ней подвешивает вторую.

```
Pair(a, b):
    if (a->x > b->x) swap(a, b); // корень - меньший из двух
    ListInsert(a, b); // в список детей a добавим b
    return a;
```

`Merge` — объединить два списка. `Add` — один `Merge`. Уже получили `Add` и `Merge` за $\mathcal{O}(1)$ худшем.

```
DecreaseKey(a, newX):
    a->x = newX;
    ListDelete(a); // удалили a из списка её отца
    root = Pair(root, a);
```

Теперь и `DecreaseKey` за $\mathcal{O}(1)$ в худшем. `Delete` $\forall$ не минимума это `DecreaseKey` + `ExtractMin`.

Осталась самая сложная часть – ExtractMin.
Чтобы найти новый минимум нужно пройтись по всем детям. Пусть их k.

```
def ExtractMin:
    root = Pairing(root->child)
def Pairing(a): # пусть наши списки питоно-подобны, "a- список куч
    if |a| = 0 return Empty
    if |a| = 1 return a[0]
    return Pair(Pair(a[0],a[1]), Pairing(a[2:]));
```

Время работы $\Theta(k)$. Результат сей магии – список детей сильно ужался. Точный амортизационный анализ читайте в оригинальной работе Тарьяна. Сейчас важно понять, что поскольку $a_i = t_i + \Delta\varphi$, из-за потенциала поменяется также время работы `Add`, `Merge`, `DecreasyKey`. В итоге получатся заявленные ранее.

7.6.1. История. Ссылки.

[Tutorial]. Красивый и простой функциональный PairingHeap. Код. Картинки.

In practice, pairing heaps are faster than binary heaps and Fibonacci heaps.[2]. Many studies have shown pairing heaps to perform better than Fibonacci heaps in implementations of Dijkstra's algorithm and Prim's minimum spanning tree algorithms.[5]

[Pairing heap]. Fredman, Sleator, Tarjan 1986. Здесь они доказывают оценку $\mathcal{O}(\log n)$ и ссылаются на свою работу, опубликованную годом ранее – Сплэй-Деревья.
[Rank-Pairing Heap]. Tarjan 2011. Скрестили Pairing и кучу Фибоначчи.

Оценки времени работы DecreasyKey в PairingHeap рождались в таком порядке:
- $\Omega(\log\log n)$ – Fredman 1999.
- $\mathcal{O}(2^{2\sqrt{\log\log n}})$ – Pettie 2005.
- Небольшая модификация Pairing Heap, которая даёт $\mathcal{O}(\log\log n)$ – Amr Elmasry 2009.

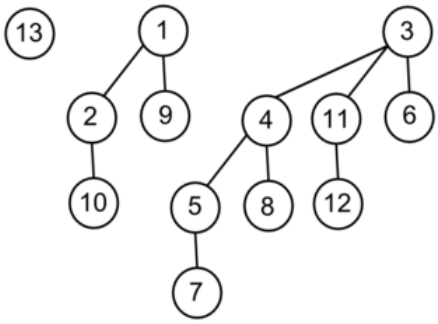
Оценка $\mathcal{O}(\log\log n)$ для оригинальной Pairing Heap на 2019-й год не доказана.

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[11]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[11][12]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[a]	$\Theta(\log n)$	$O(\log n)$[b]
Fibonacci[11][13]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Pairing[3]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$o(\log n)$[a][c]	$\Theta(1)$
Brodal[16][d]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[18]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$

7.7. (*) Биномиальная куча (Vuillemin'78)

7.7.1. Основные понятия

Определим понятие «биномиальное дерево» рекурсивно.



Def 7.7.1. *Биномиальным деревом ранга* 0 *или* T_0 *будем называть одну вершину. Биномиальным деревом ранга* $n+1$ *или* T_{n+1} *будем называть дерево* T_n*, к корню которого подвесили еще одно дерево* T_n *(порядок следования детей не важен). При этом биномиальное дерево должно удовлетворять свойству кучи (значение в вершине не меньше значения в предках).*

Выпишем несколько простых свойств биномиальных деревьев.

Lm 7.7.2. $|T_n| = 2^n$

Доказательство. Индукция по рангу дерева (далее эта фраза и проверка базы индукции будет опускаться). База: для $n = 0$ дерево состоит из одной вершины.
Переход: $|T_{n+1}| = |T_n| + |T_n| = 2^n + 2^n = 2^{n+1}$. ■

Lm 7.7.3. $degRoot(T_n) = n$

Доказательство. $degRoot(T_{n+1}) = degRoot(T_n) + 1 = n + 1$ ■

Lm 7.7.4. Сыновьями T_n являются деревья $T_0, T_1, .., T_{n-1}$.

Доказательство. Сыновья T_{n+1} – все сыновья T_n, т.е. $T_0, .., T_{n-1}$, и новый T_n. ■

Lm 7.7.5. $depth(T_n) = n$

Доказательство. $depth(T_{n+1}) = max(depth(T_n), 1 + depth(T_n)) = 1 + depth(T_n) = 1 + n$ ■

• Как хранить биномиальное дерево?

```
1 struct Node:
2     Node *next, *child, *parent;
3     int x, rank;
```

Здесь `child` – ссылка на первого сына, `next` – ссылка на брата, `x` – полезные данные, которые мы храним. Список сыновей вершины `v`: `v->child`, `v->child->next`, `v->child->next->next`, ...
Теперь определим понятие «биномиальная куча».

Def 7.7.6. *Биномиальная куча – список биномиальных деревьев различного ранга.*

У любого числа n есть единственное представление в двоичной системе счисления $n = \sum_i 2^{k_i}$. В биномиальной куче из n элементов деревья будут иметь размеры как раз 2^{k_i}. Заметим, что в списке не более $\log_2 n$ элементов.

7.7.2. Операции с биномиальной кучей

`Add` и `ExtractMin` выражаются, также как и в левацкой куче, через `Merge`. Чтобы выразить `ExtractMin` заметим, что дети корня любого биномиального дерева по определению являются биномиальной кучей. То есть, после удаления минимума нужно сделать `Merge` от кучи, образованной детьми удалённой вершины и оставшихся деревьев.

`DecreaseKey` – обычный `SiftUp`, работает по лемме Lm 7.7.5 за $\mathcal{O}(\log n)$.

В чём проблема `Merge`, почему просто не соединить два списка? После соединения появятся биномиальные деревья одинакового ранга. К счастью, по определению мы можем превратить их в одно дерево большего ранга

```
Node* join(Node* a, Node* b): // a->rank == b->rank
    if (a->x > b->x) swap(a, b);
    b->next = a->child, a->child = b; // добавили b в список детей a
    return a;
```

Теперь пусть у нас есть список с деревьями возможно одинакового ранга. Отсортируем их по рангу и будем вызывать `join`, пока есть деревья одного ранга.

```
list<Node*> Normalize(list<Node*> &a):
    list<Node*> roots[maxRank+1], result;
    for (Node* v : a) roots[v->rank].push_back(v);
    for (int i = 0; i <= maxRank; i++)
        while (roots[i].size() >= 2):
            Node* a = roots[i].back(); roots[i].pop_back();
            Node* b = roots[i].back(); roots[i].pop_back();
            roots[i+1].push_back(join(a, b));
        if (roots[i].size()): result.push_back(roots[i][0]);
    return result;
```

На каждом шаге цикла `while` уменьшается общее число деревьев $\Rightarrow$ время работы `Normalize` равно $|a| + maxRank = \mathcal{O}(\log n)$. Можно написать чуть умнее, будет $|a|$.

7.7.3. Add и Merge за $\mathcal{O}(1)$

У нас уже полностью описана классическая биномиальная куча. Давайте её ускорять. Уберём условие на «все ранги должны быть различны». То есть, новая куча – список произвольных биномиальных деревьев. Теперь `Add`, `Merge`, `GetMin`, очевидно, работают за $\mathcal{O}(1)$. Но `ExtractMin` теперь работает за длину списка. Вызовем после `ExtractMin` процедуру `Normalize`, которая укоротит список до $\mathcal{O}(\log_2 n)$ корней. Теперь время `ExtractMin` самортизируется потенциалом $\varphi =$ `Roots` (длина списка, количество корней).

Теорема 7.7.7. Среднее время работы `ExtractMin` равно $\mathcal{O}(\log n)$

Доказательство. $t_i =$ `Roots` + `maxRank`, $\Delta\varphi \leqslant \log_2 n -$ `Roots` $\Rightarrow a_i = t_i + \Delta\varphi = \mathcal{O}(\log n)$.
Заметим также, $0 \leqslant \varphi \leqslant n \Rightarrow$ среднее время `ExtractMin` $\mathcal{O}(\log n)$. ■

7.8. (*) Куча Фибоначчи (Fredman,Tarjan'84)

Отличается от всех вышеописанных куч тем, что умеет делать `DecreaseKey` за $\mathcal{O}(1)$. Является апгрейдом биномиальной кучи. Собственно биномиальные кучи практической ценности не име-

ют, они во всём хуже левацкой кучи, а нужны они нам, как составная часть кучи Фибоначчи.

Если `DecreaseKey` будет основан на `SiftUp`, как ни крути, быстрее $\log n$ он работать не будет. Нужна новая идея для `DecreaseKey`, вот она: отрежем вершину со всем её поддеревом и поместим в список корней. Чтобы «отрезать» за $\mathcal{O}(1)$ нужно хранить ссылку на отца и двусвязный список детей (`left`, `right`).

```
struct Node:
    Node *child, *parent, *left, *right;
    int x, degree; // ранг биномиального дерева равен степени
    bool marked; // удаляли ли мы уже сына у этой вершины
```

• Пометки marked

Чтобы деревья большого ранга оставались большого размера, нужно запретить удалять у них много детей. Скажем, что `marked` – флаг, равный единице, если мы уже отрезали сына вершины. Когда мы пытаемся отрезать у v второго сына, отрежем рекурсивно и вершину v тоже.

```
list<Node*> heap; // куча - список корней биномиальных деревьев
void CascadingCut(Node *v):
    Node *p = v->parent;
    if (p == NULL) return; // мы уже корень
    p->degree--; // поддерживаем степень, будем её потом использовать, как ранг
    v->left->right = v->right, v->right->left = v->left; // удалили из списка
    heap.push_back(v), v->marked = 0; // начнём новую жизнь в качестве корня!
    if (p->parent && p->marked++) // если папа - корень, ничего не нужно делать
        CascadingCut(p); // у p только что отрезали второго сына, пора сделать его корнем

void DecreaseKey(int i, int x): // i - номер элемента
    pos[i]->x = x; // pos[i] = обратная ссылка
    CascadingCut(pos[i]);
```

Важно заметить, что когда вершина `v` становится корнем, её `mark` обнуляется, она обретает новую жизнь, как корневая вершина ранга `v->degree`.

Def 7.8.1. *Ранг вершины в Фибоначчиевой куче – её степень на тот момент, когда вершина последний раз была корнем.*

Если мы ни разу не делали `DecreaseKey`, то `rank` = `degree`. В общем случае:

Lm 7.8.2. `v.rank` = `v.degree` + `v.mark`

Заметим, что по коду ранги нам нужны только в `Normalize`, то есть, в тот момент, когда вершина является корнем. В доказательстве важно будет, что у любой вершины ранги детей различны.

Теорема 7.8.3. `DecreseKey` работает в среднем за $\mathcal{O}(1)$

Доказательство. `Marked` – число помеченных вершин. Пусть φ = `Roots` + 2`Marked`.
Амортизированное время операций кроме `DecreaseKey` не поменялось, так как они не меняют `Marked`. Пусть `DecreaseKey` отрезал $k+1$ вершину, тогда Δ`Marked` $\leqslant -k$, Δ`Roots` $\leqslant k+1$, $a_i = t_i + \Delta\varphi = t_i + \Delta$`Roots` $+ 2\Delta$`Marked` $\leqslant (k+1) + (k+1) - 2k = \Theta(1)$. ■

7.8.1. Фибоначчиевы деревья

Чтобы оценка $\mathcal{O}(\log n)$ на `ExtractMin` не испортилась нам нужно показать, что $size(rank)$ – всё ещё экспоненциальная функция.

Def 7.8.4. *Фибоначчиево дерево F_n – биномиальное дерево T_n, с которым произвели рекурсивное обрезание: отрезали не более 1 сына, и рекурсивно запустились от выживших детей.*

Оценим S_n – минимальный размер дерева F_n

Lm 7.8.5. $\forall n \geqslant 2$: $S_n = 1 + S_0 + S_1 + \cdots + S_{n-2}$

Доказательство. Мы хотим минимизировать размер.
Отрезать ли сына? Конечно, да! Какого отрезать? Самого толстого, то есть, F_{n-1}. ■

Заметим, что полученное рекуррентное соотношение верно также и для чисел Фибоначчи:

Lm 7.8.6. $\forall n \geqslant 2$: $Fib_n = 1 + Fib_0 + Fib_1 + \cdots + Fib_{n-2}$

Доказательство. Индукция. Пусть $Fib_{n-1} = Fib_0 + Fib_1 + \cdots + Fib_{n-3}$, тогда
$1+Fib_0+Fib_1+\cdots+Fib_{n-2} = (1+Fib_0+Fib_1+\cdots+Fib_{n-3})+Fib_{n-2} = Fib_{n-1}+Fib_{n-2} = Fib_n$ ■

Lm 7.8.7. $S_n = Fib_n$

Доказательство. База: $Fib_0 = S_0 = 1, Fib_1 = S_1 = 1$. Формулу перехода уже доказали. ■

Получили оценку снизу на размер дерева Фибоначчи ранга n: $S_n \geqslant \frac{1}{\sqrt{5}}\varphi^n$, где $\varphi = \frac{1+\sqrt{5}}{2}$.
И поняли, почему куча называется именно Фибоначчиевой.

7.8.2. Завершение доказательства

Фибоначчиева куча – список деревьев. Эти деревья **не являются Фибоначчиевыми** по нашему определению. Но размер дерева ранга k не меньше S_k.

Покажем, что новые деревья, которые мы получаем по ходу операций `Normalize` и `DecreaseKey` не меньше Фибоначчиевых.
$\forall v$ дети v, отсортированные по рангу, обозначим x_i, $i = 0..k-1$, $rank(x_i) \leqslant rank(x_{i+1})$.
Будем параллельно по индукции доказывать два факта:

1. Размер поддерева ранга k не меньше S_k.
2. $\forall$ корня v $\quad rank(x_i) \geqslant i$.
   То есть, ранг детей поэлементно не меньше рангов детей биномиального дерева.

Про размеры: когда v было корнем, его дети были не меньше детей биномиального дерева того же ранга, до тех пор, пока ранг v не поменяется, у v удалят не более одного сына, поэтому дети v будут не меньше детей фибоначчиевого дерева того же ранга.
Теперь рассмотрим ситуации, когда ранг меняется.

Переход #1: v становится корнем. Детей v на момент, когда v в предыдущий раз было корнем, обозначим x_i. Новые дети x'_i появились удалением из x_i одного или двух детей. $x'_i \geqslant x_i \geqslant i$ ■

Переход #2: `Join` объединяет два дерева ранга k. Раньше у корня i-й ребёнок был ранга хотя бы i для $i = 0..k-1$. Теперь мы добавили ему ребёнка ранга ровно k, отсортируем детей по рангу, теперь $\forall i = 0..k \quad rank(x_i) \geqslant i$. ■

Лекция #8: Рекурсивный перебор

8-я пара, 2024/25

Вспомним задачу с практики «в каком порядке расположить числа a_i, чтобы сумма $\sum a_i b_i$ была максимальна?» Если задача кажется вам слишком простой, можете представить себе более сложную: «то же, но каждое число можно двигать вправо-влево не больше чем на два».

Пусть вы придумали решение. Как надёжно проверить его корректность, если вы уже не первом курсе и под рукой нет тестирующей системы с готовыми тестами?

1. Сгенерить случайный тест.
2. Запустить медленное, зато точно правильное решение.

Откуда взять медленное решение? Рекурсивный перебор всех возможных вариантов. Для задах выше нужен перебор перестановок с него и начнём.

8.1. Перебор перестановок

• next_permutation

Если вы пишете на языке C++ можно воспользоваться `next_permutation`.
Такой вариант отработает корректно даже, если a содержит одинаковые элементы.

```
sort(a.begin(), a.end()); // минимальная перестановка
do { ...
} while (next_permutation(a.begin(), a.end())); // все перестановки массива a
```

Можно также перебирать перестановки a не меняя порядок исходного массива.

```
vector<int> p = {0, 1, ... n-1};
do { // перемешанный массив a: a[p[0]], a[p[1]], a[p[2]],...
} while (next_permutation(p.begin(), p.end())); // все перестановки массива a
```

Перестановок $n!$. Если ещё хотим посчитать для каждой перестановки целевую функцию $\sum a_i b_i$, время работы решения будет $\mathcal{O}(n! \cdot n)$.

• Рекурсивное решение

Перебираем, что поставить на первую позицию...
Для каждого варианта перебираем, что поставить на вторую позицию...

```
void go(int i): // i -- позиция в перестановке
    if (i == n):
        // что-нибудь сделать с нашей перестановкой
        return
    for (int x = 0; x < n; x++)
        if (!used[x]):
            used[x] = 1 // далее нельзя использовать элемент x
            p[i] = x, go(i+1); // поставили x, перебираем дальше
            used[x] = 0 // а в других ветках рекурсии можно
go(0);
```

Рекурсивный вариант более гибкий:
1. По ходу рекурсии можно насчитывать нужные нам суммы.
2. Можно перебирать не все перестановки, а только нужные.

```
void go(int i, int s): // i -- позиция в перестановке
    if (i == n):
        best = max(best, s); // решение сложной из двух версий задачи выше
        return
    for (int x = 0; x < n; x++)
        if (!used[x]):
            if (abs(i - x) > 2) continue; // пропускаем лишние
            used[x] = 1 // далее нельзя использовать элемент x
            p[i] = x, go(i+1, s + p[i]*b[i]); // пересчитали сумму
            used[x] = 0 // а в других ветках рекурсии можно
go(0);
```

Получили код, работающий ровно за количество «перестановок, которые нужно перебрать». Ещё говорят за $\mathcal{O}$(ответа), имея в виду задачу «вывести все перестановки такие что».

8.2. Перебор множеств и запоминание

Пусть у нас есть n предметов, у каждого есть вес w_i и мы хотим выбрать подмножество с суммой весов ровно S. *Решение рекурсивным перебором:* каждый предмет или берём, или не берём.

```
void go(int i, int sum):
    if (sum > S) return // оптимизация!
    if (i == n)
        if (sum == S) // набрали подмножество суммы S
        return
    used[i]=0, go(i + 1, sum) // не берём
    used[i]=1, go(i + 1, sum + w[i]) // берём
```

Количество множеств 2^n, перебор работает за $\mathcal{O}(2^n)$. Если включить оптимизацию и перебирать мн-ва только суммы $\leqslant S$, то за $\mathcal{O}$(ответа). Массив `used` нужен только, если хотим сохранить само множество, а не только проверить «можем ли набрать». В любом случае сумму весов, как и раньше, насчитываем по ходу рекурсии.

• Запоминание

Результат работы рекурсии зависит только от параметров функции. Второй раз вызываемся с теми же пераметрами? Ничего нового мы уже не найдём (для конкретной задачи: если раньше не нашли множества суммы S, то и в этот раз не найдём).

```
set<pair<int,int>> mem; // запоминание
void go(int i, int sum):
    if (sum > S) return // оптимизация!
    if (i == n) return // конец
    if (mem.count({i,S})) return; // были уже в таком состоянии?
    mem.insert({i,S}); // запомним, что теперь «уже были»
    used[i]=0, go(i + 1, sum) // не берём
    used[i]=1, go(i + 1, sum + w[i]) // берём
```

• Время работы перебора с запоминанием

Для каждой комбинации параметров i, sum зайдём в рекурсию не более одного раза. Комбинаций $n{\cdot}S$ (i до n, sum до S). Время работы $\mathcal{O}(nS)$, если S мало, это лучше старого $\mathcal{O}(2^n)$.

• **Ограбление банка (рюкзак)**

Унести предметы суммарного веса $\leqslant W$ максимальной суммарной стоимости.

```
void go(int i, int sw, int scost):
    if (sw > W) return // оптимизация!
    if (i == n)
        best = max(best, scost);
        return
    go(i + 1, sw, scost) // не берём
    go(i + 1, sw + w[i], scost + cost[i]) // берём
go(0, 0, 0); cout << best << endl;
```

Поменяем перебор. Пусть он нам возвращает «какую ещё стоимость можно набрать из оставшихся предметов, если свободного места в рюкзаке W».

```
int go(int i, int W):
    if (W <= 0 || i == n) return 0
    return max(go(i + 1, W), // не берём
        go(i + 1, W - w[i], cost + cost[i])); // берём, уменьшили свободное место
cout << go(0, 0) << endl;
```

В таком виде можно добавить запоминание: `map<pair<int,int>, int>` — для каждой пары $\{i, W\}$, от которой мы уже запускались хранить результат.

```
map<pair<int,int>, int> mem;
int go(int i, int W):
    if (W <= 0 || i == n) return 0
    if (mem.count({i,W})) return mem[{i,W}];
    return mem[{i,W}] = max(go(i + 1, W), // не берём
        go(i + 1, W - w[i], cost + cost[i])); // берём, уменьшили свободное место
cout << go(0, 0) << endl;
```

Время работы теперь $\mathcal{O}(n \cdot W)$ — для каждой пары $\{i, W\}$ считаемся один раз.

• **Технические оптимизации**

Конечно, можно вместо `map` использовать `unordered_map` (но от пары нет хеша, поэтому нужно сперва `pair<int,int>` → `int64_t`), или даже двумерный массив (он же вектор векторов).

8.3. Перебор путей (коммивояжёр)

Задача: мы развозчик грузов, есть один грузовик и n заказов «что куда отвезти», начинаем в точке s, нужно всё развести за минимальное время. С точки зрения графов: найти путь, проходящий по всем выделенным точкам минимальной суммарной длины.

Решение рекурсивным перебором: перебираем, как перестановки, куда поехать сперва, куда поехать потом, куда дальше...

```
int go(int cnt, int v, vector<int> &used): // cnt - сколько заказов уже выполнили
    if (cnt == n): return 0 // посетили всё, что хотели
    int best = INT_MAX; // лучший (минимальный) вариант
    for (int i = 0; i < n; i++) // выбираем, какой заказ обработать следующим
        if (!used[i]):
            used[i] = 1 // теперь доехать от v до i и рекурсивно вызваться
            best = min(best, dist(v,i) + go(cnt+1, i, used));
            used[i] = 0
    return best
```

Всегда можно добавить запоминание. Добавим. `mem: int,vector<int>` → `int`.
Время работы $2^n n \cdot n \cdot$ `map`: всего 2^n различных `used`, n различных $v \Rightarrow$ дойдём до цикла `for` мы $2^n n$ раз, $2^n n$ раз сделаем n итераций цикла.

8.4. Разбиения на слагаемые

Задача: сколько есть разбиений числа N на строго возрастающие слагаемые?
Пример: $6 = 1+2+3, 6 = 1+5, 6 = 2+4, 6 = 6$
Решение перебором #1: перебираем сперва первое слагаемое, затем второе...

```
int go(int n, int x): // последнее слагаемое было x ⇒ наше и следующие >x
    if (n == 0): return 1 // ровно 1 способ разбить 0 на слагаемые
    int ans = 0;
    for (int y = x + 1; y <= n; y++)
        ans += go(n - y, y) // перебрали очередное слагаемое y
    return ans
go(N, 0) // первым слагаемым попробуем все 1..N
```

Решение перебором #2: каждое слагаемое или берём или не берём

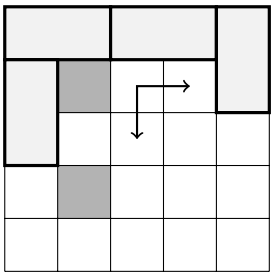
```
int go(int n, int x):
    if (n == 0): return 1 // ровно 1 способ разбить 0 на слагаемые
    return go(n, x - 1) + (n<x ? 0 : go(n-x, x))
```

Сейчас оба решения работают за $\mathcal{O}$(ответа). Разбиений на слагаемые $2^{\Theta(\sqrt{N})}$.

После добавления запоминания превое решение будет работает за $\mathcal{O}(N^3)$, второе за $\mathcal{O}(N^2)$.
Время работы: (сколько раз мы зайдём в `go(n,x)`) $\cdot$ (число рекурсивных вызовов).

8.5. Доминошки и изломанный профиль

Решим задачу про покрытие доминошками: есть доска с дырками размера $w \times h$. Сколько способов замостить её доминошками (фигуры 1×2) так, чтобы каждая не дырка была покрыта ровно один раз? Для начала напишем рекурсивный перебор, который берёт первую непокрытую клетку и пытается её покрыть. Если перебирать клетки сверху вниз, а в одной строке слева направо, то есть всего два способа покрыть текущую клетку.



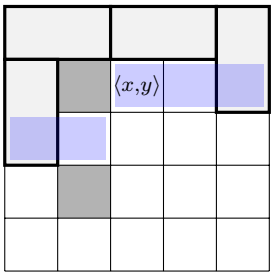
```
int go(int x, int y):
    if (x == w) x = 0, y++; // начали следующую строку
    if (y == h) return 1; // все строки заполнены, 1 способ закончить заполнение
    if (!empty[y][x]) return go(x + 1, y);
    int result = 0;
    if (y + 1 < h && empty[y + 1][x]):
        empty[y + 1][x] = empty[y][x] = 0; // поставили вертикальную доминошку
        result += go(x + 1, y);
        empty[y + 1][x] = empty[y][x] = 1; // убрали за собой
    if (x + 1 < w && empty[y][x+1]): // аналогично для горизонтальной
        empty[y + 1][x] = empty[y][x] = 0;
        result += go(x + 1, y);
        empty[y + 1][x] = empty[y][x] = 1;
    return result;
```

`go(x, y)` вместо того, чтобы каждый раз искать с нуля первую непокрытую клетку помнит «всё, что выше-левее (x, y) мы уже покрыли». Возвращает функция `go` число способов докрасить всё до конца. `empty` – глобальный массив, ячейка пуста $\Leftrightarrow$ там нет ни дырки, ни доминошки. Время работы данной функции не более $2^{\texttt{число доминошек}} \leqslant 2^{wh/2}$. Давайте теперь, как и задаче про клики добавим к нашему перебору запоминание. Что есть состояние перебора? Вся матрица.

Получаем следующий код:

```
vector<vector<bool>> empty;
map<vector<vector<bool>>, int> m; // запоминание
int go(int x, int y):
    if (x == w) x = 0, y++; // начали следующую строку
    if (y == h) return 1; // все строки заполнены, 1 способ закончить заполнение
    if (!empty[y][x]) return go(x + 1, y);
    if (m.count(empty)) return m[empty];
    int result = &m[empty];
    if (y + 1 < h && empty[y + 1][x]):
        empty[y + 1][x] = empty[y][x] = 0; // поставили вертикальную доминошку
        result += go(x + 1, y);
        empty[y + 1][x] = empty[y][x] = 1; // убрали за собой
    if (x + 1 < w && empty[y][x+1]): // аналогично для горизонтальной
        empty[y + 1][x] = empty[y][x] = 0;
        result += go(x + 1, y);
        empty[y + 1][x] = empty[y][x] = 1;
    return result;
```

Теорема 8.5.1. Количество состояний динамики $\mathcal{O}(2^w hw)$.



Доказательство. Когда мы находимся в клетке (x, y). Что мы можем сказать про покрытость остальных? Все клетки выше-левее (x, y) точно `not empty`. А все ниже-правее? Какие-то могли быть задеты уже поставленными доминошками, но не более чем на одну «изломанную строку» снизу от (x, y). Кроме этих w все клетки находятся в исходном состоянии. ■

• Дальнейшая оптимизация

По ходу доказательства теоремы мы также придумали способ запоминать ответ лучше чем `map<vector<vector<bool>>, int> m`: `int m[w][h][2^w]` и обращаться `m[x,y,A]`, где `A` – w бит соответствующей изломаной строки (скошеного профиля).

Лекция #9: Динамическое программирование 1

9-я пара, 2024/25

9.1. Базовые понятия

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

9.1.1. 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?

9.1.2. Динамика назад

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

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

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».

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

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 $\Theta(n)$.

9.1.3. 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 $f[x] = 1 + \min(f[x-1], f[x-7], f[\frac{x}{2}], f[\frac{x}{3}])$.

• **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 $f[x+1], f[x+7], \ldots$

9.1.4. Ленивая динамика

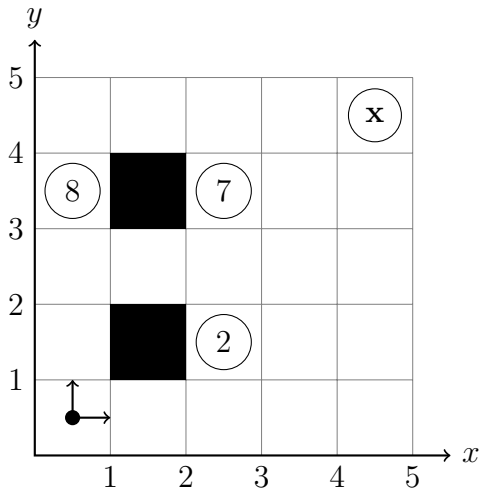
Это рекурсивный способ писать динамику назад, вычисляя значение только для тех состояний, которые действительно нужно посчитать.

```
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 (x - 7 >= 1) relax(r, calc(x - 7)); // стандартная ошибка: написать f[x-7]
    if (x % 2 == 0) relax(r, calc(x / 2));
    if (x % 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;`.

9.2. 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 $(width-1, height-1)$.

9.3. 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] + value[x,y]) // f[x,y] получилось из f[x-1,y]
    x--;
else
    y--;
```

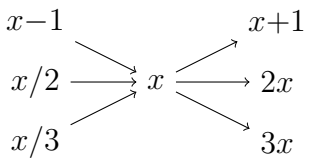
Второму способу нужно меньше памяти, но обычно он требует больше строк кода.

• 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.

9.4. 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 9.4.1. 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 9.4.2. 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.

9.5. 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.

9.6. Рюкзак

9.6.1. 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)

9.6.2. 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:
$$\begin{cases} [i, x] \to [i+1, x] & \text{(не брать)} \\ [i, x] \to [i+1, x+a_i] & \text{(берём в ответ)} \end{cases}$$

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

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

9.6.3. Оптимизируем память

Мы уже знаем, что, если не нужно восстанавливать ответ, то достаточно хранить лишь две строки динамики $is[i], is[i+1]$, в задаче о рюкзаке можно хранить лишь одну строку динамики:

Код 9.6.1. Рюкзак с линейной памятью

```
bool is[S+1] <-- 0;
is[0] = 1;
for (int i = 0; i < n; i++)
    for (int j = S - a[i]; j >= 0; j--) // поменяли направление, важно
        if (is[j])
            is[j + a[i]] = 1;
// Answer = is[S]
```

Путь к пониманию кода состоит из двух шагов.
(а) $is[i, x] = 1 \Rightarrow is[i+1, x] = 1$. Единицы остаются, если мы умели набирать вес i, как подмножество из i предметов, то как подмножество из $i+1$ предмета набрать, конечно, тоже сможем.
(б) После шага j часть массива $is[j..S]$ содержит значения строки $i+1$, а часть массива $is[0..j-1]$ ещё не менялась и содержит значения строки i.

9.6.4. Добавляем bitset

`bitset` — массив бит. Им можно пользоваться, как обычным массивом. С другой стороны с `bitset` можно делать все логические операции `|`, `&`, `^`, `«`, как с целыми числами. Целое число можно рассматривать, как `bitset` из 64 бит, а `bitset` из n бит устроен, как массив $\lceil \frac{n}{64} \rceil$ целых чисел. Для асимптотик введём обозначения `w` от `word_size` (размер машинного слова). Тогда наш код можно реализовать ещё короче и быстрее.

```
bitset<S+1> is;
is[0] = 1;
for (int i = 0; i < n; i++)
    is |= is << w[i]; // выполняется за O(S/w)
```

Заметим, что мы делали ранее ровно указанную операцию над битовыми массивами.
Время работы $\mathcal{O}(\frac{nS}{w})$, то есть, в 64 раза меньше, чем раньше.

9.6.5. Восстановление ответа с линейной памятью

Модифицируем код Code 9.6.1, чтобы была возможность восстанавливать ответ.

```
int last[S+1] <-- -1;
last[0] = 0;
for (int i = 0; i < n; i++)
    for (int j = S - a[i]; j >= 0; j--)
        if (last[j] != -1 && last[j + a[i]] == -1)
            last[j + a[i]] = i;
// Answer = (last[S] == -1 ? NO : YES)
```

И собственно восстановление ответа.

```
for (int w = S; w > 0; w -= a[last[w]])
    print(last[w]); // индекс взятого в ответ предмета
```

Почему это работает?
Заметим, что когда мы присваиваем `last[j+a[i]] = i`, верно, что на пути по обратным ссылкам из `j`: `last[j]`, `last[j-a[last[j]]]`, ... все элементы строго меньше i.

9.7. Квадратичные динамики

Def 9.7.1. *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.

9.8. 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, давайте улучшать.

9.8.1. Храним биты

Можно хранить не $f[i,j]$, а разность соседних $df[i,j] = f[i,j] - f[i,j-1]$, она или 0, или 1.
Храним $\Theta(nm)$ бит $= \Theta(\frac{nm}{w})$ машинных слов. Этого достаточно, чтобы восстановить ответ: мы умеем восстанавливать путь, храня только f, чтобы сделать 1 шаг назад в этом пути, достаточно знать 2 строки `f[]`, восстановим их за $\mathcal{O}(m)$, сделаем шаг.

9.8.2. Алгоритм Хиршберга (по wiki)

Общая идея. Восстановить ответ = восстановить путь из $(0,0)$ в (n,m). Хотим за $\mathcal{O}(nm)$ времени и $\mathcal{O}(m)$ памяти восстановить клетку (row, col) этого пути в строке $row = \frac{n}{2}$ (посередине). После этого сделать 2 рекурсивных вызова для кусков задачи $(0,0) \to (row, col)$ и $(row, col) \to (n,m)$.

Пусть мы ищем НОП (LCS) для последовательностей `a[0..n)` и `b[0..m)`.
Обозначим $n' = \lfloor \frac{n}{2} \rfloor$. Разделим задачу на подзадачи посчитать НОП на подотрезках $[0..n') \times [0..j)$ и $[n'..n) \times [j..m)$. Как выбрать оптимальное j? Для этого насчитаем две квадратные динамики:

$f[i,j]$ — НОП для первых i символов a и первых j символов b и
$g[i,j]$ — НОП для последних i символов a и последних j символов b.

Нас интересуют только последние строки — $f[n']$ и $g[n-n']$, поэтому при вычислении можно хранить лишь две последние строки, $\mathcal{O}(m)$ памяти.

Выберем j: $f[n',j] + g[n-n', m-j] = \max$, сделаем два рекурсивных вызова от $a[0..n') \times b[0..j)$ и $a[n'..n) \times b[j..m)$, которые восстановят нам половинки ответа.

9.8.3. Оценка времени работы Хиршберга

Заметим, что глубина рекурсии равна $\lceil \log_2 n \rceil$, поскольку n делится пополам.

Lm 9.8.1. Память $\Theta(m + \log n)$

Доказательство. Для вычисления f и g мы используем $\Theta(m)$ памяти, для стека рекурсии $\Theta(\log n)$ памяти. ■

Lm 9.8.2. Время работы $\Theta(nm)$

Доказательство. Глубина рекурсии $\mathcal{O}(\log n)$.
Суммарный размер подзадач на i-м уровне рекурсии $= m$. Например, на 2-м уровне это $i + (m-i) = m$. Значит $Time \leqslant nm + \lceil \frac{n}{2} \rceil m + \lceil \frac{n}{4} \rceil m + \ldots \leqslant 4nm$. ■

Подведём итог проделаной работы:

Теорема 9.8.3.
Мы умеем искать НОП с восстановлением ответа за $\Theta(nm)$ времени, $\Theta(m + \log n)$ памяти.

Алгоритм полностью описан в wiki.

9.8.4. (*) Алгоритм Хиршберга (улучшенный)

Пусть мы ищем НОП (LCS) для последовательностей `a[0..n)` и `b[0..m)`.
Пишем обычную динамику $lcs[i,j] = \langle$длина НОП для `a[0..i)`, `b[0..j)`, ссылка назад$\rangle$. Будем хранить только две последние строки динамики. Если раньше ссылку из i-й строки мы хранили или на i-ю, или на $(i-1)$-ю, теперь для восстановления ответа будем для строк $[\frac{n}{2}..n)$ хранить ссылку на то место, где мы были в строке номер $\frac{n}{2}$.
TODO: здесь очень нужна картинка.

```
def relax(i, j, pair, add):
    pair.first += add
    lcs[i,j] = min(lcs[i,j], pair)

def solve(n, a, m, b):
    if n <= 2 or m <= 2:
        return naive_lcs(n, a, m, b) # запустили наивное решение

    # ВАЖНО: обязательно нужно хранить только 2 последние строчки lcs
    # Для простоты чтения кода, эта оптимизация здесь специально не сделана
    lcs[] <-- -INF, lcs[0,0] = 0; # инициализация
    for i = 0..n-1:
        for j = 0..m-1:
            relax(i, j, lcs[i,j-1], 0)
            relax(i, j, lcs[i-1,j], 0)
            if (i > 0 and j > 0 and a[i-1] == b[j-1]):
                relax(i, j, lcs[i-1,j-1], 1)
            if (i == n/2):
                lcs[i,j].second = j # самое важное, сохранили, где мы были в n/2-й строке

    # нашли клетку, через которую точно проходит последовательность-ответ
    i, j = n/2, lcs[n,m].second;
    return solve(i, a, j, b) + solve(n-i, a+i, m-j, b+j)
```

Заметим, что и глубина рекурсия, и ширина, и размеры всех подзадач будут такими же, как в доказательстве Thm 9.8.3 $\Rightarrow$ оценки те же.

Теорема 9.8.4.
Новый алгоритм ищет НОП с восстановлением ответа за $\Theta(nm)$ времени, $\Theta(m + \log n)$ памяти.

9.8.5. Область применение идеи Хиршберга

Данным алгоритмом (иногда достаточно простой версии, иногда нужна вторая) можно восстановить ответ без ухудшения времени работы для огромного класса задач. Например

1. Рюкзак со стоимостями
2. Расстояние Левенштейна
3. «Задача о погрузке кораблей», которую мы обсудим на следующей паре
4. «Задача о серверах», которую мы обсудим на следующей паре

Лекция #9: Формула включения-исключения

2024/25

9.9. (*) Разминочные задачи

Задача #1: количество чисел от 1 до n, которые не делятся ни на одно из простых $p_1, \dots, p_k$.

Идея: возьмём все числа, для каждого вычтем i вычтем кратные p_i, те, что кратны $p_i p_j$ вычли два раза, добавим обратно...
Решение за $\mathcal{O}(2^k)$: ответ $= \sum\limits_{A \in [0,2^k)} (-1)^{|A|} \lfloor \frac{n}{\prod_{i \in A} p_i} \rfloor$.
Разберёмся, почему именно такой знак перед слагаемым. Рассматриваем множество A: $|A| = k$. Индукция по k. База $k = 0$. Переход: мы учли числа, кратные всем p_i: $i \in A$ уже $\sum_{i=0}^{k-1} (-1)^i \binom{k}{i}$ раз. А хотим учесть 0 раз. Добавим к сумме $(-1)^k$, получится как раз 0.

Задача #2: пусть теперь даны произвольные числа $a_1, a_2, \dots, a_k$.
Идея решения не поменяется, чисел кратных и a_1, и a_2 $\lfloor \frac{n}{lcm(a_1,a_2)} \rfloor$, итого $\sum\limits_{A \in [0,2^k)} (-1)^{|A|} \lfloor \frac{n}{lcm(p_i \mid i \in A)} \rfloor$.

Задача #3. Стоим в клетке $(0, 0)$, хотим в (n, n), за ход можно или вправо на 1, или вверх на 1. В клетках $(x_1, y_1) \dots (x_k, y_k)$ ямы. Сколько есть путей, не проходящих по ямам?

Если бы ям не было, ответ был бы $\binom{2n}{n}$.
Если ямы одна, ответ $\binom{2n}{n} - \binom{x+y}{y}\binom{2n-x-y}{n-y}$. Для удобства обозначим $\binom{x+y}{x} = g(x, y)$.
В общем случае обозначим $(x_0, y_0) = (n, n)$ и напишем динамику f_i – сколько способов дойти до i-й ямы и не провалиться в предыдущие. $f_i = g(x_i, y_i) - \sum_j f_j g(x_i - x_j, y_i - y_j))$. Здесь j – яма, в которую мы первой провалились по пути в i-ую. Считаем f_i в порядке возрастания $\langle x_i, y_i \rangle$.

9.10. (*) Задачи с формулой Мёбиуса

Задача #4. Сколько есть множеств натуральных чисел A: $|A| = k \leqslant 100, lcm(A) = n \leqslant 10^{12}$.
Заметим, все $a_i \mid n \Rightarrow$ найдём $D(n)$ – множество делителей n, будет выбирать только $a_i \in D(n)$. Всего множеств сейчас $|D(n)|^k$. Вычтем те, у которых lcm меньше n. Пусть $lcm = \frac{n}{d}$, тогда нужно прибавить к ответу $\mu_d |D(\frac{n}{d})|^k$, где μ_d – коэффициент, который предстоит найти.
$\mu_d = 0 - \sum_{z|d} x_z$, а $\mu_1 = 1$. Получается $\mu = 1, -1, -1, 0, -1, 1, \dots$.
В математике μ называется функцией Мёбиуса [wiki], [itmo] и имеет более простую формулу: $\mu(p_1 p_2 \dots p_k) = (-1)^k$ для простых различных p_i, иначе $0 \Rightarrow$ все $\mu_{d|n}$ считаются за $\mathcal{O}(D(n))$.

Итого: $ans = \sum_{d|n} \mu_d |D(\frac{n}{d})|^k$. Время работы $\mathcal{O}(factorize(n) + D(n) \log k)$.

TODO: to be continued

Лекция #10: Динамическое программирование 2

10-я пара, 2024/25

10.1. bitset

`bitset` – структура для хранения N бит.
Обладает полезными свойствами и массива, и целых чисел. Заметим, что с целыми 64-битными числами мы можем делать логические операции `|`, `&`, `«`, `^` за один такт. То есть, если рассматривать число, как массив из 64 бит, *параллельно за один процессорный такт применять операцию* `OR` *к нескольким ячейкам массива*. `bitset<N>` хранит массив из $\lfloor \frac{N}{64} \rfloor$ целых чисел. Число 64 – константа, описывающая свойство современных процессоров. В дальнейшем будем все алгоритмы оценивать для абстрактной `w-RAM` машины, машины, на которой за 1 такт производятся базовые арифметические операции с w-битовыми регистрами.

w – сокращение от `word size`

```
bitset<N> a, b; // N - константа; в C++, к сожалению, размер bitset-а должен быть константой
x = a[i], a[j] = y; // O(1) - операции с массивом
a = b | (a << 100), a &= b; // O(N/w) - битовые операции
```

10.1.1. Рюкзак

Применим новую идею к задаче о рюкзаке:

```
bitset<S+1> is;
is[0] = 1; // изначально умеет получать пустым множеством суммарный вес 0
for (int i = 0; i < n; i++)
    is |= is << a[i]; // если is[w] было 1, теперь is[w + a[i]] тоже 1
```

10.2. НОП → НВП

На прошлой паре уже решили НОП за квадрат. В общем случае люди не умеют за $\mathcal{O}(n^{2-\varepsilon})$, в некоторых случаях можно быстрее. Пусть мы ищем НОП a и b, и *все элементы* b *различны*.

1. Хеш-таблицей сделаем предподсчёт «где такой элемент в b»: `position[`b_i`] = i`.
2. Сгенерим массив p_i = `position[`a_i`]`.
3. Утверждение: НОП(a,b) = НВП(p). Более того — есть биекция между возрастающими подпоследовательностями p и общими подпоследовательностями a и b.

10.3. НВП за $\mathcal{O}(n \log n)$

Пусть дана последовательность $a_1, a_2, \ldots, a_n$.
С прошлой пары уже умеем искать за $\mathcal{O}(n^2)$ времени. Улучшим.

Код 10.3.1. Алгоритм поиска НВП за $\mathcal{O}(n \log n)$

```
x[] <-- inf
x[0] = -inf, answer = 0; // x[len] - минимально возможный конец посл-ти длины len
for (int i = 0; i < n; i++):
    int j = lower_bound(x, x + n, a[i]) - x;
    x[j] = a[i], answer = max(answer, j);
```

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), некоторые не берём. Состояние этого процесса можно описать (k, len, i) – мы рассмотрели первые k чисел, из них len взяли в ответ, последним взяли номер 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 → решение динамикой» – максимизировать $len[k, i]$. Но можно сделать $i[k, len]$ и минимизировать конец выбранной последовательности $x = a[i]$ $(x[k, len])$. Пойдём вторым путём, поймём, как вычислять $x[k, len]$ быстрее чем $\mathcal{O}(n^2)$.

Lm 10.3.2. $\forall k, len \quad x[k, len] \leqslant x[k, len + 1]$

Доказательство. Из посл-ти длины $len{+}1$, выкинем первый элемент, получим посл-ть длины len с таким же концом. ■

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

Lm 10.3.3. В реализации Code 10.3.1 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 Lm 10.3.2 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 его позицию $xp[j]$:

```
1 x[j] = a[i], xp[j] = i; // a[i] - значения, i - позиция
2 prev[i] = xp[j-1];
```

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

10.4. Задача про погрузку кораблей

Дан склад грузов, массив $a_1, a_2, \ldots, a_n$. Есть бесконечное множество кораблей размера S. При погрузке должно выполняться $\sum a_i \leqslant S$. Грузим корабли по одному, погруженный корабль сразу уплывает. Погрузчик может брать только самый левый/правый груз из массива. *Задача:* минимизировать число кораблей, использованное для перевозки грузов.

Представим себе процесс погрузки: k кораблей уже погружено полностью, на складе остался отрезок грузов $[L, R]$. Состояние такого процесса описывается (k, L, R). В будущем будем использовать круглые скобки для описания состояния процесса и квадратные для состояния динамики. Такое видение процесса даёт нам динамику $k[L, R] \to \min$. Переходы:

$$(k, L, R) \to (k{+}1, L', R'), \text{ при этом } sum[L..L') + sum(R'..R] \leqslant S$$

Состояний n^2, время работы $\mathcal{O}(n^4)$.

10.4.1. Измельчение перехода

Идея оптимизации динамики в таких случаях – «измельчить переход». Сейчас переход – «погрузить корабль с нуля целиком», новый переход будет «погрузить один предмет на текущий

корабль или закончить его погрузку». Итак, пусть w – суммарный размер грузов в текущем корабле. Переходы:

$$\begin{cases} (k,w,L,R) \to (k,w+a_L,L+1,R) & \text{погрузить самый левый, если } w+a_L \leqslant S \\ (k,w,L,R) \to (k,w+a_R,L,R-1) & \text{погрузить самый правый, если } w+a_R \leqslant S \\ (k,w,L,R) \to (k+1,0,L,R) & \text{начать погрузку следующего корабля} \end{cases}$$

Что выбрать за состояние динамики, что за функцию?
$0 \leqslant k,L,R \leqslant n$, $0 \leqslant w \leqslant S$, выбираем $w[k,L,R] \to min$, поскольку w не ограничена через n.
Минимизируем, так как при прочих равных выгодно иметь максимально пустой корабль.
Получили больше состояний, но всего 3 перехода: n^3 состояний, время работы $\mathcal{O}(n^3)$.

10.4.2. Использование пары, как функции динамики

Можно сделать ещё лучше: $\langle k,w\rangle[L,R]$.
Минимизировать сперва k, при равенстве k минимизировать w.
Теперь мы сохраняем не все состояния процесса, а из пар $\langle k_1,w_1\rangle \leqslant \langle k_2,w_2\rangle$ только $\langle k_1,w_1\rangle$.
Действительно, если $k_1=k_2$, то выгодно оставить пару с меньшим w, а если $k_1<k_2$, то можно отправить корабль $\langle k_1,w_1\rangle \to \langle k_1+1,0\rangle \leqslant \langle k_2,w_2\rangle$ $(k_1+1 \leqslant k_2, 0 \leqslant w_2)$. К сожалению, переход $(k,w,L,R) \to (k+1,0,L,R)$ теперь является петлёй.
Динамике же на вход нужен граф без циклов. Поэтому делаем так:

$$(k,w,L,R) \to \begin{cases} (k,w+a_L,L+1,R) & \text{если } w+a_L \leqslant S \\ (k+1,a_L,L+1,R) & \text{если } w+a_L > S \\ \text{аналогичные переходы для } R \to R-1 \end{cases}$$

Итог: $\mathcal{O}(n^2)$ времени, $\mathcal{O}(n^2)$ памяти.
Заметим, что без восстановления ответа можно хранить только две строки, $\mathcal{O}(n)$ памяти, а при восстановлении ответа применим алгоритм Хиршберга, что даёт также $\mathcal{O}(n)$ памяти.

10.5. Рекуррентные соотношения

Def 10.5.1. *Линейное рекуррентное соотношение:*
даны $f_0,f_1,\ldots,f_{k-1}$, $\forall n$ $f_n = f_{n-1}a_1 + f_{n-2}a_2 + \ldots f_{n-k}a_k + b$

Задача: даны $f_0,\ldots,f_{k-1};a_1,\ldots,a_k;b$, найти f_n.
Очевидно решение динамикой за $\mathcal{O}(nk)$.

Научимся решать быстрее сперва для простейшего случая — числа Фибоначчи.

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Напомним, как работает умножение матриц: строка левой умножается скалярно на столбец правой. То есть, первое равенство читается как $(F_n = 1\cdot F_{n-1}+1\cdot F_{n-2}) \wedge (F_{n-1} = 1\cdot F_{n-1}+0\cdot F_{n-2})$

Lm 10.5.2. Возведение в степень можно сделать за $\mathcal{O}(\log n)$

Доказательство. Пусть определена ассоциативная операция $a \circ b$, $a^n = \underbrace{a \circ a \circ \cdots \circ a}_{n}$ тогда:

$$a^{2k} = (a^k)^2 \quad \wedge \quad a^{2k+1} = (a^k)^2 \circ a \quad \wedge \quad a^1 = a$$

Итог: рекурсивная процедура возведения в степень n, делающую не более $2\log n$ операций $\circ$. ■

Следствие 10.5.3.
F_n можно посчитать за $\mathcal{O}(\log n)$ арифметических операций с числами порядка F_n.
F_n можно посчитать по модулю P за $\mathcal{O}(\log n)$ арифметических операций с числами порядка P.

Вернёмся к общему случаю, нужно увидеть возведение матрицы в степень.

Теорема 10.5.4. Существует решение за $\mathcal{O}(k^3 \log n)$

Доказательство.
$$\begin{pmatrix} f_{n+k+1} \\ f_{n+k} \\ f_{n+k-1} \\ \dots \\ f_{n+1} \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & \dots & a_{k-1} & a_k & b \\ 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_{n+k} \\ f_{n+k-1} \\ f_{n+k-2} \\ \dots \\ f_n \\ 1 \end{pmatrix} = A \cdot V$$

В общем случае иногда выгодно $\mathcal{O}(nk)$, иногда $\mathcal{O}(k^3 \log n)$. $\exists$ решение за $\mathcal{O}(k \log k \log n)$ ■

10.5.1. Пути в графе

Def 10.5.5. *Матрица смежности C графа G: C_{ij} – есть ли ребро между (i, j) в G.*

Задача: найти количество путей из s в t длины ровно k.
Решение динамикой за $\mathcal{O}(kn^2)$: $f[k, v]$ – количество путей из s в v длины ровно k.
Пересчёт $f[k, v] = \sum_i f[k-1, i] \cdot C_{i,v} \Rightarrow f_k = C \cdot f_{k-1} = C^k f_0$.
Подсчёт $f[k, t]$ в лоб будет работать за $\mathcal{O}(kn^2)$, с быстрого помощью возведения матрицы в степень за $\mathcal{O}(n^3 \log k)$, так как одно умножение матриц работает за n^3.

10.6. Задача о почтовых отделениях

На прямой расположены города в координатах $x_1, x_2, \dots, x_n$, население городов $w_1, w_2, \dots, w_n$ ($w_i > 0$). Нужно в каких-то k из этих городов $i_1, i_2, \dots i_k$ открыть почтовые отделения, минимизируя при этом суммарное расстояние по всем людям до ближайшего почтового отделения:

$$\sum_j w_j \min_t |x_j - x_{i_t}| \longrightarrow \min$$

Упростим себе жизнь сортировками. Пусть $x_1 < x_2 < \dots < x_n$, $i_1 < i_2 < \dots < i_k$.
Города разобьются на k отрезков — к какому почтовому отделению относится город.
Кстати, границы отрезков будут проходить в $\frac{1}{2}(x_{i_t} + x_{i_{t+1}})$.

Оптимальный центр для отрезка $[l, r)$ будем обозначать $m[l, r]$.
Суммарную стоимость отрезка обозначим $cost(l,r) = \sum_{i \in [l,r]} w_i |x_i - x_{m[l,r]}|$.
Задача в выборе границ отрезков $p_1{=}1, p_2, p_3, \dots, p_{k+1}{=}n{+}1 : \sum_{i=1..k} cost(p_i, p_{i+1}{-}1) \to \min$.

Задача решается динамикой:
$f[k, n]$ — стоимость разбиения первых n городов на k отрезков.
$p[k, n]$ — оптимальная граница k-го отрезка, в него входят точки $(p_{k,n}, n]$.
Тогда $ans = f[K,N]$, $f[0, 0] = 0$, $\boxed{f[k, n] = f[k{-}1, p_{k,n}] + cost(p_{k,n}{+}1, n)}$.
Как найти оптимальное $p[k, n]$? Перебрать циклом for все n вариантов.

Время работы $\mathcal{O}(kn^2) + n^2 \cdot$ `calcCost`, где $cost(l, r)$ в лоб считается за $\mathcal{O}(n^2)$. Можно быстрее.

Lm 10.6.1. Зная $m[l,r]$, можно посчитать $cost(l,r)$ за $\mathcal{O}(1)$

Доказательство. Обозначим $m = m[l,r]$.

$$\sum_{i=l..r} w_i|x_i - x_m| = \sum_{i=l..m} w_i(x_m - x_i) + \sum_{i=m..r} w_i(x_i - x_m) = x_m(\sum_{i=l..m} w_i - \sum_{i=m..r} w_i) - \sum_{i=l..m} x_i w_i + \sum_{i=m..r} x_i w_i$$

Получили четыре суммы на отрезках, каждая считается через префиксные суммы. ■

Lm 10.6.2. $m[l,r] = \min i\colon \sum_{j\in[l,i]} w_j \geqslant \sum_{j\in(i,r]} w_j$

Доказательство. Задача с практики. ■

Lm 10.6.3. $\forall l$ все $m[l,r]$ вычисляется за $\mathcal{O}(n)$ двумя указателями: $r\uparrow \Rightarrow m[l,r]\nearrow$.

```
int i = l;
for (int r = l; r <= n; r++):
    while (sum(l, i) < sum(i+1, r)) // сумма на отрезке за O(1)
        i++;
    m[l,r] = i;
```

Теорема 10.6.4. Задача про почтовые отделения решена динамикой за $\mathcal{O}(n^2k)$

Доказательство. Сперва за $\mathcal{O}(n^2)$ предподсчитали все $m[l,r]$, затем за $\mathcal{O}(n^2k)$ $f[k,n]$ ■

Замечание 10.6.5. Далее мы научимся считать динамику $f[k,n]$ быстрее.
Если мы получаем время $o(n^2)$, то придётся улучшать уже подсчёт $m[l,r]$: для пары $\langle l,r\rangle$ оптимальное $m[l,r]$ находится за $\mathcal{O}(\log n)$ бинпоиском.

10.6.1. Оптимизация Кнута

Предположим, что $p_{k-1,n} \leqslant p_{k,n} \leqslant p_{k,n+1}$,
тогда в решении можно перебирать $p_{k,n}$ не от 0 до $n-1$, а между $p_{k-1,n}$ и $p_{k,n+1}$. *Итого:*

```
// обнулили f[] и p[]
for (int k = 2; k <= K; k++) // порядок перебора состояний следует из неравенств
    for (int n = N; n >= k; n--): // порядок перебора состояний следует из неравенств
        f[k, n] = ∞;
        for (int i = p[k-1, n]; i <= p[k, n+1]; i++): // уже посчитаны
            int tmp = f[k-1, i] + cost(i+1, n);
            if (tmp < f[k, n])
                f[k, n] = tmp, p[k, n] = i;
```

Теорема 10.6.6. Описанное решение работает $\mathcal{O}(n^2)$

Доказательство. $Time = \sum_{n,k}(p_{k,n+1} - p_{k-1,n})$. Заметим, что все кроме $n+k$ слагаемые присутствуют в сумме и с +, и с -, значит, сократятся $\Rightarrow Time \leqslant (n+k)n = \mathcal{O}(n^2)$. ■

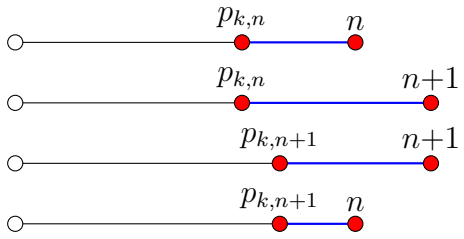
10.6.2. (*) Доказательства неравенств

Теперь докажем корректность $p_{k-1,n} \leqslant p_{k,n} \leqslant p_{k,n+1}$. Заметим сразу, что не для всех возможных оптимальных ответов это неравенство верно, но если уже фиксированы любые $p_{k-1,n}$ и $p_{k,n+1}$, дающие оптимальный ответ, то $\exists p_{k,n}$, удовлетворяющее обоим неравенствам и дающее оптимальный ответ $f_{k,n}$.

Lm 10.6.7. $\forall k,n\ \forall$оптимальных $[k,n]$ и $[k,n+1]$ разбиений верно, что $p_{k,n+1} \geqslant p_{k,n}$

Доказательство. Рассмотрим оптимальные решения задач $[k, n]$ и $[k, n{+}1]$.
Центр последнего отрезка, $m[p_{k,n}, n]$, обозначим $q_{k,n}$.
Центр последнего отрезка, $m[p_{k,n+1}, n]$, обозначим $q_{k,n+1}$.
Доказывать будем от противного, то есть, $p_{k,n+1} < p_{k,n}$.
Рассмотрим четыре разбиения:

1. Оптимальное для $[k, n]$
2. Копия 1, в последний отрезок добавили точку $n{+}1$
3. Оптимальное для $[k, n{+}1]$
4. Копия 3, из последнего отрезка убрали точку $n{+}1$



Функции f от разбиений обозначим f_1, f_2, f_3, f_4. В последнем отрезке каждого из разбиений нужно выбрать центр. В 1-м это $q_{k,n}$, в 3-м $q_{k,n+1}$, а в 2 и 4 можем взять любой.

Чтобы прийти к противоречию достаточно получить $f_4 < f_1$ (противоречит с тем, что f_1 оптимально). От противного мы уже знаем, что $f_3 < f_2$.

• **Случай #1:** $q_{k,n} \geqslant q_{k,n+1}$. Пусть 2-е разбиение имеет центр $q_{k,n}$, а 4-е центр $q_{k,n+1}$.
При $f_1 \to f_2$ функция увеличилась на $w_{n+1}(x_{n+1} - x_{q_{k,n}})$.
При $f_3 \to f_4$ функция уменьшилась на $w_{n+1}(x_{n+1} - x_{q_{k,n+1}})$.
$q_{k,n} \geqslant q_{k,n+1} \Rightarrow$ уменьшилась хотя бы на столько же, на сколько увеличилась $f_3 < f_2 \Rightarrow f_4 < f_1$.

• **Случай #2:** $q_{k,n} < q_{k,n+1}$. Пусть 2-е разбиение имеет центр $q_{k,n+1}$, а 4-е центр $q_{k,n}$.
Напомним что $p_{k,n+1} < p_{k,n} \leqslant q_{k,n} < q_{k,n+1}$.
Обозначим $f_{23} = f_2 - f_3$, $f_{14} = f_1 - f_4$. Докажем $0 < f_{23} \leqslant f_{14}$. Первое уже есть, нужно второе.
Доказываем $f_{14} - f_{23} \geqslant 0$. При вычитании сократится всё кроме последнего отрезка.
В последнем отрезке сократится всё кроме расстояний для точек $(p_{k,n+1}, p_{k,n}]$.
Множество точек в f_{14} и f_{23} одно и то же, а расстояния считаются до $q_{k,n}$ и $q_{k,n+1}$ соответственно.
Заметим, что расстояние до $q_{k,n+1}$ больше и берётся со знаком минус $\Rightarrow$
$f_{14} - f_{23} \geqslant 0 \Rightarrow f_{14} > 0 \Rightarrow f_1 > f_4 \Rightarrow f_1$ не оптимален. Противоречие. ■

Lm 10.6.8. Для задач, где $p_{k,n} \leqslant p_{k,n+1}$ верно и $p_{k-1,n} \leqslant p_{k,n}$.

Доказательство. От противного. Посмотрим на цепочки для восстановления ответа
$n \to p_{k,n}{-}1 \to \ldots$ и $n \to p_{k-1,n}{-}1 \to \ldots$ Если в $p_{k,n} < p_{k-1,n}$, то вторая цепочка изначально обгоняет первую, первая в итоге должна догнать и перегнать (в первой больше отрезков, а конец у них общий. ■

10.6.3. Оптимизация методом «разделяй и властвуй»

> **Divide et impera!** — *принцип государственной власти, согласно которому, лучший метод управления разнородным государством — разжигание и использование вражды между его частями. Одним из первых применил на практике политику Гай Юлий Цезарь.*

Пусть мы уже насчитали строку динамики $f[k{-}1]$, то есть, знаем $f[k{-}1, x]$ для всех x.

Найдём строку p_k методом разделяй и властвуй, через неё за $\mathcal{O}(n)$ посчитаем $f[k]$.

Уже доказали Lm 10.6.7 $\forall n,k\ p_{k,n} \leqslant p_{k,n+1}$.
Напишем функцию, которая считает $p_{k,n}$ для всех $n \in [l,r]$, зная, что $L \leqslant p_{k,n} \leqslant R$.

```
1 void go(int l, int r, int L, int R):
2     if (l > r) return; // пустой отрезок
3     int m = (l + r) / 2;
4     Найдём p[k,m] в лоб за O(R − L + 1)
5     go(l, m - 1, L, p[k,m]);
6     go(m + 1, r, p[k,m], R);
7 go(1, n, 1, n); // посчитать всю строку p[k] для x = 1..n
```

Теорема 10.6.9. Время работы пересчёта $f[k-1] \to p_k$ всего лишь $\mathcal{O}(n \log n)$

Доказательство. Глубина рекурсии не более $\log n$ (длина отрезка уменьшается в 2 раза).
На каждом уровне рекурсии выполняются подзадачи $(L_1,R_1),(L_2,R_2),\dots(L_k,R_k)$.
$L_1 = 1, R_i = L_{i+1}, R_k = n \Rightarrow$ время работы $= \sum(R_i - L_i + 1) = (n-1) + k \leqslant 2n = \mathcal{O}(n)$ ■

10.6.4. Стресс тестирование

Это нужно для проверки инварианта, который позволяет применять разделяй и властвуй.

Как вы видели из доказательства Lm 10.6.7, оно весьма нетривиально.
Обозначим за P_1 вероятность того, что вам попадётся динамика, где можно применить оптимизацию Кнута (Section 10.6.1) или «Разделяй и Властвуй» (Section 10.6.3). Обозначим за P_2 вероятность того, что задача попадётся, и вы сможете доказать требуемое $\forall n,k\ p_{k,n} \leqslant p_{k,n+1}$.
Пусть $P_1 > 0$.

$$\frac{P_2}{P_1} \approx 0$$

Но зачем доказывать, если можно написать версии с/без оптимизации и пострессить?

Есть несколько способов.

#1. Взять решение без Кнута, с Кнутом и на случайных тестах сравнить ответ.
#2. На случайных тестах сравнить массивы `f[n,k]` целиком.
#3. Поставить в решении без Кнута внутри `assert`$(\forall n,k\ p_{k,n} \leqslant p_{k,n+1})$.

Второе более сильное свидетельство. Гораздо более сильное. `(2)` всегда лучше `(1)`.
С третьим нужно быть аккуратным. Во-первых, аккуратным на крайних значениях n, k. Во-вторых, это сработает только если $\forall$ корректного ответа для `p[n,k]` выполняются такие неравенства (в смысле, есть случаи, когда `assert` падает, а решение всё равно работает).

Доказательство с точки зрения программирования: пострессить на правильном наборе тестов.
Тут важно понимать, что крайние случаи в основном проявляются на **маленьких** тестах.
Например, в задаче про почтовые отделения это $2 \leqslant k \leqslant n \leqslant 8$ и веса, координаты в $[1,8]$.
На $500 \leqslant k \leqslant 1000$ таких тестов рандомом не найти.

Лекция #11: Динамическое программирование 3

11-я пара, 2024/25

11.1. Динамика по подотрезкам

Рассмотрим динамику по подотрезкам (практика, задача 3). 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: $r \uparrow\, l \downarrow$
Можно рассмотреть ациклический граф на вершинах $[l, r]$ и рёбрах $[l, m], [m+1, r] \to [l, r]$, соответствующий нашему решению. Задача, которую мы решаем, уже не выражается в терминах «поиск пути» или «количество путей» в графе. Верным остаётся лишь то, что ответ в вершине $[l, r]$ можно выразить через ответы для подзадач, то есть, для соседей вершины $[l, r]$.

Занимательный факт: все задачи динамического программирования кроме «динамики по подотрезкам», которые мы решали ранее или решим сегодня, таки выражаются, как «поиск пути» или «количество путей».

11.2. Комбинаторика

Дан комбинаторный объект, например, перестановка. Научимся по объекту получать его номер в лексикографическом порядке и наоборот по номеру объекту восстанавливать объект.

Например, есть 6 перестановок из трёх элементов, лексикографически их можно упорядочить, как вектора: $(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)$.
Занумеруем с нуля, тогда номер перестановки $(3, 1, 2)$ – 4.
Цель – научиться быстро переходить между номером объекта и самим объектом.

Зачем это может быть нужно?

(a) Закодировать перестановку минимальным числом бит, сохранить на диск
(b) Использовать номер, как индекс массива, если состоянием динамики является перестановка

• Объект → номер (перестановки)

На примере перестановок изучим общий алгоритм.
Нужно посчитать количество перестановок лексикографически меньших p.
$\{a_1, a_2, \ldots, a_n\} < \{p_1, p_2, \ldots, p_n\} \Leftrightarrow \exists k$: $a_1 = p_1, a_2 = p_2, \ldots a_{k-1} = p_{k-1}, a_k < p_k$. Переберём k и a_k, после этого к ответу нужно прибавить количество способов закончить префикс $a_1, \ldots, a_k$, для перестановок это $(n - k)!$. Здесь a_k может быть любым числом, не равным $a_1, \ldots a_{k-1}$.

```
1 vector<bool> was(n+1);
2 int result = 0;
3 for (int k = 1; k <= n; k++) {
4     was[p[k]] = true;
5     for (int j = 1; j < p[k]; j++)
6         if (!was[j]) // выбираем a_1 = p_1, ..., a_{k-1} = p_{k-1}, a_k = j
7             result += factorial[n-k];
8 }
```

Время работы $\mathcal{O}(n^2)$. Для перестановок существует алгоритм за $\mathcal{O}(n \log n)$, наша цель – лишь продемонстрировать общую схему.

• Объект → номер (правильные скобочные последовательности)

Пусть у нас один тип скобок и `'('` < `')'`. Ищем количество ПСП, меньших s, $|s| = n$. Переберём k, если $s_k =$ `')'`, попытаемся заменить, на `'('`. Сколько способов закончить, зависит только от k и текущего баланса b, разности числа открывающих и закрывающих скобок в $s[1..k) +$ `'('`. Предподсчитаем динамику $dp[k, b]$ – число способов закончить. База: $dp[n, 0] = 1$.

$$dp[k, b] = dp[k + 1, b + 1] + (b = 0 \ ? \ 0 : dp[k + 1, b - 1])$$

```
int balance = 0, result = 0;
for (int k = 0; k < n; k++) {
    if (s[k] == ')')
        result += dp[k, balance+1]
    balance += (s[k] == '(' ? 1 : -1);
}
```

Время работы алгоритма $\mathcal{O}(n)$, время предподсчёта – $\mathcal{O}(n^2)$.

Замечание 11.2.1. Математики поговаривают, что `dp[n,k]` – разность цэшек. Зная это, можно придумать алгоритм за $\mathcal{O}(n)$ без предподсчёта.

• Номер → объект (перестановки)

Чтобы получить по лексикографическом номеру k сам объект, нужно строить его слева направо. Переберём, что стоит на первом месте в перестановке. Пусть стоит d, сколько способов продолжить перестановку? $(n-1)!$ способов. Если $k \leqslant (n-1)!$, то нужно ставить минимальную цифру, иначе уменьшить k на $(n-1)!$ и попробовать поставить следующую...

```
vector<bool> was(n+1);
for (int i = 1; i <= n; i++)
    for (int d = 1;; d++) {
        if (was[d]) // нельзя поставить d на i-ю позицию, т.к. уже используется ранее
            continue;
        if (k <= factorial[n-i]) {
            p[i] = d, was[d] = 1; // ставим d на i-ю позицию
            break;
        }
        k -= factorial[n-i]; // пропускаем (n-i)! перестановок, начинающихся с d
    }
```

Время работы $\mathcal{O}(n^2)$. Для перестановок есть алгоритм за $\mathcal{O}(n \log n)$.

• Номер → объект (правильные скобочные последовательности)

Действуем по той же схеме: пытаемся на первую позицию поставить сперва `'('`, затем `')'`. Разница лишь в том, что чтобы найти «число способов дополнить префикс до правильной скобочной последовательности», нужно пользоваться предподсчитанной динамикой `dp[i,balance]`.

```
int balance = 0;
for (int i = 0; i < n; i++)
    if (k <= dp[i+1, balance+1])
        s[i] = '(', balance++;
    else
        k -= dp[i+1, balance+1], // пропускаем все последовательности, начинающиеся с '('
        s[i] = ')', balance--;
```

11.3. 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 11.3.1. $A \subseteq B \Rightarrow f(A) \leqslant f(B)$

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

11.4. Динамика по подмножествам

Теперь решим несколько простых задач.

• **Число бит в множестве (размер множества)**

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

Заметим, что аналогичный результат можно было получить простым перебором:

```
void go(int i, int A, int result) {
    if (i == n) {
        bit_cnt[A] = result;
        return;
    }
    go(i + 1, A, result);
    go(i + 1, A | (1 << i), result + 1);
}
go(0, 0, 0);
```

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

• **Сумма в множестве.**

Пусть i-й элемент множества имеет вес w_i,
задача: $\forall A$ найти найти сумму весов $s[A] = \sum_{x \in A} w_x$.
Рекурсивное решение почти не поменяется:

```
go(i + 1, A | (1 << i), result + w[i])
```

В решении динамикой, чтобы насчитать $s[A]$, достаточно знать любой единичный бит числа A.
Научимся поддерживать старший бит числа, обозначим его up.

```
up = 0;
for (int A = 1; A < (1 << n); A++):
    if (A == 2^(up+1)) up++;
    s[A] = s[A ^ (1 << up)] + w[up];
```

11.5. Гамильтоновы путь и цикл

Def 11.5.1. *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:

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

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

• Оптимизируем память.

Строка динамики $is[A]$ состоит из n бит, её можно хранить, как одно машинное слово (int).
Физический смысл тогда будет такой:
$ends[A]$ – множество вершин, на которые может заканчиваться путь, проходящий по A.

• Оптимизируем время.

Теперь можно убрать из нашего кода перебор вершины v,
за $\mathcal{O}(1)$ проверяя, есть ли общий элемент у $g[x]$ и $ends[A]$:

```
for (int i = 0; i < n; i++)
    ends[1 << i] = 1 << i;
for (int A = 0; A < (1 << n); A++)
    for (int x = 0; x < n; x++) // Переберём следующую вершину
        if (x ∉ A && g[x] ∩ ends[A] ≠ ∅)
            ends[A | (1 << x)] |= 1 << x;
```

Время работы $\mathcal{O}(2^n n)$, память $\mathcal{O}(2^n)$ машинных слов.
Предполагается, что с числами порядка n все арифметические операции происходят за $\mathcal{O}(1)$.

• Цикл.

База. Начнём с первой вершины. Она, как и все, точно лежит в цикле.
Динамика такая же, насчитали $ends[2^n-1]$. $\mathcal{O}(2^n n)$.
Проверили, что $g[0]$ и $ends[2^n-1]$ имеют общий элемент.

11.6. Вершинная покраска

• **Задача:** покрасить вершины графа в минимальное число цветов так, чтобы соседние вершины имели различные цвета.

Сразу заметим, что вершины одного цвета образуют так называемое «независимое множество»: между вершинами одного цвета попарно нет рёбер.

Предподсчитаем для каждого множества A, $is[A]$ – является ли оно независимым.
Если считать в лоб, $2^n n^2$, но можно и за 2^n:

```
A₁ = A ^ (1 << up); // всё кроме одной вершины
is[A] = is[A₁] && (g[up] ∩ A == ∅)
```

Теперь решим исходную задачу динамикой:
$f[A]$ – минимальное число цветов, чтобы покрасить вершины из множества A.

• **Решение за $\mathcal{O}(4^n)$**

Переберём A, B: $A \subset B \wedge is[B \setminus A]$. B называется надмножеством A.
Динамика вперёд: переход $A \to B$.

```
for A=0..2^n-1
    for B=0..2^n-1
        if A ⊂ B and is[B \ A]
            relax(f[B], f[A] + 1)
```

Время работы $2^n \times 2^n = 4^n$.

11.7. Вершинная покраска: решение за $\mathcal{O}(3^n)$

• **Перебор надмножеств**

Научимся быстрее перебирать все надмножества A. Можно просто взять все $n - |A|$ элементов, которые не лежат в A и перебрать из $2^{n-|A|}$ подмножеств. Можно проще, не выделяя отдельно эти $n - |A|$ элементов.

```
for (A = 0; A < 2^n; A++)
    for (B = A; B < 2^n; B++, B |= A)
        if is[B \ A]
            relax(f[B], f[A] + 1)
```

Благодаря «`B |= A`», понятно, что мы перебираем именно надмножества A. Почему мы переберём все? Мы знаем, что если бы мы выделили те $n - |A|$ элементов, то перебирать нужно было бы все целые число от 0 до $2^{n-|A|} - 1$ в порядке возрастания. Следующее число получается операцией «`+1`», которая меняет младший 0 на 1, а хвост из единиц на нули. Ровно это сделает наша операция «`+1`», разница лишь в том, что биты нашего числа идут вперемешку с «единицами множества A». Пример (красным выделены биты A):

1101011111 Число B
1101100000 Число $B + 1$
1101101001 Число $(B + 1) \,|\, A$

Теорема 11.7.1. Время работы 3^n

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

Доказательство. Другой способ доказать теорему.
Мы считаем $\sum_A 2^{n-|A|} = \sum_C 2^{|C|} = \sum_k \binom{n}{k} 2^k = (1+2)^n = 3^n$ ■

• Перебор подмножеств

Можно было бы наоборот перебирать $A \subset B$ по данному B.

```
for (B = 0; B < 2^n; B++)
    for (C = B; C > 0; C--, C &= B) // все непустые подмножества B
        if is[C]
            relax(f[B], f[B \ C] + 1)
```

Заметим некое сходство: операцией «`C--`» я перехожу к предыдущему подмножеству, операцией «`C &= B`» я гарантирую, что в каждый момент времени C – всё ещё подмножество. Суммарная время работы также $\mathcal{O}(3^n)$. Важная тонкость: мы перебираем все подмножества кроме пустого.

Лекция #12: Динамическое программирование 4

12-я пара, 2024/25

12.1. Вершинная покраска: решение за $\mathcal{O}(2.44^n)$

Def 12.1.1. *Независимое множество A называется максимальным по включению, если $\forall x \notin A \quad A \cup \{x\}$ – не независимо.*

Теорема 12.1.2. $\forall$ графа из n вершин количество максимальных по включению независимых множеств не более $3^{\frac{n}{3}} \approx 1.44^n$.

Доказательство. Рассмотрим алгоритм, перебирающий все максимальные по включению независимые множества. Возможно, некоторые множества он переберёт несколько раз, но каждое хотя бы один раз. Пусть v – вершина минимальной степени, x – степень v.

Идея. Если в максимальном по включению множестве отсутствует v, должен присутствовать один из x её соседей $\Rightarrow$ одна из $x+1$ вершин (v и её соседи) точно лежит в ответе.

```
1 void solve(A, g): // A - текущее независимое множество
2     if (g is empty): // g - граф, вершины которого можно добавить в A
3         print(A) // A точно максимальное по включению
4         return
5     v = вершина минимальной степени в g
6     solve(A ∪ {v}, g \ {v, соседи[v]})
7     for (u : соседи[v])
8         solve(A ∪ {u}, g \ {u, соседи[u]})
9 solve(0, graph) // 0 - пустое множество
```

Lm 12.1.3. Строка с `print` вызовется $\mathcal{O}(1.44^n)$

Поскольку $\forall u \in$ соседи$[v] \quad degree[u] \geqslant degree[v] = x$ имеем
время работы $T(n) \leqslant (x+1)T(n-(x+1)) \leqslant (x+1)^{\frac{n}{x+1}}$. [1]

Чтобы максимизировать эту величину, нужно продифференцировать по x, найти 0...
Опустим эту техническую часть, производная имеет один корень в точке $x+1 = e = 2.71828...$
Но x – степень, поэтому $x+1$ целое $\Rightarrow$ максимум или в $x+1=2$, или в $x+1=3$.
Получаем $2^{\frac{n}{2}} \approx 1.41^n$, $3^{\frac{n}{3}} \approx 1.44^n \Rightarrow T(n) \leqslant 3^{n/3} \approx 1.44^n$. ■

• **Алгоритм за** 2.44^n

Мы умеем за 3^n перебирать A и B – независимые подмножества графа $G \setminus A$, будем перебирать не все независимые, а только максимальные по включению.

Теорема 12.1.4. Время работы 2.44^n

Доказательство. $\sum_A 1.44^{n-|A|} = \sum_C 1.44^{|C|} = \sum_k \binom{n}{k} 1.44^k = (1+1.44)^n = 2.44^n$ ■

[1] Почему не выгодно вычитать из n сперва x_1, затем x_2? Потому что $x_1 x_2 \leqslant (\frac{x_1+x_2}{2})^2$.

12.2. Вершинная покраска: решение за $\mathcal{O}^*(2^n)$

Будем считать число покрасок в ровно k цветов. $\min k$ можно будет найти бинпоиском. На практике мы для каждого множества A за $\mathcal{O}(2^n)$ найдём $f[A]$ – число независимых подмножеств. Каждый цвет – какое-то независимое подмножество. Давайте возьмём k любых, даже пересекающихся. Есть $f[2^n-1]^k$ способов выбрать. Мы посчитали лишние способы. Те, которые в итоге покроют не все n вершин. Используем формулу включения-исключения.

$$ans = \sum_A (-1)^{n-|A|} f[A]^k$$

12.3. 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: $(\bigcup_{i\in I} A_i = U) \wedge (|I| \to \min)$. 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.

12.4. Bit reverse

Ещё одна простая задача: развернуть битовую запись числа.
Сделать предподсчёт за $\mathcal{O}(2^n)$ для всех n битовых чисел.
Например 00010110 $\to$ 01101000 при $n = 8$. Решение заключается в том, что если откинуть младший бит числа, то `reverse` остальных битов мы уже знаем, итак:

```
1 reverse[0] = 0;
2 for (int x = 1; x < (1 << n); x++)
3     reverse[x] = (reverse[x >> 1] >> 1) + ((x & 1) << (n - 1));
```

Задача, как задача. Понадобится нам, когда будет проходить `FFT`.

12.5. Meet in the middle

12.5.1. Количество клик в графе за $\mathcal{O}(2^{n/2})$

Сперва напишем рекурсивное решение за $\mathcal{O}(2^n)$.
Перебираем по очереди вершины и каждую или берём, или не берём.

```
1 int CntCliques(int i, int A):
2     if (A == 0) return 1;
3     return CntCliques(i+1, A & ~2^i) + ((A & 2^i) ? CntCliques(i+1, A & g[i]) : 0);
4 print(CntCliques(0, 2^n-1));
```

Здесь A – множество вершин, которые мы ещё можем добавить в клику.
$g[i]$ – множество соседей вершины A.

Параметр i можно не таскать за собой, а вычислять `i = lower_bit(A)`.
Функцию `lower_bit` мы умеем реализовать за $\mathcal{O}(1)$ (см. практику).

У нас есть код, работающий за $\mathcal{O}(2^n)$. Код является перебором. Ответ (`int`, который вернёт

перебор) зависит только от параметра `A` ⇒ можно добавить запоминание (мем**ои**зацию) от `A`. Перебор превратится при этом в «ленивую динамику».

```
1 int CountCliques(int A):
2     if (A == 0) return 1;
3     if (f[A] != 0) return f[A];
4     int i = lower_bit(A);
5     return f[A] = CountCliques(A ^ 2^i) + CountCliques(A & g[i]);
```

Мы реализовали естественную идею «если уже вычисляли значение функции от A, зачем считать ещё раз? можно просто запомнить результат». От применения такой оптимизации к любому перебору асимптотика времени работы точно не ухудшилась, константа времени работы могла увеличиться. В данном конкретном случае случилось чудо:

Теорема 12.5.1. Перебор с запоминанием `CountCliques` работает за $\mathcal{O}(2^{n/2})$

Доказательство. Посчитаем число пар (i, A) для которых мы заходили в `CountCliques`.
Пар с $i \leqslant \frac{n}{2}$ не больше 2^i потому что $i \geqslant$ глубины рекурсии,
на каждом уровне рекурсии мы ветвились на 2 ветки.
Пар с $i > \frac{n}{2}$ не больше 2^{n-i} благодаря запоминанию:
у множества A первые i битов нули ⇒ различных A не более 2^{n-i}. ■

• Meet in the middle.

В доказательстве отчётливо прослеживается идея «разделения на две фазы».
Запишем то же решение иначе: разобьём n вершин на два множества – A и B по $\frac{n}{2}$ вершин.
Для всех $2^{n/2}$ подмножеств $B_0 \subseteq B$ за $\mathcal{O}(2^{n/2})$ предподсчитаем `CountCliques[`B_0`]`.
Можно тем же перебором, можно нерекурсивно, как в задаче с практики.
Теперь будем перебирать подмножество вершин $A_0 \subseteq A$, посчитаем множество соседей A_0 в B: $N = \bigcap\limits_{x \in A_0} g[x]$, добавим к ответу уже посчитанное значение `CountCliques[N(`A_0`)]`.
Сравним реализации на основе идеи «meet in the middle» и перебора с запоминанием:

1. «meet in the middle» использует массив `CountCliques[]`,
   рекурсивный перебор использует хеш-таблицу `f[]`.
2. «meet in the middle» можно написать без рекурсии ⇒ константа времени работы меньше.
3. «meet in the middle» работает $\Theta(2^{n/2})$, а рекурсивный перебор $\mathcal{O}(2^{n/2})$, что часто гораздо меньше. Чтобы интуитивно это понять, примените рекурсивный перебор к сильно разреженному графу, заметьте, как быстро уменьшается A.

12.5.2. Рюкзак без весов за $\mathcal{O}(2^{n/2})$

Задача: дано множество чисел $A = \{a_1, \ldots, a_n\}$ выбрать подмножество с суммой ровно S.

Решение. Разобьём A на две половины по $\frac{n}{2}$ чисел, в каждой половине для каждого из $2^{n/2}$ подмножеств посчитаем сумму, суммы сложим в массивы B_1 и B_2.
Нужно выбрать $x_1 \in B_1, x_2 \in B_2$: $x_1 + x_2 = S$.
Мы это умеем делать несколькими способами: (отсортируем B_1, B_2, пустим два указателя) или (сложим все $x_1 \in B_1$ в хеш-таблицу, $\forall x_2 \in B_2$ поищем $S - x_2$ в хеш-таблице).

12.5.3. Рюкзак с весами за $\mathcal{O}(2^{n/2}n)$

Задача: «унести в рюкзаке размера W вещи суммарной максимальной стоимости».

Решение. Разобьём вещи произвольным образом на две половины по $\frac{n}{2}$ вещей.
В каждой половине есть $2^{n/2}$ подмножеств, мы можем для каждого такого подмножества насчитать за $\mathcal{O}(2^{n/2})$ его вес и стоимость (или рекурсия, или динамика).
Получили два массива пар $\langle w_1, cost_1\rangle$`[]`, $\langle w_2, cost_2\rangle$`[]`.
Зафиксируем, что мы возьмём из первой половины: $\langle w_1, cost_1\rangle[i]$.
Из второй половины теперь можно взять любую пару, что $w_2[j] \leqslant W - w_1[i]$.
Среди таких хотим максимизировать $cost_2[j]$.
Заранее отсортируем массив $\langle w_2, cost_2\rangle$`[]` по w_2 и насчитаем все максимумы на префиксах.

Алгоритм 12.5.2. *Алгоритм за* $\mathcal{O}(2^{n/2}n)$

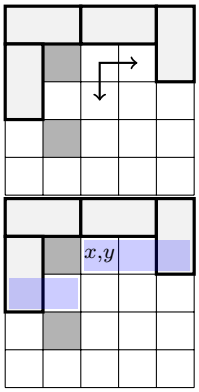
```
генерируем массивы <w1, cost1>[], <w2, cost2>[] // O(2^{n/2})
сортируем пары <w2, cost2>[] по w2 // O(2^{n/2} n)
f = префиксные максимумы на массиве cost2[] // O(2^{n/2})
for i=0..2^{n/2}-1 do // O(2^{n/2} n)
    j = lower_bound(w2[], W - w1[i])
    relax(answer, cost1[i] + f[j])
```

Можно вместо бинпоиска отсортировать оба массива и воспользоваться методом двух указателей, что лучше, так как $\mathcal{O}(sort + n)$ иногда меньше чем $\mathcal{O}(n \log n)$ (CountSort, BucketSort).

Общая схема meet-in-the-middle – разделить задачу размера n на две части по $\frac{n}{2}$.

12.6. Динамика по скошенному профилю

Ещё раз посчитаем число способов покрыть доску размера $w \times h$ доминошками. Мы уже решили эту задачу в теме «рекурсия». Сейчас мы назвали бы это «ленивой динамикой». Напишем ту же динамику нерекурсивно. Мы остановились на том, что хотим считать `cnt[x,y,A]`.



Маска A – число до 2^w, биты в изломаном профиле. i-й бит маски: $(A \gg i)\&1$. Если клетка $[x, y]$ уже покрыта $\Leftrightarrow (A \gg i)\&1 \Rightarrow$ переход $[x, y, A] \to [x{+}1, y, A{-}2^i]$. Иначе всегда есть переход: $[x, y, A] \to [x{+}1, y, A \mid 2^i]$ и, если не покрыта $[x{+}1, y]$, то ещё и $[x, y, A] \to [x{+}1, y, A \mid 2^i \mid 2^{i+1}]$.

```
m[0,0,0] = 1
for (int y = 0; y < h; y++)
    for (int x = 0; x < w; x++):
        x1 = x + 1, y1 = y; // следующая за [x,y] клетка [x1,y1]
        if (x1 == w) x1 = 0, y1++;
        for (int A = 0; A < (1 << w); A++)
            if (bit(A, x))
                cnt[x1][y1][A - (1<<x)] += cnt[x][y][A]
            else:
                cnt[x1][y1][A | (1<<x)] += cnt[x][y][A]
                if (!bit(A, x+1)) cnt[x1][y1][A | (3<<x)] += cnt[x][y][A]
```

Написав без рекурсии, мы получили следующие плюсы-минусы:

+ Нет рекурсии ⇒ меньше константа времени работы
+ Если хранить только один слой, получим динамику `cnt[A]`, уменьшим память память до 2^w.
- Чуть сложнее код в том месте, где мы по старому `A` пытаемся посчитать новое `A`.
- В рекурсивной версии из-за ленивости посещаются только достижимые состояния.
  Например, в задаче про доминошки 10×10 из $10 \cdot 10 \cdot 1024$ состояний посетим только 9371.

12.7. Поиск максимального независимого множества за $\mathcal{O}(1.38^n)$

Поиск клики и независимого множества – одна и та же задачу (наличие ребра $\rightleftarrows$ отсутствие).

Алгоритм. Выберем вершину v максимальной степени и переберём брать её, или не брать.
Если не брать, v просто удалится из графа: рекурсивный вызов $n \to n-1$
Если брать, из графа удалится v и все её соседи: рекурсивный вызов $n \to n-1-\deg v$.
Если максимальная степень вершины не более 2, граф – набор путей и циклов, и задачу можно решить жадно без перебора $\Rightarrow \max \deg v \geqslant 3 \Rightarrow T(n) \leqslant T(n-1) + T(n-4) \approx 1.38^n$.

Лекция #13: Графы и базовый поиск в глубину

13-я пара, 2024/25

13.1. Опеределения

Def 13.1.1. *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 называть граф без петель и кратных рёбер.

Def 13.1.2. *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 13.1.3. 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.

13.2. 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:

```
1 struct Edge { int from, to, weight; };
2 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.

13.2.1. Мультисписок

Рёбра, смежные с v, лежат в односвязном списке $head[v]$, $next[head[v]]$, $next[next[head[v]]]$...
Все перечисленные элементы – номера рёбер.
По номеру ребра e можем хранить любую информацию про него, например, куда оно ведёт.

```
struct MultiList {
    struct Edge { int next, to; };
    vector<int> head; // для каждой вершины первое ребро
    vector<Edge> es; // все рёбра графа
    int e; // количество рёбер
    MultiList(int n, int m) :
        head(n, -1), // -1 = признак конца списка
        es(m), // максимальное число рёбер в графе
        e = 0 { } // изначально рёбер нет
    void addEdge(int a, int b) { // одно ориентированное ребро
        es[e] = {head[a], b}, head[a] = e++;
    }
    void adjacent(int v) { // все рёбра смежные с v
        for (int e = head[v]; e != -1; e = es[e].next)
            cout << es[e].next << " "; // а можем делать что-нибудь другое
    }
};
```

По сути эти те же «списки смежности», но более аккуратно сохранённые.

Пусть $V = E = 10^6$, граф случайный. Оценим память.
На 64-битной машине `vector<vector<int>> g` будет в итоге весить $X = ?_1 = 3 \cdot 8 + ?_2$ мегабайта (сам по себе вектор – 3 указателя). Можно подумать, что $?_2 = E \cdot 4$, но нет, в векторе `size` $\neq$ `capacity` $\Rightarrow$ нужно проводить эксперимент. На двух экспериментах видим, что $?_2 \approx E \cdot 12$ и $E \cdot 16$. Итого X ≈36-38 мегабайт.

Мультисписок $12 = 3 \cdot 4$ мегабайта. Разница в ≈3 раза. По скорости создания/удаления (выделить память, добавить элементы, освободить память) мультисписок будет в ≈10 раз быстрее.

13.3. 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 13.3.1. *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!

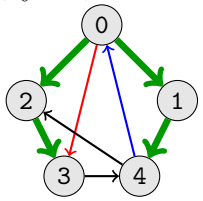
```
bool dfs(int v):
    if (v == goal) { path ← v; return 1; }
    mark[v] = 1;
    for (int x : g[v])
        if (dfs(x)) { path ← x; return 1; }
    return 0;
dfs(start);
```

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

13.4. 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 13.4.1. 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++; // время входа
    ...
    t_out[v] = T++; // время выхода
```

13.5. Топологическая сортировка

Def 13.5.1. *DAG – Directed Acyclic Graph (ориентированный ациклический граф).*
Рёбра (1→2)(2→4)(1→3)(3→4) *образуют DAG, рёбра* (1→2)(2→3)(3→1) *– нет.*

Def 13.5.2. *Топологической сортировкой орграфа –*
сопоставление вершинам номеров $ind[v]$: $\forall e(a \to b)\ ind[a] < ind[b]$.

Lm 13.5.3. Топологическая сортировка $\exists$ iff граф ациклический.

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 ациклическим, 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 13.5.4. *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.

13.6. Поиск цикла

В неорграфе: если пришли dfs-ом в вершину, где уже были, зациклились. Пример: $1 \to 2 \to 3 \to 4 \to 5 \to 3$. Восстановление: какой-то суффикс пройденного пути. Конечно, нужно запретить ходить из v по ребру, по которому вошли в v.

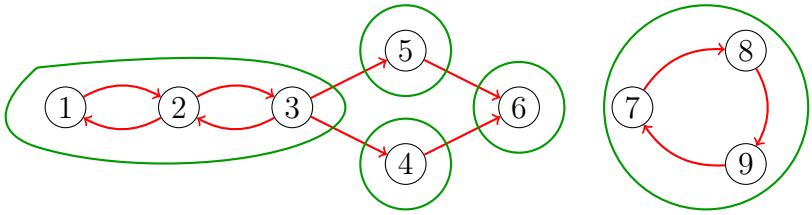
В орграфе: чуть аккуратнее. Вошли в вершину, в которую уже вошли, но ещё не вышли $\Rightarrow$ зациклились. В примере (1→2)(2→4)(1→3)(3→4) когда мы второй раз входим в 4, мы из неё уже вышли.

```
bool dfs(int v):
    color[v] = 1 // уже вошли, иногда говорят «серые вершины»
    ...
    color[v] = 2 // уже вышли, иногда говорят «чёрные вершины»
```

13.7. Компоненты сильной связности

Def 13.7.1. *В* **орграфе** *вершины a и b сильно связны iff существуют пути $a \rightsquigarrow b$ и $b \rightsquigarrow a$.*

Def 13.7.2. *Компонента сильной связности – класс эквивалентности отношения сильной связности (рефлексивно, симметрично, транзитивно).*



Научимся выделять компоненты сильной связности.

• **Алгоритм за $\mathcal{O}(kE)$, где k – число компонент**

Берём вершину v, запускаем 2 `dfs` из v: 1-й по прямым (вершины, достижимые из v)» и 2-й по обратным рёбрам (вершины, из которых достижима v). Помеченные 2 раза вершины – к.с.с v.

• **Алгоритм за $\mathcal{O}(V+E)$**

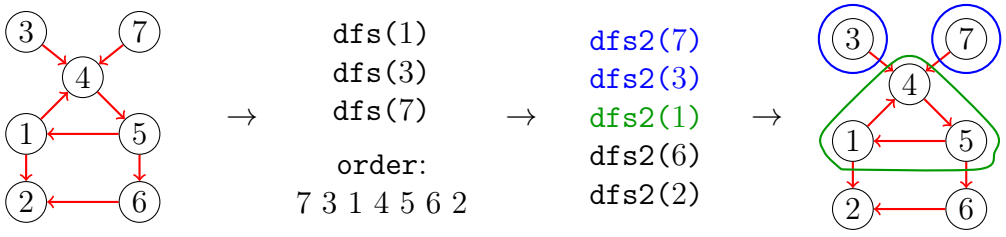
1. Выпишем вершины в порядке убывания времени выхода из `dfs`.

```
void dfs(int v):
    ...
    order.push_front(v);
for (int v = 0; v < n; v++)
    if (!used[v])
        dfs(v); // dfs ходит по прямым рёбрам графа
```

Поскольку наш код делает то же, что и topsort (Algo 13.5.4), иногда говорят «запустим топсорт».

2. Перебираем вершины в полученном порядке. Если вершина v ещё не покрашена, красим «*не покрашенные вершины, достижимые из неё по обратным рёбрам*». Утверждается, что покрасим ровно компоненту сильной связности v.

```
int cc = 0; // количество компонент
for (int v : order)
    if (color[v] == 0)
        dfs2(v, ++cc); // dfs2 ходит по обратным рёбрам графа
```



Def 13.7.3. *Конденсация орграфа $G = \langle V, E \rangle$ – новый граф, результат стягивания компонент сильной связности. Вершинами конденсации являются компоненты сильной связности G, а*

рёбра G: $(a, b) \in E$ *порождают рёбра конденсации* $(color[a], color[b])$.
Петли, то есть случай $color[a] = color[b]$*, в конденсацию обычно не добавляют.*

Теорема 13.7.4. Корректность алгоритма сильной связности за $\mathcal{O}(V + E)$

Доказательство. `dfs2(v)` точно обошёл всю компоненту сильной связности v. Почему он не обошёл лишнего? Пусть есть вершина x, достижимая по обратным рёбрам, но не достижимая по прямым. У неё есть компонента сильной связности $C(x)$. Утверждается, что до v мы уже перебрали хотя бы одну $a \in C(x) \Rightarrow C(x)$ уже была покрашена. См. лемму. ■

Lm 13.7.5. Если есть ребро $x \to v$, но x и v в разных компонентах, $\exists\, a \in C(x)$: $pos_a < pos_v$

Доказательство. Достаточно доказать для соседних компонент в конденсации.

Пусть x' – первая вершина $\in C(x)$, до которой дошёл dfs.
Пусть v' – первая вершина $\in C(v)$, до которой дошёл dfs. Если мы сперва посетили v', то в x' из неё мы не дойдём $\Rightarrow$ в `order` вся $C(v)$ будет записана позже всей $C(x)$. Если мы сперва посетили x' – мы не выйдем из неё, пока не посетим все вершины $C(x) \Rightarrow$ пройдём ребро, ведущее в $C(v) \Rightarrow$ обойдём всю $C(v) \Rightarrow$ из v выйдем до того, как выйдем из $x \Rightarrow x'$ – искомая вершина a. ■

Лекция #14: Поиск в глубину (часть 2)

14/15-я пара, 2024/25

14.1. Примеры кода

```
struct Edge { int to, weight; }; // конструктуры/деструкторы/классы не нужны
vector<vector<Edge>> g(n); // наш граф
g[a].push_back({a, w}); // добавить ребро
function<void(int)> dfs = [&](int v) { // наш dfs
```

14.2. Эйлеровость

Def 14.2.1. *Рёберно простой цикл/путь* ***эйлеров****, если содержит все рёбра графа.*

Def 14.2.2. *Граф* ***эйлеров****, если содержит эйлеров цикл.*

Теорема 14.2.3. Связный **неорграф** эйлеров iff все степени вершин чётны.

Доказательство. Возьмём ∀ цикл, удалим его рёбра, в оставшихся компонентах связности по индукции есть эйлеровы циклы, соединим всё это счастье. Чтобы найти ∀ цикл, возьмём ∀ вершину v, пойдём жадно вперёд, удаляя пройденные рёбра: поскольку все степени чётны, остановиться мы можем только в v. ■

• Алгоритм построения

Будем делать всё, как в доказательстве теоремы. Сперва найдём ∀ цикл.

```
vector<set<int>> g; // наивный способ хранения, удобно удалять рёбра
void dfs( int v ) {
    if (!g[v].empty()) {
        int x = *g[v].begin();
        g[v].erase(x), g[x].erase(v);
        dfs(x);
        answer.push_back(edge(v, x));
    }
}
```

Слово `dfs` уже намекает на рекурсию. Чтобы в оставшихся компонентах связности найти эйлеровы циклы и вставить в наш в нужные места, сделаем `+1` рекурсивный вызов...

```
vector<set<int>> g; // наивный способ хранения, удобно удалять рёбра
void dfs( int v ) {
    while (!g[v].empty()) { // единственное изменение кода
        int x = *g[v].begin();
        g[v].erase(x), g[x].erase(v);
        dfs(x); // dfs(x) найдёт цикл одной из компонент и вставит ровно сюда
        answer.push_front(edge(v, x));
    }
}
```

Чтобы алгоритм работал за линию, нам бы от `set`-а избавиться. Будем удалять лениво:

```
struct Edge { int a, b, isDeleted; };
vector<Edge> edges;
vector<vector<int>> ids; // каждое ребро встретится два раза
```

В цикле `while` вынимаем все рёбра, пропускаем с пометкой `isDeleted`.
Идя из вершины v по ребру e, мы попадём в вершину $e.a + e.b - v$.

Lm 14.2.4. Слабосвязный **орграф** эйлеров iff у всех вершин входящая степень равна исходящей.

Доказательство и алгоритм аналогичны. Реализация алгоритма даже проще: орребро не нужно удалять из вектора другой вершины $\Rightarrow$ для хранения графа достаточно `vector<vector<Edge>`.

14.3. Раскраски

Def 14.3.1. *Задача вершинной покраски – раскрасить вершины графа так, чтобы смежные были разных цветов.*

Что мы уже знаем?

Покрасить вершины в 2 цвета мы умеем за $\mathcal{O}(V+E)$ dfs-ом.
Покрасить в минимальное число цветов мы умеем динамикой за $\mathcal{O}(2.44^n)$. На самом деле красить уже в 3 цвета трудно (люди умеют только за экспоненту).

Как же красить большие графы?

Жадно за $\mathcal{O}(V+E)$! Получится не минимальное число цветов, но тоже неплохо.

• **Алгоритм**

Берём вершину минимальной степени, удаляем её из графа. Красим всё кроме неё, и в конце докрашиваем её в минимально возможный цвет.

Реализация за $\mathcal{O}(V+E)$: нужно уметь быстро выбирать вершину минимальной степени. Степени только уменьшаются причём ровно на 1 $\Rightarrow$ поддерживаем `list[degree]` – для каждой степени список всех вершин такой степени и указатель на минимальную степень. Собственно удаление вершины: каждого из её соседей за $\mathcal{O}(1)$ переместить в соседний список.

Lm 14.3.2. Если в каждый момент алгоритм $\min(degree_v) \leqslant x$, то наш алгоритм успешно покрасит граф в $\leqslant x+1$ цветов.

Пример: любое дерево покрасится в 2 цвета.

Утверждение 14.3.3. Поскольку в любом планарном графе есть вершина степени $\leqslant 5$, наш алгоритм раскрасит его в $\leqslant 6$ цветов.

14.4. Рёберная двусвязность

Def 14.4.1. *Говорят, что две вершины рёберно двусвязны iff между ними есть два рёберно не пересекающихся пути.*

Def 14.4.2. *Рёберная двусвязность – отношение эквивалентности на вершинах неорграфа. Классы эквивалентности – компоненты рёберной двусвязности.*

Теорема 14.4.3. Рёберная двусвязность – отношение эквивалентности.

Доказательство. Рефлексивность и симметричность очевидны.
Транзитивность: (u, v) и (v, w) рёберно двусвязны, доказываем двусвязность (u, w).
Возьмём два пути $u \leftrightarrow v$, они образуют рёберно простой цикл C.
Пойдём двумя путями из w в v, остановимся, когда дойдём до цикла: мы на цикле в вершинах a и b, чтобы из них попасть в u – выберем нужное направление и пойдём по циклу C. ■

Def 14.4.4. *Ребро (a, b) называется мостом, если после его удаления a и b не связны.*

Замечание 14.4.5. После удаления из графа мостов, компоненты связности и рёберной двусвязности будут совпадать. Доказательство: ((a, b) – не мост) $\Leftrightarrow$ (a и b связны после удаления (a, b)).

Тривиальный алгоритм поиска мостов за $\mathcal{O}(VE)$: заметим, что любой мост лежит в остовном дереве $\Rightarrow$ dfs-ом найдём любой остов, каждое ребро остова проверим за $\mathcal{O}(E)$.

• **Алгоритм поиска мостов за $\mathcal{O}(V + E)$**

Начало, как и в тривиальном: обойдём граф `dfs`-ом, посмотрим на остовное дерево, которое оставит нам `dfs` и на древесное ребро $e\colon v \to x$ в нём. Теперь придумаем, как за $\mathcal{O}(1)$ проверить, является ли e мостом. Не мост iff из поддерева x можно пройти наверх в обход e. Чтобы понимать, можно ли, стоя в x, пробраться наверх, насчитаем *динамику по дереву* — самая высокая вершина, которую можно достигнуть из x, если запрещено ходить по ребру $(v \leftrightarrow x)$.

Замечание 14.4.6. В канонической реализации вместо «самой высокой вершины» обычно ищут «минимальную по времени входа», разницы никакой, просто так принято.

Замечание 14.4.7. Важно, что мы используем именно `dfs`. Не любой обход графа подойдёт, например, `bfs` не подойдёт. См. доказательство.

Ниже мы вместе с мостами найдём сразу и компоненты.
Как только выходим из `dfs` и находим мост, на нём висит новая компонента, вершины которой мы уже обошли и заботливо сложили в стек, остаётся только не забыть достать.
Для полного понимания алгоритма прочитайте комментарии в коде.

```
1  void dfs(int v, int parent):
2      stack.push(v) // чтобы восстановить компоненты двусвязности
3      time[v] = ++T // время входа (а можно использовать высоту)
4      min_time[v] = ++T; // наша динамика по дереву
5      for (int x : adjacent[v])
6          if (x != parent) { // если бы в графе были кратные рёбра, проверка была бы сложнее
7              if (time[x] == 0) { // ещё не ходили в x ⇒ идём и делаем ребро древесным
8                  int stack_level = stack.size(); // уровень стека до рекурсивного вызова
9                  dfs(x, v);
10                 if (min_time[x] > time[v]) { // мост!
11                     vector<int> component;
12                     while (stack.size() > stack_level)
13                         component.push(stack.pop())
14                 }
15                 relaxMin(min_time[v], min_time[x]);
16             } else { // обратное или прямое ребро (x или предок, или потомок v в дереве)
17                 relaxMin(min_time[v], time[x]);
18             }
19         }
```

Строки 2, 8 и 11-13 нужны, чтобы восстановить компоненты в том же `dfs`. Если мосты находятся корректно, то в 11-13 мы откусываем компоненту, которая висит на ребре $v \to x$.
`time[v]` – время входа в вершину v.
`min_time[v]` – динамика по дереву, минимальное достижимое время входа из v по путям, которые спускаются по древесным рёбрам dfs, а потом делают один шаг по произвольному ребру.

Теорема 14.4.8. Алгоритм корректно находит все мосты

Доказательство. Нужно доказать, что мы нашли мосты и только их.
Если ребро $e\colon v \to x$ мост, то из x нельзя достичь $v \Rightarrow$ `min_time[x]` $>$ `time[v]` $\Rightarrow$ наш алгоритм корректно поймёт, что e — мост.

Если $e\colon v \to x$ не мост, то есть путь p из x в v в обход e. Поскольку мы используем именно `dfs`, кроме древесных и прямых рёбер есть только обратные и только они могут вести из поддерева x наружу, причём вести они могут только в вершины $z \in$ пути $root \rightsquigarrow v$. Осталось заметить, что $min\_time[x] \leqslant time[z] \leqslant time[v] \Rightarrow$ наш алгоритм корректно поймёт, что e — не мост. ■

В неорграфе `dfs` не производит перекрёстных рёбер $\Rightarrow$, то из поддерева x есть путь, заканчивающийся на обратное ребро, ведущее в v или выше $\Rightarrow$ `min_time[x]` $\leqslant$ `time[v]`.

Замечание 14.4.9. Рёберная двусвязность – отношение эквивалентности $\Rightarrow$ относительно него граф можно сконденсировать. Получится лес, вершинами которого являются компоненты рёберной двусвязности, а рёбрами – мосты.

14.5. Вершинная двусвязность

Def 14.5.1. *Говорят, что два* **ребра** *вершинно двусвязны iff есть вершинно простой цикл, содержащий оба ребра.*

Def 14.5.2. *Вершинная двусвязность – отношение эквивалентности на* **рёбрах** *неорграфа. Классы эквивалентности – множества рёбер, компоненты вершинной двусвязности.*

Теорема 14.5.3. Вершинная двусвязность – отношение эквивалентности.

Доказательство. Рефлексивность и симметричность очевидны.
Транзитивность: (e_1, e_2) и (e_2, e_3) вершинно двусвязны, доказываем двусвязность (e_1, e_3).
Возьмём простой цикл C через e_1 и e_2. Пойдём двумя путями из концов e_3 в концы e_2, остановимся, когда дойдём до цикла: мы на цикле в различных вершинах a и b, чтобы из них попасть в концы e_1, выберем нужное направление и пойдём по циклу C. ■

Замечание 14.5.4. Если ребро мост, оно одиноко, в его компоненте двусвязности лишь одно ребро.

Def 14.5.5. *Точка сочленения – вершина, при удалении которой увеличивается число компонент связности.*

Утверждение 14.5.6. Точки сочленения – ровно те вершины, у которых есть смежные рёбра из разных компонент вершинной двусвязности.

Тривиальный алгоритм поиска точек сочленения за $\mathcal{O}(VE)$: проверим каждую вершину v за $\mathcal{O}(E)$ `dfs`-ом, проверяющим граф на связность, запретив ему ходить через v. Затем, когда уже известны точки сочленения, компоненты вершинной двусвязности можно найти за $\mathcal{O}(V+E)$

почти обычным `dfs`-ом для поиска компонент связности – ходим рекурсивно из ребра в вершину, из вершины в ребро: ребро → вершина → ребро → вершина → ..., запрещаем проходить через точки сочленения.

• **Алгоритм поиска точек сочленения и компонент двусвязности за $\mathcal{O}(V+E)$**

Почти идентичен поиску мостов. Идея та же: если мы из точки сочленения v пошли вглубь по ребру e: $v \to x$, подняться выше, не проходя через v, мы не сможем ⇒

```
min_time[x] >= time[v]
```

Это неравенство и будет критерием того, что мы нашли новую компоненту. Если неравенство окажется строгим, ребро ещё и будет мостом. По аналогии с алгоритмом поиска мостов, рёбра новой компоненты обнаружатся на вершине стека, в который мы бережно складываем все пройденные рёбра. Для полного понимания алгоритма прочитайте все комментарии в коде ниже.

```
void dfs(int v, int parent):
    time[v] = min_time[v] = ++T;
    int count = 0; // сколько компонент уже цепляется снизу за вершину v
    for (int x : adjacent[v])
        if (x != parent):
            int stack_level = stack.size();
            if (ещё не ходили по ребру {x,v}) // неорграф ⇒ по каждому ребру ходим дважды!
                stack.push({v,x})
            if (time[x] == 0) {
                dfs(x, v);
                if (min_time[x] >= time[v]) { // новая компонента
                    vector<Edge> component;
                    while (stack.size() > stack_level)
                        component.push(stack.pop())
                    // Вершина v является точкой сочленения, если на ней смыкаются хотя бы
                    // две компоненты. Одна точно есть, мы её только что нашли.
                    // Где может быть вторая? Если v — не корень, выше. Если count ⩾ 2, ниже.
                    if (parent != -1 || ++count >= 2)
                        points.push_back(v); // v — точка сочленения
                }
                relaxMin(min_time[v], min_time[x]);
            } else {
                relaxMin(min_time[v], time[x]);
            }
```

Теорема 14.5.7. Алгоритм корректно находит все компоненты и точки сочленения

Доказательство. Похоже на историю с мостами. Далее мы предполагаем, что у читателя в кеше доказательство про мосты Thm 14.4.8. Пусть мы шли в `dfs` $p \to v \to x$. Рёбра $p \to v$ и $v \to x$ лежат в одной компоненте iff есть путь $x \to$ предок v в дереве `dfs` $\Leftrightarrow min\_time[x] < time[v]$. ■

14.6. 2-SAT

Задача 2-SAT: нам дана КНФ формула φ, в каждом клозе не более двух литералов. Нужно найти выполняющий набор x_i, или сказать, что φ противоречива. Пример:

$$\varphi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_3) \;\longrightarrow\; (x_1 = 1,\ x_2 = 0,\ x_3 = 1)$$

Примеры задач, которые можно решить, используя 2-SAT:

• **Расположение геометрических объектов без наложений**

Если для каждого объекта есть только два варианта расположения, то задача расположения объектов без пересечений сводится к 2-SAT. Пример: есть множество отрезков на плоскости, каждому нужна подпись с одной из двух сторон отрезка.

• **2-List-Coloring** *(будет разобрана на практике)*

Дан неорграф граф, для каждой вершины v есть список $l[v]$ из двух цветов.
Покрасить вершины так, чтобы соседние были покрашены в разные цвета.
Сведение: x_v – номер выбранного цвета в $l[v]$, каждое ребро (a, b) для каждого цвета $c \in l[a] \cap l[b]$ задаёт клоз $\neg(x_a{=}pos_{a,c} \wedge x_b{=}pos_{b,c}) \Leftrightarrow (x_a{=}\overline{pos_{a,c}} \vee x_b{=}\overline{pos_{b,c}})$

• **Кластеризация на два кластера** *(2-sat можно заменить на раскраску в 2 цвета)*

Даны объекты, и матрица расстояний d_{ij} (непохожести объектов). Нужно разбить объекты на два множества: max из диаметров $\to$ min. Решение: бинпоиск по ответу, внутри 2-SAT.

14.6.1. Решение 2-SAT за $\mathcal{O}(nm)$

Попробуем жадно сказать $x_1 = 0$. Если был клоз вида $(x_1 = 1 \vee x_i = e)$, то поскольку $x_1 = 0$ мы точно можем сделать вывод $x_i = e$. Осталось увидеть «граф следствий», в котором $2n$ вершин вида $x_i = 0$, $x_i = 1$ и рёбра – выводы, которые мы можем сделать. Сделаем dfs из вершины $x_1 = 0$. Если по ходу такого dfs-а мы посетим две вершины с противоположным смыслом: $x_i = 0$, $x_i = 1$, это противоречие. Если пришли к противоречию $\Rightarrow x_1 = 1$, запустим dfs уже из неё. Если всё равно противоречие $\Rightarrow$ решений нет. Если не пришли к противоречию $\Rightarrow$ все сделанные нами выводы корректны, мнение мы уже не изменим.

Время работы: dfs пускаем не более $2n$ раз, каждый отработает за $\mathcal{O}(m)$.

14.6.2. Решение 2-SAT за $\mathcal{O}(n + m)$

Клоз $(a \vee b) \Leftrightarrow$ двум импликациям $(\neg a \Rightarrow b)$ и $(\neg b \Rightarrow a)$.
Построим граф следствий. Вершины графа – литералы x_i и $\neg x_i$.
Для каждого клоза $(a \vee b)$ проведём рёбра $\neg a \to b$ и $\neg b \to a$.
Если n – число неизвестных, m – число клозов, мы получили $2n$ вершин и $2m$ рёбер.

Lm 14.6.1. Решение 2-SAT – для каждого i в графе следствий выбрать ровно одну из двух переменных x_i, $\neg x_i$ так, чтобы не было рёбер из выбранных в невыбранные.

Lm 14.6.2. Путь в графе следствий $x_i \rightsquigarrow \neg x_i$, означает что в любом корректном решении $x_i = 0$.

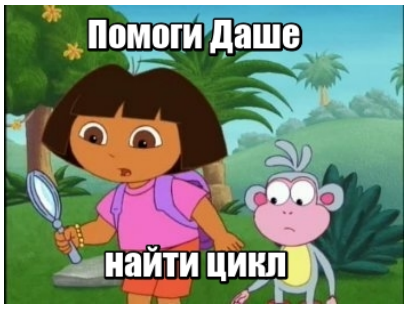
Следствие 14.6.3. $\exists i$: x_i и $\neg x_i$ в одной компоненте сильной связности $\Rightarrow$ формула противоречива

Теорема 14.6.4. $\exists i$: x_i и $\neg x_i$ в одной компоненте сильной связности $\Leftrightarrow$ формула противоречива

Доказательство. Пусть $\forall i$ x_i и $\neg x_i$ в разных компонентах. Алгоритм расстановки значений x_i:

1. Топологически отсортируем конденсацию графа.
2. $k[x_i]$ – номер компоненты в топологическом порядке. $\forall$ ребра графа $a \to b$ верно $k[a] \leqslant k[b]$.
3. Для каждой переменной i выберем x_i iff $k[x_i] > k[\neg x_i]$.

Корректность решения:



Если в решение есть противоречие, то по Lm 14.6.1 есть выбранный y, невыбранный z и ребро $y \to z$. Рёбра добавлялись парами $\Rightarrow$ есть и ребро $\neg z \to \neg y$. Из $\exists$-я этих рёбер делаем вывод: $k[z] \geqslant k[y] \wedge k[\neg y] \geqslant k[\neg z]$. Из выбранности y, не выбранности z вывод: $k[y] > k[\neg y] \wedge k[\neg z] > k[z]$.

■

Итого мы получили решение а $\mathcal{O}(V+E)$: построить граф + найти ксс + вывести ответ. Заметьте, сами компоненты хранить не нужно, мы строим только массив `k[]`.

• **Упрощение реализации**

Предпологая, что ксс мы ищем через два dfs-а, можно модифицировать второй из них:

```
1 void paintComponent(int v) {
2     used[v] = 1;
3     if (x[v/2] == -1) // пусть вершины x_i, ¬x_i идут парами; -1 значит неопределено
4         x[v/2] = 1 - v%2; // для x_{v/2} выберем не v, так как компоненту v перебираем раньше
5     for (int x : graph[v])
6         if (!used[x])
7             paintComponent(x);
8 }
```

Лекция #15: Введение в теорию сложности

16-я пара, 2024/25

15.1. Decision/search/разрешимость

• Decision/search problem

Если в задаче ответ – `true`/`false`, то это *decision problem (задача распознавания)*.
Иначе это *search problem (задача поиска)*. **Примеры:**

1. Decision. Проверить, есть ли x в массиве a.
2. Search. Найти позицию x в массиве a.
3. Decision. Проверить, есть ли путь из a в b в графе G.
4. Search. Найти сам путь.
5. Decision. Проверить, есть ли в графе подклика размера хотя бы k.
6. Search. Найти максимальный размер подклики.
7. Search. Найти подклику максимального размера.

Decision problem f можно задавать, как язык (множество входов) $L = \{x\colon f(x) = \texttt{true}\}$.

• Почти все decision-задачи алгоритмически не разрешимы!

Любое множество $L \subseteq \mathbb{N}$ это decision-задача $\Rightarrow$ всего задач $|2^{\mathbb{N}}|$ (несчётно).
Алгоритм — строка на питоне, строк счётное число ($|\mathbb{N}|$), ведь их можно занумеровать.

$$|2^{\mathbb{N}}| > |\mathbb{N}| \Rightarrow \text{ почти все задачи неразрешимы}$$

Тут можно сослаться на общую теорему Кантора ($\left|2^A\right| > |A|$),
либо вспомнить «диагонализацию Кантора» [wiki] для $|\mathbb{R}| = |2^{\mathbb{N}}|$, $|\mathbb{R}| > |\mathbb{N}|$.

Посмотрим на примеры неразрешимых задач. Примеров много[1], much enough[2], see also[3].
Можно выделить класс задач «предсказание будущего сложного процесса».
Например, «в игре жизнь [wiki] достижимо ли из состояния A состояние B?». Или не в игре...

Канонический пример — «проблема останова» (halting problem):
Дана программа, остановится ли она когда-нибудь на данном входе?

Теорема 15.1.1. Halting problem алгоритмически не разрешима.

Доказательство. От противного. Пусть есть алгоритм `terminates(code,x)`, всегда останавливающийся, и возвращающий `true` iff `code` останавливается на входе `x`. Рассмотрим программу:

```
1 def invert(code):
2   if terminates(code,code): while (true)
```

Если `invert(invert)` останавливается, то должен зависнуть, и наоборот. Противоречие. ■

Теорема Успенского-Райса [ifmo]: любое нетривиальное свойство программ неразрешимо.
«Нетривиальное» = хотя бы одна программа ему удовлетворяет, но не все программы.
Примеры: программа возвращает только простые числа; решает задачу о рюкзаке.

15.2. Основные классы

• **DTime, P, EXP (классы для decision задач)**

Def 15.2.1. $\mathrm{DTime}(f(n))$ – *множество задач распознавания, для которых* $\exists C > 0$ *и детерминированный алгоритм, работающий на всех входах не более чем* $C \cdot f(n)$*, где* n *– длина входа.*

Def 15.2.2. $\mathrm{P} = \bigcup\limits_{k>0} \mathrm{DTime}(n^k)$. *Т.е. задачи, имеющие полиномиальное решение.*

Def 15.2.3. $\mathrm{EXP} = \bigcup\limits_{k>0} \mathrm{DTime}(2^{n^k})$. *Т.е. задачи, имеющие экспоненциальное решение.*

Теорема 15.2.4. Об иерархии по времени
$\mathrm{DTime}(f(n)) \subsetneq \mathrm{DTime}(f(n)\log^2 f(n))$ для конструктивных по времени f (см. ниже).

Доказательство. Задача, которую нельзя решить за $f(n)$: завершится ли данная программа за $f(n)\log f(n)$ шагов. Подробнее на практике и в wiki, там же дана более сильная формулировка теоремы, требующая большей аккуратности при доказательстве. ■

Следствие 15.2.5. $\mathrm{P} \neq \mathrm{EXP}$ т.к. $\mathrm{P} \subseteq \mathrm{DTime}(2^n) \subsetneq \mathrm{DTime}(2^{2n}) \subseteq \mathrm{EXP}$

15.3. NP (non-deterministic polynomial)

Def 15.3.1. $\mathrm{NP} = \{L\colon \exists M$, *работающий за полином от* $|x|$, $\forall x \left((\exists y\, M(x,y) = 1) \Leftrightarrow (x \in L)\right)\}$

Неформально: NP – класс языков L: $\forall\, x \in L$, $\exists$ такая подсказка $y(x)$, что мы за полином сможем проверить, что $x \in L$. Напомним, «язык» ≡ «decision задача».
Ещё более неформально: «NP – класс задач, ответ к которым можно проверить за полином». Подсказку y так же называют свидетелем того, что x лежит в L.

• **Примеры NP-задач**
1. HAMPATH $= \{G \mid G$ – неорграф, в котором есть гамильтонов путь$\}$. Подсказка y – путь. M получает вход $x = G$, подсказку y, проверяет, что y прост, $|y| = n$ и $\forall (e \in y)\ e \in G$.
2. k-CLIQUE – проверить наличие в графе клики размером k. Подсказка y – клика.
3. IS-SORTED – отсортирован ли массив? Она даже лежит в P.

Def 15.3.2. $\mathrm{coNP} = \{L\colon \overline{L} \in \mathrm{NP}\}$
Def 15.3.3. $\mathrm{coNP} = \{L\colon \exists M$, *работающий за полином от* $|x|$, $\forall x \left((\forall y\, M(x,y) = 0) \Leftrightarrow (x \in L)\right)\}$
Def 15.3.4. $\mathrm{coNP} = \{L\colon \exists M$, *работающий за полином от* $|x|$, $\forall x \left((\exists y\, M(x,y) = 1) \Leftrightarrow (x \notin L)\right)\}$

Неформально: дополнение языка лежит в NP или «$\exists$ свидетель того, что $x \notin L$».

Пример coNP-задачи: PRIME – является ли число простым. Подсказкой является делитель. На самом деле PRIME ∈ P, но этого мы пока не умеем понимать.

Замечание 15.3.5. $\mathrm{P} \subseteq \mathrm{NP}$ (можно взять пустую подсказку, M просто решит задачу).

Замечание 15.3.6. Вопрос $\mathrm{P} = \mathrm{NP}$ или $\mathrm{P} \neq \mathrm{NP}$ остаётся открытым (wiki). Предполагают, что $\neq$.

15.4. NP-hard, NP-complete

Def 15.4.1. ∃ *полиномиальное сведение (по Карпу) задачи* A *к задаче* B*:* $(A \leqslant_P B) \Leftrightarrow$ ∃ *алгоритм* f*, работающий за полином,* $(x \in A) \Leftrightarrow (f(x) \in B)$

Замечание 15.4.2. f работает за полином $\Rightarrow |f(x)|$ полиномиально ограничена $|x|$

Def 15.4.3. ∃ *сведение по Куку задачи* A *к задаче* B*:* $(A \leqslant_C B) \Leftrightarrow \exists M$*, решающий* A*, работающий за полином, которому разрешено обращаться за* $\mathcal{O}(1)$ *к решению* B*.*

Ещё говорят «задача A сводится к задаче B».

В обоих сведениях мы решаем задачу A, используя уже готовое решение задачи B.
Т.е. доказываем, что «A не сложнее B». Различие: в первом случае решением B можно воспользоваться один раз (и инвертировать ответ нельзя), во втором случае полином раз.

Def 15.4.4. NP-hard = NPh = $\{L\colon \forall A \in \text{NP} \quad A \leqslant_P L\}$

NP-трудные задачи – класс задач, которые не проще любой задачи из класса NP.

Def 15.4.5. NP-complete = NPc = NPh ∩ NP

NP-полные задачи – самые сложные задачи в классе NP.
Если мы решим хотя бы одну из NPc за полином, то решим все из NP за полином.
Хорошая новость: все NP-полные по определению сводятся друг к другу за полином.

Замечание 15.4.6. Когда хотите выразить мысль, что задача трудная в смысле решения за полином (например, поиск гамильтонова пути), **неверно** говорить «это NP задача» (любая из P тоже в NP) и странно говорить «задача NP-полна» (в этом случае вы имеете в виду сразу, что и трудная, и в NP). Логично сказать «задача NP-трудна».

Lm 15.4.7. $A \leqslant_P B, B \in \text{P} \Rightarrow A \in P$

Доказательство. Сведение f работает за n^s, B решается за $n^t \Rightarrow A$ решается за n^{st}. ■

Lm 15.4.8. $A \leqslant_P B, A \in \text{NPh} \Rightarrow B \in \text{NPh}$

Доказательство. $\forall L \in \text{NP}$ (∃$f\colon L$ сводится к A функцией $f(x)$) ∧ ($A \leqslant_P B$ функцией $g(x)$) ⇒ L сводится к B функцией $g(f(x))$ (за полином). ■

15.5. NH, NP-полные задачи существуют!

Def 15.5.1. BH = BOUNDED-HALTING*: вход* $x = \langle \underbrace{11\ldots1}_{k}, M, x\rangle$*, проверить,* ∃ *ли такой* y*:* $M(x, y)$ *остановится за* k *шагов и вернёт* `true`*.*

BH ∈ NP. Подсказка – такой y. Алгоритм – моделирование k шагов M за $\mathcal{O}(\text{poly}(k))$.

Важно, что если бы число k было записано, используя $\log_2 k$ бит, моделирование работало бы за экспоненту от длины входа, и нельзя было бы сказать «задача лежит в NP».

Докажем, что BH ∈ NPc. На экзамене доказательство можно сформулировать в одно предложение, здесь же оно для понимания расписано максимально подробно.

Теорема 15.5.2. BH = BOUNDED-HALTING ∈ NPc

Доказательство. NPc = NP ∩ NPh. Мы уже показали Def 15.5.1, что BH лежит в NP.
Теперь покажем, что BH ∈ NPh. Для этого нужно взять $L \in$ NP и свести его к BH.
Пусть $L \in$ NP $\Rightarrow \exists$ полиномиальный M, проверяющий подсказки для L.
Полиномиальный $\Leftrightarrow \exists P(n)$, ограничивающий время работы M. $(x \in L) \Leftrightarrow \exists y\ M(x, y) = 1$.
Программа M всегда отрабатывает за $P(|x|)$, если запустить её с будильником $P(|x|)$, она не поменяется. Рассмотрим $f(x) = \langle \underbrace{11\ldots1}_{P(|x|)}, M, x \rangle$.

Получили полиномиальное сведение: $(x \in L) \Leftrightarrow (\exists y\ M(x, y) = 1) \Leftrightarrow (f(x) \in$ BH$)$.
Заметьте, зная L, мы не умеем предъявить ни M, ни f, мы лишь знаем, что $\exists M, f$. ■

15.6. Сведения, новые NP-полные задачи

Началось всё с того, что в 1972-м Карп опубликовал список из 21 полной задачи, и дерево сведений. Кстати, в его работе [pdf] все сведения крайне лаконичны. Итак, приступим:

Чтобы доказать, что $B \in$ NPh, нужно взять любую $A \in$ NPh и свести A к B полиномиально. Пока такая задача A у нас одна – BH. На самом деле их очень много.

Чтобы доказать, что $B \in$ NPc, нужно ещё не забыть проверить, что $B \in$ NP.
Во всех теоремах ниже эта проверка очевидна, мы проведём её только в доказательстве первой.

• BH → CIRCUIT-SAT → SAT → 3-SAT → k-INDEPENDENT → k-CLIQUE

Def 15.6.1. CIRCUIT-SAT. *Дана схема, состоящая из входов, выхода, гейтов* `AND`, `OR`, `NOT`. *Проверить, существует ли набор значений на входах, дающий* `true` *на выходе.*

Теорема 15.6.2. CIRCUIT-SAT ∈ NPc

Доказательство. Подсказка – набор значений на входах ⇒ CIRCUIT-SAT ∈ NP.
Сводим BH к CIRCUIT-SAT ⇒ нам даны программа M, время выполнения t, вход x.
За время t программа обратится не более чем к t ячейкам памяти.
Обозначим за $s_{i,j}$ состояние `true`/`false` j-й ячейки памяти в момент времени i.
$s_{0,j}$ – вход, $s_{t,output}$ – выход, $\forall i \in [1, t]$ $s_{i,j}$ зависит от $\mathcal{O}(1)$ переменных $(i-1)$-го слоя.
Сейчас значение s_{ij} – произвольная булева формула f_{ij} от $\mathcal{O}(1)$ переменных из слоя s_{i-1}.
Перепишем f_{ij} в КНФ-форме, чтобы получить гейты вида `AND`, `OR`, `NOT`. Получили $\mathcal{O}(t^2)$ булевых гейтов ⇒ по (M, t, x) за полином построили вход к CIRCUIT-SAT. ■

Теорема 15.6.3. SAT ∈ NPc

Доказательство. В разборе практики смотрите сведение из CIRCUIT-SAT. ■

Теорема 15.6.4. 3-SAT ∈ NPc

Доказательство. Сводим SAT к 3-SAT. Пусть есть клоз $(x_1 \vee x_2 \vee \cdots \vee x_n), n \geqslant 4$.
Введём новую переменную w и заменим его на $(x_1 \vee x_2 \vee w) \wedge (x_3 \vee \cdots \vee x_n \vee \overline{w})$. ■

Def 15.6.5. k-INDEPENDENT*: есть ли в графе независимое подмножество размера k?*

Теорема 15.6.6. k-INDEPENDENT $\in$ NPc

Доказательство. Наша формула – m клозов $(l_{i1} \vee l_{i2} \vee l_{i3})$, где l_{ij} – литералы.
Построим граф из ровно $3m$ вершин – l_{ij}. $\forall i$ добавим треугольник (l_{i1}, l_{i2}, l_{i3}) (итого $3m$ рёбер).
В любое независимое множество входит максимум одна вершина из каждого треугольника.
$\forall k = 1..n$ соединим все вершины $l_{ij} = x_k$ со всеми вершинами $l_{ij} = \overline{x_k}$.
Теперь $\forall k = 1..n$ в независимое множество нельзя одновременно включить x_k и $\overline{x_k}$.
Итог: $\exists$ независимое размера $m \Leftrightarrow$ у 3-SAT было решение. ■

Def 15.6.7. k-CLIQUE *задача: есть ли в графе полное подмножество вершин размера k?*

Теорема 15.6.8. k-CLIQUE $\in$ NPc

Доказательство. Есть простое двустороннее сведение k-CLIQUE $\leftrightarrow$ k-INDEPENDENT.
c_{ij} – есть ли ребро между i и j вершинами. Создадим новый граф: $c'_{ij} = \overline{c_{ij}} \wedge (i \neq j)$. ■

15.7. Задачи поиска

Def 15.7.1. $\overline{\text{NP}}$, $\overline{\text{NPc}}$, $\overline{\text{NPh}}$ – *аналогичные классы для задач поиска подсказки.*

• Сведение задач минимизации, максимизации к decision задачам

Пусть мы умеем проверять, есть ли в графе клика размера k.
Чтобы найти размер максимальной клики, достаточно применить бинпоиск по ответу.
Это общая техника, применимая для максимизации/минимизации численной характеристики.

• Сведение search задач к decision задачам

Последовательно фиксируются биты (части) подсказки y.

Пример: выполняющий набор для SAT.
Пусть $M(\varphi)$ проверяет выполнимость формулы φ (M – решение SAT).
$\varphi[x_i{=}e]$ – формула, полученная из φ подстановкой x_i`:=`e. Индукция:

```
while n > 0:
    if M(φ[x_n=0]) = 1 then r_n = 0 else r_n = 1
    φ ← φ[x_n=r_n]; n--
```

Пример: k-INDEPENDENT.
Взять или выкинуть первую вершину?
`if` $(G \setminus \{1\}) \in k$-INDEPENDENT `then` G `\=` $\{1\}$
`else k--`, ans `∪=` $\{1\}$, G `\=` $\{N(1)\}$ (выкинуть соседей 1-й вершины)

• Решение NP-полных задач

Пусть вам дана NP-полная задача. С одной стороны плохо – для неё нет быстрого решения. С другой стороны её можно свести к SAT, для которого несколько десятилетий успешно оптимизируются специальные SAT-solvers. Например, вы уже можете решать k-CLIQUE, построив вход к задаче SAT и скормить его `python3` пакету pycosat.
А ещё можно принять участие в соревновании.

15.8. Гипотезы

Гипотеза 15.8.1. $P \neq NP$

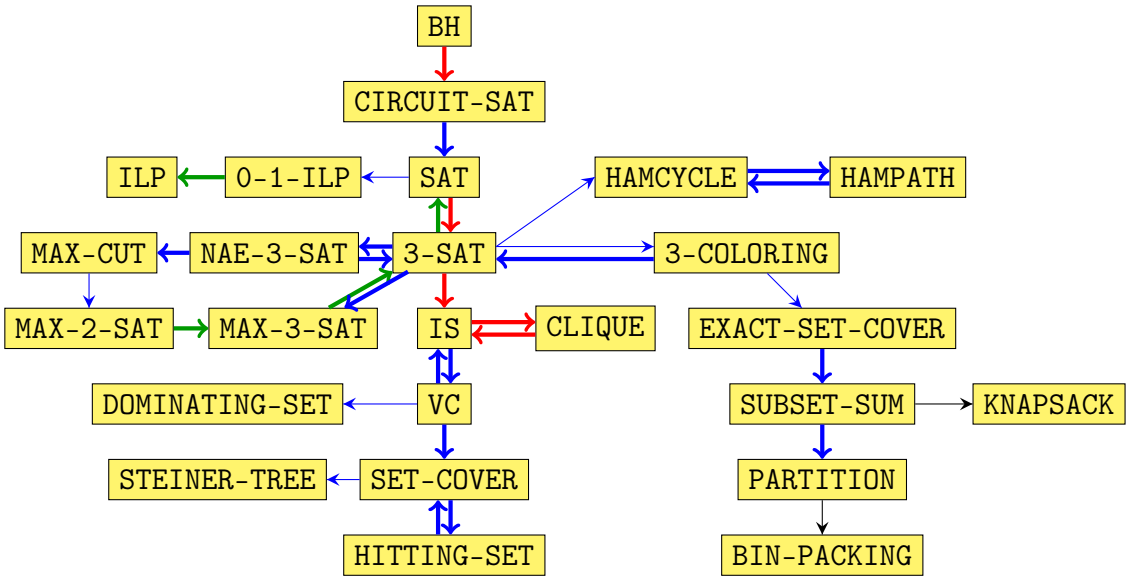
Гипотеза 15.8.2. ETH (exponential time hypothesis): $\nexists$ решения за $2^{o(n)}$ для 3-SAT

Гипотеза 15.8.3. SETH (strong ETH): $\forall \varepsilon\, \exists k$: $\nexists$ решения за $(2-\varepsilon)^n$ для k-SAT

Lm 15.8.4. SETH $\Rightarrow$ ETH $\Rightarrow$ P $\neq$ NP

15.9. Дерево сведений

Легенда:
- **Красное** – было на лекции.
- **Синее** – было в задачах практики/дз, нужно знать на экзамене.
- Бледносинее – было в задачах практики/дз, не будет на экзамене.
- **Зелёное** – очевидное вложение.
- Чёрное – будет доказано и использовано в будущем.



Лекция #16: (*) Дополнительная сложность

29 января 2021

16.1. (*) Алгоритм Левина

Есть универсальный алгоритм, который для любой задачи из $\overline{\text{NP}}$ работает за время близкое к оптимальному. Рассмотрим любую задачу поиска $L \in \overline{\text{NP}}$.

Пусть $M(x, y)$ – полиномиальный чекер для L. Рассмотрим любой корректный алгоритм A, решающий L. $\exists S_A$ – программа на питоне, имплементирующая A. S_A – строка, то есть, число в 256-ичной системе счисления. Рассмотрим соответствующее ей число N_A.
N_A – константа. Пусть $t(|x|)$ – время работы A.

Будем перебирать все программы и время работы.

```
1 solve(x):
2     for (T = 1;; T *= 2) // время работы T
3         for (P = 1; 2^P <= T; P++) // программа P
4             y = call(P, T, x) // запускаем T шагов программы P на входе x
5             if M(x, y)
6                 return y
```

Если решений нет, наш алгоритм успешно зависнет. Далее обозначим $|x| = n$.
Если решение $\exists$, как только $T \geqslant \max(2^{N_A}, t(n)) = F = \Theta(t(n))$, мы найдём решение.

Оценим время работы.
Одна итерация внешнего `for` работает за $\mathcal{O}(M \cdot T \log T)$. Суммарное время:
$\sum_T (M \cdot T \log T) = \Theta(M \cdot F \log F) = \Theta(M \cdot t(n) \log t(n))$ (помним, что 2^{N_A} – просто константа).

Итого, мы получили алгоритм, который для любой $\overline{\text{NP}}$ задачи при существовании решения находит его за время $\Theta(M \cdot t \log t)$, где t – время работы оптимального алгоритма.
Например, для гамильтонова пути это будет $\Theta(V \cdot t \log t)$.

Полученный алгоритм называется *алгоритмом Левина*.

Следствие 16.1.1. Если P = NP, то алгоритм Левина решает все $\overline{\text{NP}}$ задачи за полином.

16.2. (*) Расщепление

Мы будем решать задачу поиска максимального независимого множества (IS). Но, конечно, все наши идеи подходят также для максимальной клики и минимального вершинного покрытия.

Итак, простейший алгоритм расщепления:
берём вершину v и или берём её в независимое множества (первая ветка рекурсии), или не берём (вторая ветка). Берём $\Rightarrow$ удалим и v и соседей. Не берём $\Rightarrow$ удалим v.

Время работы $T(n) = T(n - deg[v] - 1) + T(n-1) \Rightarrow$ выгодно взять v: $deg[v] = \max$.

Осталось заметить
1. Если $\exists$ вершина степени 1 её выгодно жадно взять в IS.
2. Если $\exists$ вершина v степени 2. Если мы берём ровно одного из её соседей, можно заменить его на v $\Rightarrow$ есть только два разумных варианта – или взять v, или её обоих соседей $\Rightarrow$ модифицируем наш граф: удалим v, смерджим её соседей a, b в одну вершину ab. Вариант «брать v» перешёл в «не брать ab», вариант «брать a, b» перешёл в «брать ab».

Ответ уменьшился ровно на 1, а число вершин на 2. При оценке времени нам важно будет, что как только появляется вершина степени 2, сразу $n \to n-2$.

Остался случай $\forall v\ deg[v] \geqslant 3$. Уже получили асимптотику $\boxed{T(n) = T(n-1) + T(n-4) \leqslant 1.3803^n}$.
В дальнейшем анализе рекурренты $T(n-a_1) + T(n-a_2) + \dots$ будем обозначать $[a_1, a_2, \dots]$.

Замечание 16.2.1. На самом деле все асимптотики α^n нужно записывать как $\mathcal{O}^*(\alpha^n)$, так как на каждом шаге присутствует ещё полиномиальное время на выбор нужного случая и, возможно, модификацию графа. Сейчас мы сосредоточены на минимизации α, про полином не думаем.

• **Разбор случаев** $deg[v] \geqslant 3$
3a. $\forall v\ deg[v] = 3 \Rightarrow$ после расщепления сразу попадаем в случай (2): $\exists v\ deg[v] = 2$,
итого $T(n) \leqslant T(n-1-2) + T(n-1-3-2) = [3, 6] \leqslant 1.174^n$. На самом деле появится сразу много вершин степени 2, оценку можно сильно улучшить, нам хватит такой.
3b. $\exists v\ deg[v] = 3 \Rightarrow$ по непрерывности $\exists u\ deg[u] \geqslant 4 \wedge$ у u есть сосед степени 3. Расщепляемся по u: $T(n) \leqslant T(n-1-2) + T(n-1-deg[u]) = [3, 5] \leqslant 1.194^n$. Из случаев (3b) и (3a) хуже (3b) $\Rightarrow$ если появляются вершины степени 3, можно считать, что это именно (3b).
4a. $\forall v\ deg[v] = 4 \Rightarrow$ после расщепления сразу попадаем в случай (3b): $\exists v\ deg[v] = 3$, итого $T_{4a}(n) \leqslant T_{3b}(n-1) + T_{3b}(n-5) = [1+3, 1+5, 5+3, 5+5] = [4, 6, 8, 10] \leqslant 1.239^n$.
4b. $\exists v\ deg[v] = 4 \Rightarrow$ по непрерывности $\exists u\ deg[u] \geqslant 5 \wedge$ у u есть сосед степени 4. Расщепляемся по u: $T_{4b}(n) \leqslant T_{3b}(n-1) + T(n-6) = [4, 6, 6] \leqslant 1.2335^n$.
5a. $\forall v\ deg[v] = 5 \Rightarrow$ после расщепления сразу попадаем в случай (4b): $\exists v\ deg[v] = 4$, итого $T(n) \leqslant T_{4b}(n-1) + T_{4b}(n-6) = [5, 7, 7, 10, 12, 12] \leqslant 1.2487^n$.
Поскольку $T_{4a} \geqslant T_{4b}$, важно, что именно 4b, это точно так, если $n > 1+5+25$, то есть, не $\mathcal{O}(1)$.
5b. $\exists v\ deg[v] = 5 \Rightarrow$ по непрерывности $\exists u\ deg[u] \geqslant 6 \wedge$ у u есть сосед степени 5. Расщепляемся по u: $T(n) \leqslant T_{4b}(n-1) + T(n-7) = [5, 7, 7, 7] \leqslant 1.2413^n$.
6a. $\forall v\ deg[v] = 6 \Rightarrow$ после расщепления сразу попадаем в случай (5b): $\exists v\ deg[v] = 5$, итого $T(n) \leqslant T_{5b}(n-1) + T_{5b}(n-7) = [6, 8, 8, 8, 12, 14, 14, 14] \leqslant 1.24499^n$.
7. $\exists v\ deg[v] \geqslant 7 \Rightarrow T(n) \leqslant T(n-1) + T(n-8) = [1, 8] \leqslant 1.23206^n$

Итого время работы $= \max\{T_{3a}, T_{3b}, T_{4a}, T_{4b}, T_{5a}, T_{5b}, T_{6a}, T_7\} = T_{5a} = [5, 7, 7, 10, 12, 12] \leqslant \boxed{1.2487^n}$
P.S. Заметим, что $[1, 7] \approx 1.2554^n \Rightarrow$ все части анализа необходимы.

• **Алгоритм**
Анализ сложен. А алгоритм прост. Выбираем ребро (a, b): $deg[a] = \min, deg[b] = \max$.
В первую очередь минимизируем $deg[a]$, при равных максимизируем $deg[b]$).
Если $deg[a] \leqslant 2$, применяем жадность, иначе расщепляемся по b.

Наша оценка 1.2487^n не точна, её можно улучшать и улучшать. Например, на самом деле $T_{5a} \approx \max(T_{4b}, T_{4a}, T_{3b}, T_{4a})$, так как в обеих ветках рекурсии мы очень не скоро получим что-то кроме этих четырёх случаев. Но получим, в (2) степени увеличиваются, $deg_{ab} = deg_a + deg_b - 2$.

16.3. (*) Оценка с использованием весов

Повторим анализ. Сперва более простую версию.
1. Если $\exists$ вершина степени 1 её выгодно жадно взять в IS.
2. $\forall v\ deg[v] = 2 \Rightarrow$ пути и циклы, задача решается жадно.
3. $\exists v\ deg[v] \geqslant 3 \Rightarrow T(n) = T(n-1) + T(n-deg-1) \leqslant T(n) = T(n-1) + T(n-4) \leqslant 1.3803^n$

Новая идея: сопоставим вершинам разной степени разные веса.

$w_1 = 0, w_2 = 0.5, w_3 = 1, w_4 = 1, \ldots$ (веса больших степеней тоже 1).
Для вершин степени 1 мы по-прежнему планируем запускать жадность.

Заметим:
a) $k = \sum_v w_{deg[v]}[v] \leqslant n$
b) $k = 0 \Rightarrow$ граф пуст, ответ известен
c) $w_i \leqslant w_{i+1} \Rightarrow$ после любых расщеплений $k \searrow$

Заменим число вершин n на суммарный вес вершин k. $k \leqslant n \Rightarrow T(k) \leqslant T(n)$.

3. $\max_v deg[v] = 3 \Rightarrow$ расщепляемся по 3-ке. Степень её соседей уменьшится, их вес уменьшится на 0.5 и в случае $3 \to 2$, и в случае $2 \to 1$. В ветке с удалением соседей то же произойдёт с их соседями. Итого $T(k) \leqslant T(k-1-3{\cdot}0.5) + T(k-1-5{\cdot}0.5) = [2.5, 3.5] \leqslant 1.263^k$.
4. $\max_v deg[v] = 4 \Rightarrow$ расщепляемся по 4-ке. Разберём 2 случая.
Все соседи 3-ки: $T(k) \leqslant T(k-1-4{\cdot}0.5) + T(k-1-4{\cdot}0.5) = [3, 3] \leqslant 1.25993^k$.
Все соседи 4-ки: $T(k) \leqslant T(k-1) + T(k-1-4) = [1, 5] \leqslant 1.3248^k$.
Случаев больше, их разбор здесь не представлен, худшим из них будет «все 4-ки».

• Применяем идею весов по полной.

1. Если $\exists$ вершина степени 1 её выгодно жадно взять в IS.
2a. Если $\exists$ две смежных вершины a, b степени 2, заменим $u - a - b - v$ на $u - v$.
2b. Если $\exists$ вершина степени 2, удалим её, стянем её соседей.
Если их степени были 3, 3, новая вершина имеет степень $3{+}3{-}2 = 4$.

Подберём теперь веса. Самые простые: $w_1 = w_2 = 0, w_3 = 0.8, w_4 = 1, w_5 = 1, \ldots$
Заметим, что попадая в состояние $\exists v\ deg[v] \leqslant 2$, мы сразу уменьшим k хотя бы на $2w_3 - w_4 = 0.6$.

3a. $\forall v\ deg[v] = 3 \Rightarrow T(k) \leqslant T_2(k - 4 \cdot w_3) + T_2(k - 6 \cdot w_3) = [3.2 + 0.6, 4.8 + 0.6] = [3.8, 5.4] \leqslant 1.165^k$.
3b. $\exists v\ deg[v] = 3$ и максимальный сосед тройки это 4. Расщепляемся по соседу. Случаи.
Соседи 3333: ...
Соседи 3444: ...
3c. $\exists v\ deg[v] = 3$ и максимальный сосед $\geqslant 5$. Расщепляемся по соседу. Случаи.
Соседи 33333: ...
Соседи 34444: ...
Соседи 35555: ...
4. ...

Как вы понимаете, разбор случаев весьма трудоёмкий. Может быть, его можно как-то запрограммировать? =)

Ссылки: [wiki] [Robson'2001, 1.1888^n]

Кстати, $1.1888^n \approx 2^{n/4}$, то есть, для $n \approx 100$ алгоритм точно не плох.

Лекция #17: Рандомизированные алгоритмы

17/18-я пара, 2024/25

17.1. Случайные числа в C++. Немного про rand().

Начиная с C++11 для генерации случайных чисел принято использовать вихрь Мерсенна:

```
1 mt19937 gen;
2 for (int i = 0; i < n; i++) // n целых чисел из [0, M)
3     a[i] = gen() % M; // gen() даст 32-битное целое случайное число.
```

Более подробный пример про случайные числа в C++ сможно посмотреть здесь.

Обратите внимание, что функциями `rand`, `srand`, `random_shuffle` уже давно не пользуются. Одна из причин: `rand()` возвращает псевдослучайное число от `0` до `RAND_MAX`, под Windows `RAND_MAX` $= 2^{15} - 1 = 32767$, что слишком мало, а в Linux `RAND_MAX` $= 2^{31} - 1$, что убивает потенциальную кроссплотформенность.

17.2. Определения: RP, coRP, ZPP

Рандомизированными называют алгоритмы, использующие случайные биты.

Первый тип алгоритмов: «решающие decision задачи, работающие всегда за полином, ошибающиеся в одну сторону». Строго это можно записать так:

Def 17.2.1. $\mathrm{RP} = \{L\colon \exists M \in \mathrm{PTime} \begin{cases} x \notin L \Rightarrow M(x,y) = 0 \\ x \in L \Rightarrow \Pr_y[M(x,y) = 1] \geqslant \frac{1}{2} \end{cases}\}$

x – вход, y – подсказка из случайных бит. Расшифровка: RP = randomized polynomial time.
То есть, если $x \notin L$, M не ошибается, иначе работает корректно с вероятностью хотя бы $\frac{1}{2}$.
Если для какого-то y алгоритм M вернул 1, то это точно правильный ответ, $x \in L$.

Def 17.2.2. $\mathrm{coRP} = \{L\colon \exists M \in \mathrm{PTime} \begin{cases} x \in L \Rightarrow M(x,y) = 1 \\ x \notin L \Rightarrow \Pr_y[M(x,y) = 0] \geqslant \frac{1}{2} \end{cases}\}$

Если для какого-то y алгоритм M вернул 0, то это точно правильный ответ, $x \notin L$.

• **Сравнение классов** NP, RP

Если ответ 0, оба алгоритма для $\forall$ подсказки выдадут 0. Если ответ 1, для NP-алгоритма $\exists$ хотя бы одна подсказка, а RP-алгоритм должен корректно работать хотя бы на половине подсказок.

• **Понижение ошибки**

Конечно, алгоритм, ошибающийся с вероятностью $\frac{1}{2}$ никому не нужен.

Lm 17.2.3. Пусть M – RP-алгоритм, ошибающийся с вероятностью p.
Запустим его k раз, если хотя бы раз вернул 1, вернём 1. Получили алгоритм с ошибкой p^k.

Например, если повторить 100 раз, получится вероятность ошибки $2^{-100} \approx 0$.

Если есть алгоритм, *корректно* работающий с близкой к нулю вероятностью p, ошибающийся с вероятностью $1 - p$, то повторив его $\frac{1}{p}$ раз, получим вероятность ошибки $(1-p)^{1/p} \leqslant e^{-1}$.

• ZPP (zero-error probabilistic polynomial time)

ZPP – класс задач, для которых есть никогда не ошибающийся вероятностный алгоритм с полиномиальным матожиданием временем работы.

Def 17.2.4. $\text{ZPP} = \{L\colon \exists P(n),\ M \textit{ такие, что } E_y[Time(M(x,y))] \leqslant P(|x|)\}$

> Про алгоритмы и для RP, и для ZPP говорят «вероятностные/рандомизированные».
> Чтобы подчеркнуть принадлежность классу, уточняют ошибку.
> RP-алгоритм обладает односторонней ошибкой. ZPP-алгоритм работает без ошибки.

Мы определили основные классы только для задач распознавания, только для полиномиального времени. Можно определить аналогичные классы для задач поиска и для любого времени.

Важно помнить, что в RP, coRP, ZPP алгоритмы работают на *всех* тестах.
Например, вероятность ошибки в RP *для любого теста* не более $\frac{1}{2}$.

17.3. Примеры

Все наши примеры про поиск и не «за полином», а гораздо быстрее.

• Часто встречающееся число

Поиск числа, которое встречается в массиве больше половины раз.
У решения этой задачи есть две версии.
ZPP-версия: «*мы уверены, что такой элемент есть, пробуем случайные, пока не попадём*».
RP-версия: «*хотим определить, есть ли такой элемент в массиве, делаем одну пробу*».

• Поиск квадратичного невычета

Для любого простого p ровно $\frac{p-1}{2}$ ненулевых остатков обладают свойством $a^{\frac{p-1}{2}} \equiv -1 \bmod p$.
Такие a называются «квадратичными невычетами».

Задача: дано p, найти невычет.
Алгоритм: пока не найдём, берём случайное a, проверяем.
Время одной проверки — возведение в степень ($\log p$ для $p \leqslant 2^w$).
Сколько в среднем понадобиться проверок? $E = 1 + \frac{1}{2}E \Rightarrow E = 2$.
Итого ZPP-версия работает за $\mathcal{O}(\log p)$.

17.4. Проверка на простоту

• Тест Ферма

Малая теорема Ферма: $\forall p \in \mathbb{P}\ \forall a \in [1..p{-}1]\ \ a^{p-1} \equiv 1 \bmod p$.
Чтобы проверить простоту p, можно взять случайное a и проверить $a^{p-1} \equiv 1 \bmod p$

Lm 17.4.1. $\exists a\colon a^{p-1} \not\equiv 1 \bmod p \Rightarrow$ таких a хотя бы $\frac{p-1}{2}$.

Доказательство. Пусть b такое, что $b^{p-1} \equiv 1 \bmod p \Rightarrow (ab)^{p-1} = a^{p-1}\,b^{p-1} \not\equiv 1 \bmod p$. ■

Мы показали, что если тест вообще работает, то с вероятностью хотя бы $\frac{1}{2}$.
К сожалению, есть составные числа, для которых тест вообще не работает – числа Кармайкла.

Утверждение 17.4.2. У числа Кармайкла a есть простой делитель не более $\sqrt[3]{a}$.

Получаем следующий всегда корректный вероятностный алгоритм:
проверим возможные делители от 2 до $\sqrt[3]{a}$, далее тест Ферма.

• **Тест Миллера-Рабина**

```
1  bool isPrime(int p):
2      // p − 1 = 2^s t,  t -- нечётно
3      if (p < 2) return 0; // 1 и 0 не простые
4      p → (s, t)
5      a = randInt(2, p-1) // [2, p-1]
6      g = pow(a, t, p) // возвели в степень по модулю
7      for (int i = 0; i < s; i++) {
8          if (g == 1) return 1;
9          if (g % p != -1 && g * g % p == 1) return 0;
10         g = g * g % p;
11     }
12     return g == 1; // тест Ферма
```

В строке 8 мы проверяем, что нет корней из 1 кроме 1 и -1.
Время работы $\mathcal{O}(\log p)$ операций «умножение по модулю».

Утверждение 17.4.3. $\forall a$ вероятность ошибки не более $\frac{1}{4}$.

• **Изученные алгоритмы**

(*) k-я порядковая статистика учит нас:
«*вместо того, чтобы выбирать медиану, достаточно взять случайный элемент*».
(*) 3-LIST-COLORING: вычеркнуть из каждого клоза случайный элемент, получить 2-SAT. Вероятность успеха этого алгоритма $\frac{2^n}{3^n}$, чтобы получить вероятность $\frac{1}{2}$, нужно было бы повторить его $1.5^n \ln 2$ раз, поэтому к RP-алгоритмам он не относится.
Ещё про него полезно понимать «*работает на 2^n подсказках из 3^n возможных*».

• (*) **Проверка** $AB = C$

Даны три матрицы $n \times n$, нужно за $\mathcal{O}(n^2)$ проверить равенство $AB = C$. Все вычисления в $\mathbb{F}_2$. Вероятностный алгоритм генерирует случайный вектор x и проверяет $A(Bx) = Cx$.

Lm 17.4.4. Вероятность ошибки не более $\frac{1}{2}$.

Доказательство. Пусть $AB \neq C \Rightarrow D = AB - C \neq 0$.
Посчитаем $\Pr_x\left[A(Bx) = Cx\right] = \Pr_x\left[Dx = 0\right]$. $\exists i, j\colon D_{i,j} \neq 0$. Зафиксируем произвольный x. $(Dx)_i = D_{i,0}x_0 + \cdots + D_{i,j}x_j + \cdots + D_{i,n-1}x_{n-1}$. $D_{i,j} \neq 0 \Rightarrow$ меняя значение x_j, меняем значение $(Dx)_i \Rightarrow \Pr_x\left[(Dx)_i = 0\right] = \frac{1}{2} \Rightarrow \Pr_x\left[Dx = 0\right] \leqslant \frac{1}{2}$. ■

17.5. ZPP = RP ∩ coRP

Lm 17.5.1. Неравенство Маркова: $x \geqslant 0 \Rightarrow \forall a\ \Pr[x > aE(x)] < \frac{1}{a}$

Доказательство. Пусть $\Pr[x > aE(x)] \geqslant \frac{1}{a}$, тогда матожидание должно быть больше чем произведение $aE(x)$ и $\frac{1}{a}$ (так как «есть хотя бы $\frac{1}{a}$ x-ов со значением $aE(x)$). Формально: $\Rightarrow E(x) > \big(aE(x)\big) \cdot \frac{1}{a} + 0 \cdot (1 - \frac{1}{a}) = E(x)$?!? ■

Следствие 17.5.2. $x \geqslant 0 \Rightarrow \forall a\ \Pr[x \leqslant aE(x)] \geqslant \frac{1}{a}$

Теорема 17.5.3. ZPP = RP ∩ coRP

Доказательство. Пусть $L \in$ ZPP $\Rightarrow \exists$ алгоритм M, работающий в среднем за $P(n)$ $\overset{Cons\ 17.5.2}{\Rightarrow}$

$\Pr[Time(M) \leqslant 2P(n)] \geqslant \frac{1}{2} \Rightarrow$ запустим M с будильником на $2P(n)$ операций. Если алгоритм завершился до будильника, он даст верный ответ, иначе вернём $0 \Rightarrow$ RP или $1 \Rightarrow$ coRP.

Пусть $L \in \text{RP} \cap \text{coRP} \Rightarrow \exists M_1(\text{RP}), M_2(\text{coRP})$, работающие за полином, ошибающиеся в разные стороны, которые можно запускать по очереди $(M_1 + M_2)$. Пусть $x \notin L \Rightarrow M_1(x) = 0$, $\Pr[M_2(x) = 0] \geqslant \frac{1}{2}$. Сколько раз нужно в среднем запустить $M_1 + M_2$? $E \geqslant 1 + \frac{1}{2}E \geqslant 2$ (всегда запустим 1 раз, затем с вероятностью $\frac{1}{2}$ получим $M_2(x) = 1$ и повторим процесс с начала). ■

Lm 17.5.4. $\text{P} \subseteq \text{coRP} \cap \text{RP} = \text{ZPP} \subseteq \text{RP} \subseteq \text{NP}$

При этом строгие ли вложения неизвестно (про все три вложения).
Зато известна масса задач, для которых есть простое RP-решение, но неизвестно P-решение.

17.6. Двусторонняя ошибка, класс BPP

Def 17.6.1. $\text{BPP} = \{L\colon \exists M \in \text{PTime} \begin{cases} x \notin L \Rightarrow \Pr_y[M(x,y) = 0] \geqslant \alpha \\ x \in L \Rightarrow \Pr_y[M(x,y) = 1] \geqslant \alpha \end{cases} \}$, *где* $\alpha = \frac{2}{3}$.

Другими словами алгоритм всегда даёт корректный ответ с вероятностью хотя бы $\frac{2}{3}$.
Для BPP тоже можно понижать ошибку: запустим n раз, выберем ответ, который чаще встречается (majority). На самом деле α в определении можно брать сколь угодно близким к $\frac{1}{2}$.

Lm 17.6.2. О понижении ошибки. Пусть $\beta > 0, \alpha = \frac{1}{2} + \varepsilon,\ \varepsilon > 0 \Rightarrow \exists n = poly(\frac{1}{\varepsilon})$ такое, что повторив алгоритм n раз, и вернув majority, мы получим вероятность ошибки не более β.

(*) Доказательство. Пусть $x \in L$, мы повторили алгоритм $n = 2k$ раз.
Если получили 1 хотя бы $k{+}1$ раз, вернём 1, иначе вернём 0.
Вероятность ошибки $\texttt{Err} = \sum\limits_{i=0}^{i=k} p_i$, где p_i – вероятность того, что алгоритм ровно i раз вернул 1.
Выпишем в явном виде формулу для p_i, при этом удобно отсчитывать i от середины: $i = k - j$.

$$p_i = p_{k-j} = \tbinom{n}{i}(\tfrac{1}{2} + \varepsilon)^{k-j}(\tfrac{1}{2} - \varepsilon)^{k+j} = \tbinom{n}{i}(\tfrac{1}{4} - \varepsilon^2)^k(\tfrac{1/2-\varepsilon}{1/2+\varepsilon})^j \leqslant \tbinom{n}{i}(\tfrac{1}{4} - \varepsilon^2)^k$$

$$\texttt{Err} = \sum_{i=0}^{i=k} p_i \leqslant \sum_{i=0}^{i=k} \binom{n}{i}(\tfrac{1}{4} - \varepsilon^2)^k = \tfrac{1}{2}2^n(\tfrac{1}{4} - \varepsilon^2)^k = \tfrac{1}{2}(1 - 4\varepsilon^2)^k$$

Хотим, чтобы ошибка была не больше β, для этого возьмём $k = (1/4\varepsilon^2) \cdot \ln\frac{1}{\beta} = poly(\frac{1}{\varepsilon})$. ■

Замечание 17.6.3. Люди не знают, кто больше BPP или NP. Из $\text{NP} \subset \text{BPP}$ следует плохое, в это люди не верят. [[zoo]] [[wiki]]

Лекция #18: Рандомизированные алгоритмы

17/18-я пара, 2024/25

18.1. Как ещё можно использовать случайные числа?

• Идеальное кодирование

Алиса хочет передать Бобу некое целое число x от 0 до $m-1$. У них есть общий ключ r от 0 до $m-1$, сгенерированный равномерным случайным распределением. Тогда Алиса передаст по открытому каналу $y = (x + r) \bmod m$. Знание **одного** такого числа не даст злоумышленнику ровно никакой информации. Боб восстановит $x = (y - r) \bmod m$.

• Вычисления без разглашения

В некой компании сотрудники стесняются говорить друг другу, какая у кого зарплата.
Зарплату i-го обозначим x_i. Сотрудники очень хотят посчитать среднюю зарплату $\Leftrightarrow$ посчитать сумму $x_1 + \cdots + x_n$. Для этого они предполагают, что сумма меньше $m = 10^{18}$, и пользуются следующим алгоритмом:

0. Первый сотрудник генерирует случайное число $r \in [0, m)$.
1. Первый сотрудник передаёт второму число $(r + x_1) \bmod m$.
2. Второй сотрудник передаёт третьему число $(r + x_1 + x_2) \bmod m$.
3. ...

n. Последний сотрудник передаёт первому $(r + x_1 + x_2 + \cdots + x_n) \bmod m$.

$*$. Первый, как единственный знающий r, вычитает его и говорит всем ответ.

• Приближения

На практике будут разобраны $\frac{1}{2}$-OPT приближения для MAX-3-SAT и VERTEX-COVER.

18.2. Парадокс дней рождений. Факторизация: метод Полларда

«В классе 27 человек $\Rightarrow$ с большой вероятностью у каких-то двух день рождения в один день»
Пусть p_k – вероятность, что среди k случайных чисел от 1 до n все различны. Оценим p_k снизу.

Lm 18.2.1. $1 - p_k \leqslant \frac{k(k-1)}{2n}$

Доказательство. $f = \#\{(i,j)\colon x_i{=}x_j\}$, $1 - p_k = \Pr[f{>}0] \leqslant E_k[f] = \frac{k(k-1)}{2} \cdot \Pr[x_i{=}x_j] = \frac{k(k-1)}{2}\frac{1}{n}$ ■

Следствие 18.2.2. При $k = o(\sqrt{n})$ получаем $1 - p_k = o(1) \Rightarrow$ с вероятностью ≈ 1 все различны.
Теперь оценим p_k при $k = \sqrt{n}$.

Lm 18.2.3. $k = \sqrt{n} \Rightarrow 0.4 \leqslant p_k \leqslant 0.8$

Доказательство. $p_k = \underbrace{\frac{n}{n} \cdot \frac{n-1}{n} \cdot \ldots \cdot \frac{n-\sqrt{n}/2}{n}}_{(n-\sqrt{n}/2)/n \leqslant x \leqslant 1} \cdot \underbrace{\frac{n-\sqrt{n}/2-1}{n} \cdot \ldots \cdot \frac{n-\sqrt{n}}{n}}_{(n-\sqrt{n})/n \leqslant x \leqslant (n-\sqrt{n}/2)/n} \Rightarrow \left(\frac{n-\sqrt{n}/2}{n}\right)^{\sqrt{n}/2}\left(\frac{n-\sqrt{n}}{n}\right)^{\sqrt{n}/2} \leqslant$
$p_k \leqslant (1)^{\sqrt{n}/2}\left(\frac{n-\sqrt{n}/2}{n}\right)^{\sqrt{n}/2} \Rightarrow e^{-1/4}e^{-1/2} \leqslant p_k \leqslant e^{-1/4} \Rightarrow 0.4723 \leqslant p_k \leqslant 0.7788$ ■

$\forall k$ можно оценить p_k гораздо точнее:

$p_k = \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \ldots \cdot \frac{n-k+1}{n} = \frac{n!}{n^k(n-k)!} \approx \frac{(n/e)^n}{n^k((n-k)/e)^{n-k}} = \frac{1}{((n-k)/n)^{n-k}e^k} = \frac{1}{(1-k/n)^{n-k}e^k} \approx \frac{1}{(e^{-1})^{(n-k)k/n}e^k}$

Первое приближение сделано по формуле Стирлинга, второе по замечательному пределу.

• Поллард

Пусть у n минимальный делитель p, тогда $p \leqslant \sqrt{n}$. Сгенерируем $\sqrt[4]{n} \geqslant \sqrt{p}$ случайных чисел.
По парадоксу дней рождений какие-то два (a и b) дадут одинаковый остаток по модулю p.
При этом с большой вероятностью у a и b разный остаток по модулю $n \Rightarrow gcd(a-b, n)$ – нетривиальный делитель n. Осталось придумать, как найти такую пару a, b.

```
1 x = random [2..n-1]
2 k = pow(n, 1.0 / 4)
3 for i=0..k-1: x = f(x)
```

Здесь функция f – псевдорандом на $[0..n)$.
Например (без док-ва) нужными нам свойствами обладает функция $f(x) = (x^2 + 1) \bmod n$.
Так мы получили x, теперь переберём y.

```
1 y = f(x)
2 for i=0..k-1:
3     g = gcd(x - y, n)
4     if (g != 1 && g != n) return g
5     y = f(y)
```

Почему так можно? Рассмотрим последовательность $x_i = f^i(x_0)$. Она зацикливается по модулю p, что выглядит как буква «ρ». Наша цель — найти две точки друг над другом. Сделав достаточно много шагов из стартовой точки окажемся на цикле. После чего остаётся пройти ещё один раз весь цикл.

Полученный нами алгоритм не работает на маленьких n. Все такие n можно проверить за $n^{1/2}$. Также стоит помнить, что вероятность его успеха $\approx \frac{1}{2}$, поэтому для вероятности ≈ 1 всю конструкцию нужно запустить несколько раз.

Сейчас асимптотика – $\mathcal{O}(n^{1/4} \cdot T(gcd))$.

Чтобы получить чистое $\mathcal{O}(n^{1/4})$, воспользуемся тем, что $gcd(a,n) = 1 \wedge gcd(b,n) = 1 \Rightarrow gcd(ab \bmod n, n) = 1 \Rightarrow$ можно вместо `gcd(x-y,n)` смотреть на группы по $\log n$ разностей: `gcd(`$\prod_i$`(x-y`$_i$`),n)`, и только если какой-то `gcd` $\neq 1$, проверять каждый `y`$_i$ отдельно.

18.3. 3-SAT и random walk

• **Детерминированное решение за** $3^{n/2}$

Пусть решение X^* существует и в нём нулей больше чем единиц. Начнём с $X_0 = \{0, 0, \ldots, 0\}$, чтобы из X_0 попасть в X^* нужно сделать не более $\frac{n}{2}$ шагов. Если X_i не решение, то какой-то клоз не выполнен, значит в X^* одна из трёх переменных этого клоза имеет другое значение. Переберём, какая. Получили рекурсивный перебор глубины $\frac{n}{2}$, с ветвлением 3. Если в X^* единиц больше, начинать нужно с $X_0 = \{1, 1, \ldots, 1\}$. Нужно перебрать оба варианта.

• **Рандомизированное решение за** $3^{n/2}$

Упростим предыдущую идею: начнём со случайного X_0. Матожидание расстояния от ответа равно $\mathsf{E} = n/2$. с вероятностью $\frac{1}{2}$ он на расстоянии $\leqslant \frac{1}{2}n$ от ответа.Сделаем $\frac{1}{2}n$ шагов, выбирая каждый раз случайное из трёх направлений. На каждом шаге с вероятностью $\frac{1}{3}$ мы приближаемся к ответу. Итого с вероятностью $\frac{1}{n+1} \cdot \frac{1}{3^{n/2}}$ мы придём в ответ. Повторим $\mathcal{O}(3^{n/2}(n+1)) \approx \mathcal{O}(1.73^n)$ раз.

• **Рандомизированное решение за** 1.5^n

Почти такой же алгоритм. Сделаем теперь не $\frac{n}{2}$, а n шагов.

В процессе доказательства нам пригодится знание $\forall \alpha \ \sum_k (\binom{n}{k}\alpha^k) = (1+\alpha)^n$.
Доказательство: раскроем скобочки.

Анализ вероятностей. Если перебор за $3^{n/2}$ перебирал все варианты, то мы перебираем 1 вариант и угадываем с вероятностью $\geqslant p = \sum_k Pr[k]\frac{1}{3^k}$, где k – расстояние Хэмминга от X_0 до X^*, а $Pr[k] = \binom{n}{k}/2^n$ – вероятность того, что при выборе случайного X_0 расстояние равно k.
$p = \sum_k \frac{1}{2^n}\binom{n}{k}\frac{1}{3^k} = \frac{1}{2^n}(1+\frac{1}{3})^n = (\frac{2}{3})^n \Rightarrow$ повторим процесс 1.5^n раз.
Кстати, поскольку $Pr[k \leqslant \frac{n}{2}] = \frac{1}{2}$, достаточно делать даже не n, а $\frac{n}{2}$ шагов.

• **Рандомизированное решение за** 1.334^n

Schoning's algorithm (1999). Получается из предыдущего решения заменой $\frac{n}{2}$ шагов на $3n$ шагов. Вероятность успеха будет не менее $(3/4)^n \Rightarrow$ время работы $\mathcal{O}^*(1.334^n)$. Доказательство. Статья.
В той же статье дан более общий результат для k-SAT: $(\frac{2}{1+1/(k-1)})^n = (\frac{2(k-1)}{k})^n$.

Теорема 18.3.1. Вероятность успеха $(3/4)^n$

Доказательство. В прошлой серии мы находились на расстоянии k от ответа и хотели дви-

гаться всё время к ответу. А теперь посчитаем вероятность p того, что мы за первых $3k$ шагов сделали k шагов в неверную сторону, $2k$ шагов в верную сторону (итого пришли в ответ).

$$p = \binom{3k}{k} (\tfrac{1}{3})^{2k} (\tfrac{2}{3})^{k}$$

Приблизим $n! \approx \left(\frac{n}{e}\right)^n \Rightarrow \binom{3k}{k} \approx \frac{(3k)^{3k}}{(2k)^{2k}k^k} = \left(\frac{27}{4}\right)^k \Rightarrow p \approx \left(\frac{27}{4}\right)^k (\frac{1}{9})^k (\frac{2}{3})^k = (\frac{1}{2})^k$.

Последний шаг такой же, как в 1.5^n: $Pr[success] = 2^{-n} \sum_k \binom{n}{k} 2^{-k} = 2^{-n}(1+\frac{1}{2})^n = \left(\frac{3}{4}\right)^n$. ■

Лучшее сегодня решение 3-SAT: алгоритм PPSZ за 1.308^n от Paturi, Pudlak, Saks, Zani (2005).

• Поиск хороших кафешек

Пусть Вы в незнакомом городе, хотите найти хорошее кафе. Рассмотрим следующие стратегии:

1. Осматривать окрестность в порядке удаления от начальной точки
2. Random walk без остановок
3. Random walk на фиксированную глубину с возвратом

Первый, если Вы начали в не очень богатом на кафе районе не приведёт ни к чему хорошему. У второго будут проблемы, если в процессе поиска вы случайно зайдёте в промышленный район. Третий же в среднем не лучше, зато минимизирует риски, избавляет нас от обеих проблем. Для решения 3-SAT мы использовали именно третий вариант.

Замечание 18.3.2. Random Walk помогает находить гамильтонов путь в случайных графах.

Angluin,Valiant'1979 `[AV79]`: (random walk + posa-rotation) за $\mathcal{O}(n \log^2 n)$ дают гамильтонов путь для случайных графов, где $m \geqslant C \cdot n \log n$.
В 2021-м идею довели до $\mathcal{O}(n)$ `[Nenadov'Steger'Su]`, в этой же статье есть описание `[AV79]`.

18.4. Лемма Шварца-Зиппеля

Lm 18.4.1. Пусть дан многочлен P от нескольких переменных над полем $\mathbb{F}$.

$$(P \not\equiv 0) \Rightarrow \Pr_x[P(x) = 0] \leqslant \tfrac{degP}{|\mathbb{F}|}$$

Доказательство. Индукция по числу переменных.
База: многочлен от 1 переменной имеет не более $\deg P$ корней. Переход: wiki. ■

Следствие 18.4.2. Задача проверки тождественного равенства многочлена нулю $\in$ coRP.

Доказательство. Подставим случайный x в поле $\mathbb{F}_q$, где (q – простое) $\wedge$ ($q > 2 \deg P$). ■

• (*) Совершенное паросочетание в произвольном графе

Дан неорграф, заданный матрицей смежности c. Нужно проверить, есть ли в нём совершенное паросочетание.

Def 18.4.3. *Матрица Татта* T: $T_{ij} = -T_{ji}, T_{ij} = \begin{cases} 0 & c_{ij} = 0 \\ x_{ij} & c_{ij} = 1 \end{cases}$

Здесь x_{ij} – различные переменные. Пример: $c = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \to \begin{bmatrix} 0 & x_{12} & x_{13} \\ -x_{12} & 0 & 0 \\ -x_{13} & 0 & 0 \end{bmatrix}$

Теорема 18.4.4. $\det T \not\equiv 0 \Leftrightarrow \exists$ совершенное паросочетание

Теорему вам докажут в рамках курса дискретной математики, нам же сейчас интересно, как проверить тождество. Если бы матрица состояла из чисел, определитель можно было бы посчитать Гауссом за $\mathcal{O}(n^3)$. В нашем случае определитель – многочлен степени n. Подставим во все x_{ij} случайные числа, посчитаем определитель по модулю $p = 10^9 + 7$. Получили вероятностный алгоритм с ошибкой $\frac{n}{p}$ и временем $\mathcal{O}(n^3)$.

Следствие 18.4.5. Мы умеем искать размер максимального паросочетания за $\mathcal{O}(n^3 \log n)$

Доказательство. Бинпоиск по ответу. Чтобы проверить, есть ли в графе паросочетание размера хотя бы k рёбер ($2k$ вершин), добавим $n-2k$ фиктивных вершин, которые соединены со всеми и проверим в полученном графе наличие совершенного паросочетания за $\mathcal{O}(n^3)$. ■

• Равенство мультимножеств

На практике изучим, как за $\mathcal{O}(n)$ времени и $\mathcal{O}(1)$ памяти проверить равенство двух мультимножеств. В будущем мы это ещё раз сделаем в теме «хеширование».

• (*) Гамильтонов путь

В 2010-м году в неорграфе научились проверять наличие гамильтонова пути за $\mathcal{O}^*(1.657^n)$. Можно почитать в оригинальной статье Бьёркланда (Björklund).

18.5. Random shuffle

```
1 void Shuffle(int n, int *a)
2     for (int i = 0; i < n; i++)
3         swap(a[i], a[rand() % (i+1)]);
```

Утверждение 18.5.1. После процедуры `Shuffle` все перестановки a равновероятные.

Доказательство. Индукция. База: после фазы $i = 0$ все перестановки длины 1 равновероятны. Переход: выбрали равновероятно элемент, который стоит на i-м месте, после этого часть массива $[0..i)$ по индукции содержит случайную из $i!$ перестановок. ■

• Применение

Если на вход даны некоторые объекты, полезно их первым делом перемешать.

Пример: построить бинарное дерево поиска, содержащее данные n различных ключей. Если добавлять их в пустое дерево в данном порядке, итоговая глубина может быть n, а время построения соответственно $\Theta(n^2)$. Если перед добавлением сделать `random shuffle`, то матожидание глубины дерева $\Theta(\log n)$, матожидание времени работы $\Theta(n \log n)$. Оба факта мы докажем при более подробном обсуждении деревьев поиска.

18.6. Дерево игры [stanford.edu]

Пусть есть игра, в которую играют два игрока. Можно построить дерево игры (возможно бесконечное) – дерево всех возможных ходов. Теперь игра – спуск по дереву игры. Для некоторых простых случаев хорошо известны «быстрые» способы сказать, кто выиграет.

• Игра на 0-1-дереве

Задача: в листьях записаны числа 0 и 1, первый победил, если игра закончилась в 1, иначе

победил второй. Пусть дерево – полное бинарное глубины n.

Решение: простой рандомизированный алгоритм «*сперва сделаем случайный ход, а второй сделаем только если не выиграли первым ходом*». Асимптотика $\mathcal{O}(1.69^n)$. Анализ в практике.

• Игра на OR-AND-дереве

Близкая по смыслу, но не игровая, а вычислительная задача. Дано полное бинарное дерево, в листьях 0 и 1. В промежуточных вершинах чередующиеся `AND` и `OR` гейты от детей.

Задача равносильна игре на 0-1-дереве: `AND` означает «чтобы первый выиграл, второй в обеих ветках должен проиграть», `OR` означает «чтобы второй проиграл, первый в одной из веток должен выиграть». Чередование `AND`/`OR` значит «ходят по очереди».

• Игра на min-max-дереве

Полное бинарное дерево. В листьях записаны $\mathbb{Z}$ числа из $[0, m)$.
Первый игрок максимизирует результат игры, второй минимизирует.

Решение #1: бинпоиск по ответу и игра на 0-1-дереве
Решение #2: (*) альфа-бета отсечение
Решение #3: (*) комбинация этих идей – алгоритм MTD-f (статья, wiki). MTD-f в своё время стал прорывом в шахматных стратегиях.

18.7. k-путь

Задача. Хотим найти k-путь – простой путь длины ровно k **вершин** из a в b. Для $n = k$ это ровно гамильтонов путь, мы умеем его искать за $2^n n$. При малых k мы пока не умеем ничего лучше перебора: n^k или точнее deg^k. В этом разделе мы предложим вероятностный алгоритм за $exp(k) \cdot poly(n)$.

• Алгоритм

Покрасим вершины в k цветов, каждую в случайный. Теперь за $\mathcal{O}(2^k E)$ найдём разноцветный k-путь, если он есть (динамика как для гамильтонова пути).
Если k-путь есть, покрасить его k^k способов, а разноцветно покрасить $k!$ способов.
Вероятность успеха $\frac{k!}{k^k} \geqslant e^{-k} \Rightarrow$ повторим $\mathcal{O}(e^k)$ раз. Итого: $\mathcal{O}^*((2e)^k)$.

18.8. HyperLoglog

Почитать: [neerc] [paper]

Задача: у нас очень мало памяти, нам потоком дают элементы множества A, нужно оценить количество различных элементов в A. Для точного ответа нужно было бы $\Omega(|A|)$ памяти $\Rightarrow$ мы можем только приблизить.

Алгоритм.
1. Захешируем числа $a \in A$, чтобы они стали случайными целыми из $[1, 2^k)$.

2. Будем поддерживать $m =$«максимальное количество подряд идущих нулей в младших битах $hash(a)$». Например, $k = 5$, $h(a) = 00110$, $f(h(a)) = 2$ нуля.

2*. Храним m=0..k $\Rightarrow$ нам нужно $\log k$ бит памяти. Если мы хотим хранить числа порядка $C = 2^k$, то памяти нужно $\log\log C$ бит, отсюда название. Например, для $C = 2^{32}$ нужно 5 бит.

3. $\Pr_a[f(h(a)) \geqslant m] = \frac{1}{2^m} \Rightarrow$ оценим кол-во различных, как 2^m.

Улучшение алгоритма. Распихаем исходные числа по R случайным кучкам. Для каждой кучки реализуем решение выше, нужно $R \log\log C$ бит памяти. Пусть m_i – максимум нулей для кучки i. Можно оценить ответ, как $\sum_i 2^{m_i}$, можно лучше, как среднегармоническое: $\alpha \frac{R^2}{\sum_i \frac{1}{2^{m_i}}}$. Где α – константа при фиксированном k, для $k=32$ используют $\alpha=0.697$, для $k=64$ $\alpha=0.709$.

18.9. (*) Квадратный корень по модулю

Задача: даны простое p и a, найти x: $x^2 = a \bmod p$.

Хорошо известны два решения: алгоритм Тоннелли-Шенкса'1973 и алгоритм Циполла'1907. Оба алгоритма вероятностны и требуют угадать квадратичный невычет.
В этой главе мы изучим только второй [1][2], как более быстрый и простой в реализации.

Алгоритм: возьмём $P(x) = (x+i)^{(p-1)/2} - 1$ и $A(x) = x^2 - a = (x-x_1)(x-x_2)$.
Посчитаем $gcd(P(x), A(x))$, если он содержит ровно один из $x-x_1$ и $x-x_2$, мы победили.

Теорема 18.9.1. Вероятность успеха $\geqslant \frac{1}{2}$. Время работы – $\mathcal{O}(\log n)$ делений чисел порядка p.

Корни существуют $\Leftrightarrow$ символ Лежандра $a^{(p-1)/2} = 1$. В реализации вместо gcd будем вычислять $R(x) = P(x) \bmod A(x)$ – возведение в степень с умножением похожим на комплексные числа:

$$(p_1x + q_1)(p_2x + q_2) = (p_1q_2 + p_2q_1)x + (p_1p_2a + q_1q_2)$$

Пусть $R(x) = bx + c$, тогда при $b \neq 0$ пробуем корень $-cb^{-1}$.

```
def root(a, p):
    if pow_mod(a, (p-1)/2, p) != 1: return -1 # символ Лежандра
    while True:
        i = random [0..p)
        bx+c = pow_mod(x+i, (p-1)/2, x^2-a) # многочлен в степени по модулю
        if b != 0:
            z = -c/b
            if z^2 = a \bmod p: return z
```

Доказательство Thm 18.9.1. Для начала посмотрим на символ Лежандра и заметим, что (α/β – невычет) $\Leftrightarrow$ (ровно один из $\{\alpha, \beta\}$ – вычет). Теперь исследуем корни многочлена из решения $P(x) = (x+i)^{(p-1)/2} - 1$. Ими являются такие z, что $(z-i)$ – квадратичный вычет.
Мы хотим оценить вероятность того, что «ровно один из $\{x_1, x_2\}$ является корнем $P(x)$» $\Leftrightarrow$ «$(i-x_1)/(i-x_2)$ – невычет». $\forall x_1 \neq x_2$ $i \xrightarrow{f} \frac{i-x_1}{i-x_2} \Rightarrow f$ – биекция. Невычетов $(p-1)/2$. ■

Замечание 18.9.2. Если длина p равна n, то E(времени работы) при «умножении по модулю» за $\mathcal{O}(n \log n)$ получается $\mathcal{O}(n^2 \log n)$.

Замечание 18.9.3. Алгоритм можно обобщить на извлечение корня k-й степени.

Лекция #19: Кратчайшие пути

19-я пара, 2024/25

Def 19.0.1. *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.

19.1. Short description

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

Задача	Время	Название	Год, Автор	Изучим?
ациклический граф	$V+E$	Динамика	–	+
$w_e = 1$	$V+E$	Поиск в ширину	1950, Moore	+
$w_e \geqslant 0$	V^2+E	–	1956, Dijkstra	+
$w_e \geqslant 0$	$V \log V + E$	Dijkstra + fib-heap	1984, Fredman & Tarjan	+
$w_e \in \mathbb{N}$, неорграф	$V+E$	–	2000, Thorup	-
$w_e \geqslant 0$	$A^* \leqslant$ Dijkstra	(A^*) A-star	1968, Nilsson & Raphael	+
любые веса	VE	–	1956, Bellman & Ford	+
$w_i \in \mathbb{Z} \cap [-N, \infty)$	$E\sqrt{V}\lceil\log(N+2)\rceil$	–	1994, Goldberg	+

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	Год, Автор	Изучим?
любые веса	V^3	–	1962, Floyd & Warshall	+
любые веса	$VE + V^2 \log V$	–	1977, Johnson	+
любые веса	$V^3/2^{\Omega(\log^{1/2} V)}$	–	2014, Williams	-

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).
Алгоритмы Гольдберга и Джонсона основаны на не известной нам пока идее потенциалов.

19.2. 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.

19.3. Модификации bfs

19.3.1. 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 19.3.1. 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 19.3.2. 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 19.3.3. 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 $\mathcal{O}(k(V + E))$.

19.3.2. 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: $\mathcal{O}(k(V+E)) = \mathcal{O}(V+E)$.

19.4. 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.
(c) С кучей Фибоначчи даст время работы $\mathcal{O}(E + V \log V)$.

(d) С деревом Ван-Эмбде-Боэса даст время работы $\mathcal{O}(E \log\log C)$ для целых весов из $[0, C)$.
(e) Существуют более крутые кучи, дающие время $\mathcal{O}(E+V \log\log V)$ на целочисленных весах.

Замечание 19.4.1. 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 $E \log V$.

19.5. A* (A-звездочка)

Представьте, что едете из Петербурга в Москву. Представьте себе дорожную сеть страны. Перед поездкой ищем кратчайший маршрут. Что делает Дейкстра? Идёт из Петербурга во все стороны, перебирает города в порядке удаления от Петербурга. То есть, попытается в частности ехать в Москву через Петрозаводск (427 км) и Выборг (136 км). Логичней было бы в процессе поиска пути рассматривать только те вершины (города, населённые пункты, развязки), проезжая через которые, теоретически можно было бы быстрее попасть в Москву...

Почему так? Дейкстра – алгоритм для SSSP. Цель Дейкстры – найти расстояния до всех вершин. При поиске расстояния до конкретной вершины t в Дейкстру можно добавить естественную оптимизацию – `break` после того, как достанем t из кучи.

Алгоритм A* можно воспринимать, как "модификацию Дейкстру, которая не пытается ехать Петербург $\rightsquigarrow$ Москва через Выборг". Обозначим начальную вершину s, конечную t, расстояние между вершинами a и b за d_{ab}. $\forall v$ оценим снизу d_{vt} *из физического смысла задачи*, как f_v.

Алгоритм A*: в Дейкстре ключ кучи d_v заменим на $d_v + f_v$.
То есть, вершины будем перебирать в порядке возрастания не d_v, а $d_v + f_v$.
Остановим алгоритм, когда вынем из кучи вершину t.

Lm 19.5.1. Если f никогда не переоценивает расстояние ($\forall v\colon 0 \leqslant f_v \leqslant dist(v,t)$), то алгоритм A* найдёт корректные расстояния. Правда при этом он может работать экспоненциально долго, так как будет доставать одну и ту же вершину из кучи много раз.

Теорема 19.5.2. Если функция f удовлетворяет неравенству треугольника: $\forall e\colon a \to b$ верно $f_a \leqslant f_b + w_{ab}$, то алгоритм A* достанет каждую вершину из кучи не более одного раза.

Доказательство. Рассмотрим вершину $a\colon d_a + f_a = \min$. Возьмём $\forall v$ и путь из неё в $a\colon\ v \to u \to \dots \to c \to b \to a$. Нужно доказать, что $d_a + f_a \leqslant (d_v + w_{vu} + \dots w_{cb} + w_{ba}) + f_a = F$.
Из неравенства $\triangle$ имеем $w_{ba} + f_a \geqslant f_b \wedge w_{cb} + f_b \geqslant f_a \wedge \dots \Rightarrow F \geqslant d_v + f_v \geqslant d_a + f_a$ ■

Следствие 19.5.3. Если для f верно неравенство треугольника, алгоритм можно остановить сразу после вынимания t из кучи.

Замечание 19.5.4. Заметим, что в доказательстве мы нигде не пользовались неотрицательностью весов. Для отрицательных обычно сложно найти функцию f с неравенством треугольника. Скоро мы введём технику потенциалов, узнав её, полезно ещё раз перечитать это место.

Замечание 19.5.5. "Оценка снизу" – лишь физический смысл функции f_v. Запускать алгоритм мы можем взяв любые вещественные числа.

$\forall f$, увеличив все f_v на константу, можно сделать $f_t = 0$. Будем рассматривать только такие f.

Теорема 19.5.6. Алгоритм A* на функции f, удовлетворяющий трём условиям: (a) неравенству треугольника, (b) $f_t = 0$, (c) $f_v \geqslant 0$ переберёт подмножество тех вершин, которые перебрала бы Дейкстра с `break`.

Доказательство. Дейкстра переберёт v: $d_v \leqslant d_t$.
A* переберёт v: $d_v + f_v \leqslant d_t + f_t = d_t \Rightarrow d_v \leqslant d_t - f_v \leqslant d_t$. ■

Сделав $f_v = \max(0, f_v)$ можно получить неотрицательность f_v. При этом в случае $w_e \geqslant 0$ неравенство треугольника не испортится.

19.6. Флойд

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

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

$$\texttt{f[k+1,i,j] = min(f[k][i][j], f[k][i][k] + f[k][k][j])}.$$

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 $V \leqslant 1000$.

19.6.1. 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]`.

19.6.2. 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 19.6.1. ($\exists$ отрицательный цикл, проходящий через i) $\Leftrightarrow$ (по окончании Флойда `d[i,i]<0`).

Lm 19.6.2. ($\nexists$ 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 19.6.3. 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;
```

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 #20: Кратчайшие пути

20-я пара, 2024/25

Мы уже знаем поиск 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.

20.1. 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".

20.2. 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 20.2.1. $\forall$ 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 20.2.2. $\forall$ 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 20.2.3. После релаксации v произошла релаксация $p_v \Rightarrow d[p_v] + w[p_v, v] < d[v]$.

Lm 20.2.4. 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 Lm 20.2.2, 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 20.2.5. Вес полученного цикла отрицательный.

Доказательство. Сложим по всем рёбрам цикла неравенство из леммы Lm 20.2.2, 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 $z\colon p_z = y$ неравенство из леммы Lm 20.2.2 строгое. ■

Из доказанных лемм следует:
Теорема 20.2.6. Алгоритм восстановления отрицательного цикла корректен.

20.3. Модификации Форд-Беллмана

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

20.3.1. 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 $\mathcal{O}(E \log V)$.

20.3.2. 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] > d[v] + w[v,x] then
7.                  d[x] = d[v] + 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] > d[v] + w[v,x]then
7.              d[x] = d[v] + w[v,x]
8.              if (!inQueue[x]) inQueue[x] = 1, q.push(x)
```

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}.

20.3.3. (*) Форд-Беллман с random shuffle

Сейчас мы оценим, сколько в худшем и лучшем случае работает Форд-Беллман с `break`.
Здесь и далее одна итерация – один раз перебрать за $\Theta(E)$ все рёбра графа.

Lm 20.3.1. На любом тесте в лучшем случае сделает 1 итерацию.

Доказательство. Рассмотрим дерево кратчайших путей. Пусть в массиве рёбер, который просматривает Форд-Беллман, все рёбра упорядочены от корня к листьям. Тогда при просмотре ребра $a \to b$ расстояние до вершины a по индукции уже посчитано верно. ■

Lm 20.3.2. На любом тесте Форд-Беллман сделает итераций не больше, чем глубина дерева кратчайших путей.

Lm 20.3.3. $\exists$ тест: Форд-Беллман в худшем случае сделает $V-1$ итерацию.

Доказательство. Возьмём граф, в котором дерево кратчайших путей – один путь дины $V-1$. Пусть в массиве рёбер, который просматривает Форд-Беллман, рёбра упорядочены от листьев к корню. Тогда после i итераций верно посчитано расстояние для ровно $i+1$ вершины. ■

• **Форд-Беллманом с `random_shuffle` #1**

```
vector<int> d(n, INFINITY);
d[s] = 0;
random_shuffle(all_edges_in_graph); // перемешали рёбра только 1 раз в начале
while (расстояния улучшаются)
    for (Edge e : all_edges_in_graph)
        relax(d[e.end], d[e.start] + e.weight);
```

Lm 20.3.4. На любом тесте матожидание числа итераций не больше $\frac{V}{2}$.

Доказательство. Худший тест: дерево кратчаших путей – путь $e_1, e_2, \dots e_{V-1}$; до каждой вершины $\exists!$кратчайший путь. Пусть $e_i\colon v_i \to v_{i+1}$ и расстояние до v_i найдено на k_i-й фазе. $Pr[k_i \neq k_{i+1}] = \frac{1}{2}$ (зависит только от порядка рёбер e_i и e_{i-1} в перестановке). Далее пользуемся линейностью матожидания. $\mathbb{E}[\#\{i\colon k_i \neq k_{i+1}\}] = \sum_i Pr[k_i \neq k_{i+1}] = \frac{V-1}{2}$. ■

• **Форд-Беллманом с `random_shuffle` #2**

Забавно, но весьма похожий код имеет уже другую оценку числа фаз.

```
vector<int> d(n, INFINITY);
d[s] = 0;
while (расстояния улучшаются)
    random_shuffle(all_edges_in_graph); // переупорядочиваем рёбра каждый раз
    for (Edge e : all_edges_in_graph)
        relax(d[e.end], d[e.start] + e.weight);
```

Lm 20.3.5. На любом тесте матожидание числа итераций не больше $\frac{V}{e-1} \approx \frac{1}{1.72}$.

Доказательство. Пусть перед началом очередной итерации мы верно знаем расстояния до вершин из множества A. Рассмотрим $v \notin A$ и путь p_v из A до v в дереве кратчайших путей. Чтобы на текущей итерации найти расстояние до v, все рёбра p_v должны быть в перестановке в том же порядке, что и в p_v. Вероятность по всем $n!$ перестановкам этого события равна $\frac{1}{|p_v|!}$. Обозначим за k число вершин, до которых мы найдём расстояние.
Пользуемся линейностью матожидания $\mathbb{E}[k] = \sum_{v \notin A} \frac{1}{|p_v|!}$. Худший случай: «дерево кратчайших путей – бамбук», $\{|p_v|\} = \{1, 2, \dots\} \Rightarrow \mathbb{E}[k] = \sum_{i \geqslant 1} \frac{1}{i!} = e - 1$. ■

20.4. Потенциалы Джонсона

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

Def 20.4.1. *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 20.4.2. $\forall s, t$ 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.

20.5. Цикл минимального среднего веса

Задача: найти цикл, у которого среднее арифметическое весов рёбер минимально.

Искомый вес цикла обозначим μ. Веса рёбер w_e.

Lm 20.5.1. Среди оптимальных ответов существует простой цикл.

Доказательство. Из оптимальных выберем цикл минимальный по числу рёбер. Пусть он не простой, представим его, как объединение двух меньших. У одного из меньших среднее арифметическое не больше. Противоречие $\Rightarrow$ посылка ложна $\Rightarrow$ простой. ■

Lm 20.5.2. Если все w_e уменьшить на x, μ тоже уменьшится на x.

• Алгоритм
Бинарный поиск по ответу, внутри нужно проверять "есть ли цикл среднего веса меньше x?"
$\Leftrightarrow$ "есть ли в графе с весами $(w_e - x)$ цикл среднего веса меньше 0"
$\Leftrightarrow$ "есть ли в графе с весами $(w_e - x)$ отрицательный цикл".
Умеем это проверять Форд-Беллманом за $\mathcal{O}(VE)$. Скоро научимся за $\mathcal{O}(E\sqrt{V} \log N)$.

Границы бинпоиска – минимальный и максимальный вес ребра.
Правую можно взять даже точнее – средний вес любого цикла (цикл умеем искать за $\mathcal{O}(E)$).
Если сейчас отрезок бинпоиска $[L, R)$, у нас уже точно есть цикл C веса меньше R.

Lm 20.5.3. $R - L \leqslant \frac{1}{V^2} \Rightarrow C$ – оптимален.

Доказательство. Рассмотрим любые два цикла, их средние веса $\frac{s_1}{k_1}$ и $\frac{s_2}{k_2}$. $(3 \leqslant k_1, k_2 \leqslant V)$.
Предположим, что $\frac{s_1}{k_1} \neq \frac{s_2}{k_2}$ и оценим снизу их разность: $|\frac{s_1}{k_1} - \frac{s_2}{k_2}| = \frac{|s_1 k_2 - s_2 k_1|}{k_1 k_2} \geqslant \frac{1}{k_1 k_2} \geqslant \frac{1}{V^2} \Rightarrow$
При $R - L \leqslant \frac{1}{V^2}$ на промежутке $[L, R)$ не может содержаться двух разных средних весов. ■

20.6. Алгоритм Карпа

Добавим фиктивную вершину s и рёбра веса 0 из s во все вершины.
Насчитаем за $\mathcal{O}(VE)$ динамику $d_{n,v}$ – вес минимального пути из s в v из ровно n рёбер.
Если для какой-то вершины пути нужной длины не существует, запишем бесконечность.

Теорема 20.6.1. Пусть $Q = \min_v \left(\max_{k=1..n-1} \frac{d_{n,v}-d_{k,v}}{n-k} \right)$. Считаем $+\infty - +\infty = +\infty$. Тогда $\mu = Q$.

Алгоритм заключается в замене бинпоика на формулу и поиске отрицательного цикла в графе с весами $(w_e - \mu - \frac{1}{V^2})$: в графе с весами $(w_e - \mu)$ есть нулевой, мы ещё чуть уменьшили веса. Осталось доказать теорему ;-). Разобьём доказательство на несколько лемм.

Lm 20.6.2. Все w_e уменьшили на $x \Rightarrow Q$ уменьшится на x.

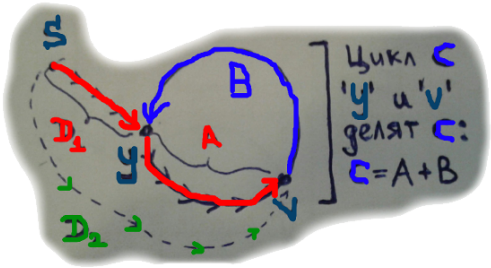
Доказательство. $d'_{n,v} = d_{n,v} - nx \Rightarrow \frac{d'_{n,v}-d'_{k,v}}{n-k} = \frac{d_{n,v}-d_{k,v}-(n-k)x}{n-k} = \frac{d_{n,v}-d_{k,v}}{n-k} - x.$ ■

Lm 20.6.3. Если доказать теорему для случая $\mu = 0$, то она верна в общем случае

Доказательство. Уменьшив все w_e на μ по Lm 20.5.2 перейдём к случаю $\mu' = 0$ По теореме получаем $Q' = 0$. Увеличим обратно веса на μ, по лемме Lm 20.6.2 получаем $Q = Q' + \mu = \mu$. ■

Lm 20.6.4. $(\mu = 0) \Rightarrow (Q = 0)$

Доказательство. Вместо сравнения $\max_{k=1..n-1} \frac{d_{n,v}-d_{k,v}}{n-k}$ с 0 достаточно сравнить с 0 максимум числителей $\max_{k=1..n-1}(d_{n,v} - d_{k,v}) = d_{n,v} - \min_{k=1..n-1} d_{k,v}$.
$\mu = 0 \Rightarrow$ нет отрицательных циклов $\Rightarrow \min_{k=1..n-1} d_{k,v}$ – вес кратчайшего пути.
Поскольку $d_{n,v}$ – вес какого-то пути, получаем $d_{n,v} - \min_{k=1..n-1} d_{k,v} \geqslant 0$.
Осталось предъявить вершину v, для которой $d_{n,v}$ – кратчайший путь от s до v. Для этого рассмотрим нулевой цикл C, любую вершину y на C, кратчайший путь из s в y. Пусть этот путь состоит из k рёбер и имеет вес D_1. Пройдём из y по циклу на $n-k$ рёбер вперёд (возможно прокружимся по циклу несколько раз). Остановились в вершине v, пройденный путь до v содержит n рёбер, имеет вес $D_1 + A$ (цикл C вершинами y и v разбился на две половины с весами A и B, при этом $A + B = 0$). Докажем что найденный путь кратчайший до v: от противного, пусть есть другой $D_2 < D_1 + A$, тогда продолжим его по циклу до y, получим $D_2 + B < D_1 + A + B = D_1$. Получили противоречие с минимальностью D_1. ■



20.7. (*) Алгоритм Гольдберга

Цель: придумать алгоритм поиска потенциалов, делающих все веса неотрицательными, работающий за $\mathcal{O}(EV^{1/2}\log N)$. В исходном графе веса из $\mathbb{Z}\cap[-N,\infty)$.
Для начала решим задачу для весов из $\mathbb{Z}\cap[-1,\infty)$.

Интересными будем называть вершины, в которые есть входящие рёбра веса -1.
Текущий граф будем обозначать G (веса меняются). Подграф G, состоящий только из рёбер веса 0 и -1 будем обозначать H. Будем избавляться от интересных вершин, следя за тем, чтобы не появлялось новых отрицательных рёбер.

20.7.1. (*) Решение за $\mathcal{O}(VE)$

Возьмём интересную вершину v и за $\mathcal{O}(E)$ сделаем её не интересной. Попробуем уменьшить её потенциал на 1. Веса входящих рёбер увеличатся на 1, а исходящих уменьшатся на 1. Если у v нет исходящих рёбер веса $\leqslant 0$, это уже успех. Теперь рассмотрим множество $A(v)$ – достижимые в H из v. Уменьшим потенциал всех вершин из $A(v)$ на 1: рёбра внутри $A(v)$ не поменяли вес, все входящие в $A(v)$ увеличили вес на 1, исходящие уменьшили на 1. $w(out(A(v)))\geqslant 1\Rightarrow$ новых отрицательных рёбер не появилось. $\forall x$: из x есть ребро в v веса -1 имеем $x\notin A(v)$ (иначе мы нашли отрицательный цикл $v\rightsquigarrow x\to v$) $\Rightarrow$ вершина v перестала быть интересной.

20.7.2. (*) Решение за $\mathcal{O}(EV^{1/2})$

Осталось научиться исправлять вершины пачками, а не по одной. Рассмотрим компоненты сильной связности в H, если хотя бы одна содержит ребро веса -1, то мы нашли отрицательный цикл. Иначе каждую компоненту можно сжать в одну вершину. Теперь H ацикличен. Добавим фиктивную вершину s, из неё нулевые рёбра во все вершины, найдём динамикой в H за $\mathcal{O}(E)$ кратчайшие расстояния от s до всех вершин. Получили дерево кратчайших путей. Слоем B_d назовём все вершины, на расстоянии d от s (заметим, $d\leqslant 0$).

Lm 20.7.1. Пусть всего интересных вершин k. Или есть слой B_d, в котором $\sqrt{k}$ интересных вершин, или есть путь от s в лист дерева, на котором $\sqrt{k}$ интересных вершин.

Доказательство. Рассмотрим v: $d_v=\min$. Пусть $d_v\leqslant-\sqrt{k}\Rightarrow$ на пути до v хотя бы $\sqrt{k}$ раз было ребро -1, хотя бы столько интересных вершин. Иначе слоёв всего $\leqslant\sqrt{k}$, по принципу Дирихле хотя бы в одном $\geqslant\sqrt{k}$ вершин. ■

• Решение для широкого слоя за $\mathcal{O}(E)$

Рассмотрим любой слой B_d. Уменьшим на 1 потенциал всех вершин $A(B_d)$ (достижимые в H). Докажем, что все вершины в B_d стали неинтересными. Рассмотрим ребро веса -1 из x в $v\in B_d$. Пусть $x\in A(B_d)\Rightarrow$ мы неверно нашли расстояние до v: мы считаем, что оно d, но есть путь $s\rightsquigarrow B_d\rightsquigarrow x\to v$ длины не более $d-1\Rightarrow x\notin A(B_d), v\in A(B_d)\Rightarrow$ вес ребра увеличится.

• Решение для длинного пути за $\mathcal{O}(E)$

Занумеруем интересные вершины на пути **от конца** к s: $v_1,v_2,v_3,\ldots$.
Исправим вершину v_1, уменьшив на 1 потенциал $A(v_1)$. Теперь, исправляя вершину v_2, заметим, что $A(v_1)\subset A(v_2)\Rightarrow$ мы можем ходить только по непосещённым вершинам.
Казалось бы достаточно, запуская dfs из v_2, пропускать помеченные dfs-ом от v_1 вершины.
Но есть ещё ребра, исходящие из $A(v_1)$ веса 1. После изменения потенциала $A(v_1)$ их вес стал 0, поэтому dfs из v_2 должен по ним пройти. Чтобы быстро находить все такие рёбра, заведём

структуру данных, в которую будем добавлять **все** пройденные рёбра.
Структура данных будет уметь делать три вещи: добавить ребро, уменьшить вес всех рёбер на 1, перебрать все рёбра веса 0 за их количество.

```
1 vector<int> edges[N];
2 int zero;
3 void add(int i, int w) { edges[zero + w].push_back(i); }
4 void subtract()        { zero += 1; }
5 vector<int> get()      { return edges[zero]; }
```

Хотим найти $A(v_i) \Rightarrow$ запускаем dfs от v_i и от всех концов рёбер, возвращённых `get()`.
После того, как нашли $A(v_i)$ вызываем `subtract()`.
Итого суммарное время поиска $A(v_1), A(v_2), A(v_3), \dots$ равно $\mathcal{O}(E)$.

• **Время работы**

Мы научились за $\mathcal{O}(E)$ получать $k \to k - \sqrt{k}$. Пока $k \geqslant \frac{k}{2}$ мы вычитаем как минимум $\sqrt{k/2} \Rightarrow$ за $\leqslant (k/2)/\sqrt{k/2} = \sqrt{k/2}$ шагов мы получим $\leqslant k/2$.
Итого $\sqrt{k/2} + \sqrt{k/4} + \dots = \Theta(\sqrt{k})$ шагов. Итого $\mathcal{O}(E\sqrt{k})$, где k – число интересных вершин.

20.7.3. (*) Общий случай

Также, как мы от графа с весами рёбер $\geqslant -1$ переходили к весам $\geqslant 0$,
можно от весов рёбер $\geqslant -(2k{+}1)$ перейти к весам $\geqslant -k$.
Выше мы делили рёбра графа на 3 группы: $\{-1, 0, (0, +\infty)\}$.
Для $-(2k{+}1)$ группы будут $\{[-(2k{+}1), -k), [-k, 0], (0, +\infty)\}$. Потенциал будем менять на $k{+}1$.

Лекция #21: DSU, MST и Йен

21-я пара, 2024/25

21.1. 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 $0 \dots n-1$.

21.1.1. Решения списками

Напишем максимально простую реализацию, получится примерно следующее:

```
list<int> sets[n]; // храним множества в явном виде
int id[n]; // для каждого элемента храним номер множества
void init(int n):
    for (int i = 0; i < n; i++)
        id[i] = i, sets[i] = {i};
int get(int i):
    return id[i]; // O(1)
void join(int a, int b):
    sets[id[a]] += sets[id[b]]; // O(1), += не определён... но было бы удобно =)
    for (int x : sets[id[b]]) // долгая часть
        id[x] = id[a];
```

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

Теорема 21.1.1. 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 21.1.2. Суммарное время работы m `join`-ов $\mathcal{O}(m + n \log n)$.

21.1.2. 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:

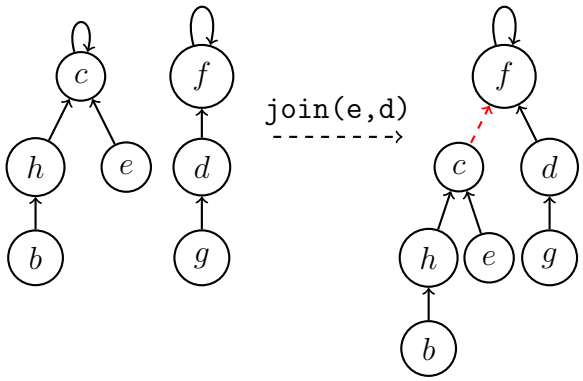
• **Эвристика сжатия путей:** все рёбра, по которым пройдёт `get` перенаправить в корень.

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

Def 21.1.3. *Ранг вершины – глубина её поддерева, если бы не было сжатия путей. Сжатие путей не меняет ранги. Ранг листа равен нулю.*

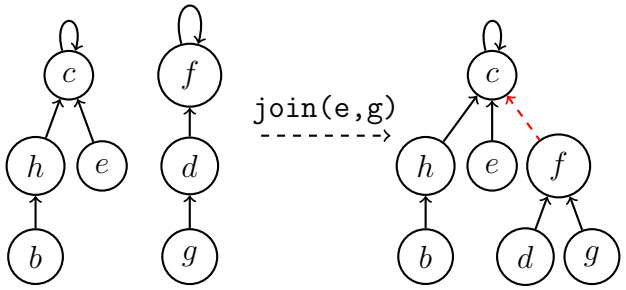
Вот базовая реализация без ранговой эвристики и без сжатия путей:

```
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 21.1.4. 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 21.1.5. 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 21.1.6. *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 21.1.7. 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. ■

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

Теорема 21.1.8. 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 21.1.9. 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 $\Rightarrow \sum a_i \leqslant n\log^* n$. ■

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

21.2.1. (*) Интуиция и $\log^{**} n$

Теорема 21.2.1. 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 21.2.2. Аналогично можно рассмотреть *трижды крутые* и даже k-раз крутые рёбра. Тогда суммарное время всех `get`-ов будет $\Theta((m + n)k + m\log^{**\cdots*} n)$.

21.2.2. (*) Введение обратной функции Аккермана

Def 21.2.3. *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 21.2.4. $A_k(n) > n$

Lm 21.2.5. $f(t) = A_t(1)$ 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 21.2.6. *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 $\alpha(t) \leqslant 4$.

21.2.3. (*) 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 21.2.7. *Порядок элемента.* $\forall$ *обычного элемента* e $\quad level(e) = \max\{k \mid r_{p_e} \geqslant A_k(r_e)\}$.

Def 21.2.8. *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 21.2.9. $\forall$ 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 21.2.10. $\forall$ 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 21.2.11. *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 21.2.12. *Потенциал. Возьмём $\varphi = \sum_e \varphi(e)$.*

Обозначим через φ_t потенциал после первых t операций.

Lm 21.2.13. $\varphi_0 = 0$, $\forall i\ \varphi_i \geqslant 0$.

Доказательство. Изначально все элементы – представители, и все $r_e = 0$. Далее все $r_e \geqslant 0$, появляются обычные элементы. Используем Lm 21.2.9 и Lm 21.2.10, получаем неотрицательность потенциалов обычных элементов. ■

Lm 21.2.14. Если у обычной вершины v увеличить ранг отца, то $\varphi(v) \searrow$.

Теорема 21.2.15. Для операции типа *join* амортизированное время $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$.

Доказательство. Пусть мы присвоили $p_b = a$. 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 21.2.16. Для операции типа *get* амортизированное время $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$.

Доказательство. После сжатия путей по Lm 21.2.14 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 Lm 21.2.9 $\exists$ 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 21.2.17. Суммарное время работы m произвольных операций с СНМ – $\mathcal{O}(m \cdot \alpha(n))$.

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

21.3. MST: Минимальное Остовное Дерево

Def 21.3.1. *Остовное дерево связного графа – подмножество его рёбер, являющееся деревом.*

Def 21.3.2. *Вес остова – сумма весов рёбер.*

Задача: дан граф, найти остов минимальной стоимости.

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

21.3.1. 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: $\mathcal{O}(sort(E) + (V+E)\alpha)$.

21.3.2. 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 21.3.3. Алгоритм Прима = алгоритм Дейкстры с операцией «`lastEdge`» вместо «`+`».

21.3.3. Алгоритм Борувки

Вес ребра w_e поменяем на пару $\langle w_e, e\rangle$. Теперь все веса различны.

Фаза алгоритма: для каждой вершины выберем минимальное по весу ребро.
Выбранные рёбра не образуют циклов (от противного: пользуемся тем, что веса различны).
Добавим все выбранные рёбра в остов. Сожмём компоненты связности. Удалим кратные рёбра.

Алгоритм: пока граф не сожмётся в одну вершину выполняем очередную фазу алгоритма.

Удалять кратные рёбра будем сортировкой рёбер.
Рёбра – пары чисел от 1 до $V \Rightarrow$ цифровая сортировка справится за $\mathcal{O}(V + E)$.

Lm 21.3.4. Время работы алгоритма Борувки $\mathcal{O}((V + E) \log V)$.

Доказательство. Фаза работает за $V + E$. Число вершин уменьшится в два раза. ■

Lm 21.3.5. Время работы алгоритма Борувки $\mathcal{O}(E + V^2)$.

Доказательство. $\mathcal{O}(V + E)$ нужно на первое удаление кратных рёбер.
Теперь всегда $E \leqslant V^2 \Rightarrow$ алгоритм работает за $\mathcal{O}(V^2 + (\frac{V}{2})^2 + (\frac{V}{4})^2 + \dots) = \mathcal{O}(V^2)$. ■

Lm 21.3.6. Время работы алгоритма Борувки $\mathcal{O}(E \cdot \lceil \log V^2/E \rceil)$.

Доказательство. V^2 за фазу уменьшается в 4 раза $\Rightarrow$ за $\lceil \log_4 V^2/E \rceil$ фаз V^2 станет не больше исходного E. По предыдущей лемме оставшаяся часть алгоритма отработает за $\mathcal{O}(E)$. ■

21.3.4. Сравнение алгоритмов

Асимптотика: из изученных нами куч и сортировок получаем, что лучше всего
Прим за $\mathcal{O}(E + V \log V)$, Прим за $\mathcal{O}(E \log_{E/V} V)$ и Краскал за $\mathcal{O}(E \log \log C)$.

- Краскал просто пишется, быстро работает.
- На очень плотных графах Прим за V^2 быстрее.
- Алгоритм Борувки мы оценили лишь сверху. В среднем по тестам он даёт линию.

Замечание 21.3.7. На всех планарных графах Борувка работает за $\mathcal{O}(V+E)$.

21.3.5. Лемма о разрезе и доказательства

Доказательства трёх алгоритмов очень похожи, все они опираются на *лемму о разрезе*. Сперва, докажем Прима ручками, затем обобщим док-во до леммы и всё сведём к лемме.

• Доказательство алгоритма Прима.

Пусть A – текущий остов. Пусть T – MST, до которого можно дополнить A: $A \subseteq T$. Пусть Прим хочет добавить ребро e и $e \notin T$, рассмотрим концы e: a_e, b_e и путь p в T между ними. e выходит из A: $a_e \in A, b_e \notin A \Rightarrow$ в пути p $\exists$ ребро e_1, выходящее из A. Прим выбрал e: w_e – минимальное исходящее из $A \Rightarrow w_{e_1} \geqslant w_e \Rightarrow$ остов $T - e_1 + e$ тоже минимальный и $A + e \subseteq T\, 0\, e_1 + e$.

Lm 21.3.8. 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 21.3.9. 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.

• Доказательство алгоритма Борувки.

$\forall v$ ребро минимальное для вершины v можно добавить. Разрез $\langle \{v\}, V \setminus \{v\}\rangle$.
Почему все эти рёбра можно добавить одновременно? Верно обобщение леммы о разрезе.

Lm 21.3.10. Пусть $\forall e \in D$ ребро e можно добавить в остов X по лемме о разрезе, и множество $X \cup D$ не содержит циклов, тогда $\exists$ минимальный остов, содержащий $X \cup D$.

Для нашего конкретного случая подойдут следующие разрезы.

Рёбра, которые хочет добавить Борувка, разобьём на компоненты связности. Каждая компонента – дерево, рёбра компоненты будем добавлять по одному от корня к листьям. Добавляя очередное ребро ведущее в лист v, используем разрез $\langle \{v\}, V \setminus \{v\}\rangle$ и лемму о разрезе. ■

21.4. (*) Алгоритм Йена

Мы научились искать кратчайший простой путь, MST.
Идея Йена помогает искать ещё и k-й минимальный (простой путь, остовное дерево, поток...)
Оригинальный алгоритм ищет именно k-й путь.

Алгоритм. Запустим Дейкстру, получим кратчайший путь p_1.
Второй простой путь p_2 отличается от p_1, переберём, совпадающий префикс путей, запретим ходить по следующему ребру, запустим Дейкстру. Итого p_2 мы нашли за $|p_1|$ запусков Дейкстры.

Чтобы найти p_3, заметим, что при поиске p_2 у нас было несколько кандидатов вида $\langle P, E, W\rangle$ – множество путей, у которых начало P, после P запрещено ходить по рёбрам из E (пока $|E| = 1$, в дальнейшем будет больше), Дейкстра из множества $\langle P, E\rangle$ нашла кратчайший, его вес – W.

p_2 – минимум по W_i, а также минимум в своём множестве путей $\langle P_i, E_i\rangle$. Оствшиеся в $\langle P_i, E_i\rangle$ пути разделятся на множества такого же вида – пройдём P_i, зафиксируем ещё несколько (возможно, 0) вершин по p_2 и свернём (запретим следующее ребро из p_2).

Нужно хранить кучу k минимальных по W_i множеств $\langle P_i, E_i\rangle$ и k раз "выбирать следующий минимальный путь": вытаскивать из кучи минимум, затем $\leqslant V$ раз запускать Дейкстру.
Итого: 2-й путь нашли за $\mathcal{O}(V \cdot \texttt{Dijkstra})$, а k-й за $\mathcal{O}(kV \cdot \texttt{Dijkstra})$.

21.5. (*) Алгоритм Эпштейна для k-го пути

[Eppstein'98]

Задача: дан орграф, на котором мы умеем быстро строить дерево кратчайших путей (например, $w_e \geqslant 0$ или ациклический), найти среди всех (необязательно простых) путей из s в t (концы фиксированы) k-ую статистику по суммарному весу рёбер.

Возможное применение: возьмём стандартную линейную динамику «дойти из 1 в n, переходя каждый раз вперёд или на 2, или на 3, собрать максимальную сумму». Попросим найти из всего множества способов дойти из 1 в n сразу k максимумов. Решение: строим ациклический граф динамики, на нём Эпштейна.

Решение за $\mathcal{O}((m+n+k)\log(mk))$.
1. Построим дерево кратчайших путей из t по обратным рёбрам. $d[v]$ — расстояние до t, p_v – отец в дереве кратчайших путей.
2. $\forall v$ будем поддерживать кучу $h[v]$ — возможные ответвления от кратчайшего пути $v \rightsquigarrow t$.
$h[v] = h[p_v] \cup \{(d[x]+w_e) - d[v] \mid e\colon v \to x\}$, в $h[p_v]$ уже лежат ответвления сделанные дальше по пути, $(d[x]+w_e) - d[v]$ это то, на сколько вес ответвления больше веса кратчайшего пути.
3. Изначально множество кандидатов $H = h[s]$, далее k раз достаём очередной минимум и добавляем новых кандидатов.

```
d[], p[] = Dijkstra(t)

for v: // динамика, перебираем вершины от корня дерева
    h[v] = h[p[v]].copy() // используем персистентность
    for e in g[v]:
        h[v].add({d[b[e]]+w[e] - d[v], e}) // запомним Δ веса и номер ребра
H = h[s]

for k:
    <Delta,e> = H.extractMin() // e : a[e] ->b[e]
```

```
11      H = H.merge(h[b[e]].c(Delta)) // увеличить все элементы на Delta
```

В качестве $h[v]$ нам нужна структура данных, которая быстро умеет

`add`, `extractMin`, `merge`, `copy`, `incAll`

Подойдёт leftist heap (левацкая куча) и хранение числа-увеличителя.

Лекция #22: Жадность и приближённые алгоритмы

22-я пара, 2024/25

> The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind. And greed-mark my words-will save not only Teldar Paper but the other malfunctioning corporation called the USA.
>
> Gordon Gekko [Michael Douglas], *Wall Street (1987)*

> There is always an easy solution to every human problem-neat, plausible, and wrong.
>
> H. L. Mencken, *"The Divine Afflatus", New York Evening Mail (November 16, 1917)*

22.1. Хаффман

Для начала заметим печальный факт: идеального архиватора не существует.

Lm 22.1.1. $\nexists$ архиватора f и целого n: $\forall x\ (|x| = n \Rightarrow f(x) < n)$

Доказательство. Пусть $\exists$. Входов 2^n, выходов $1 + 2 + \cdots + 2^{n-1} < 2^n \Rightarrow f$ – не инъекция. ■

Принцип сжатия текста: сопоставить каждому символу битовый код.
Кодирование: записать сами коды и выписать общую последовательность бит.
Раскодирование: жадно откусить от кода 1 символ.
Когда раскодирование однозначно? iff коды *безпрефиксные*.
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 $\Rightarrow \mathcal{O}(sort(\Sigma) + \Sigma)$.

Теорема 22.1.2. Алгоритм корректен.

Доказательство. Если упорядочить символы $len_i \searrow$, то $cnt_i \nearrow$ (иначе можно поменять местами два кода и уменьшить F). У каждой вершины дерева ровно два ребёнка (если ребёнок только один, код можно было бы укоротить). Значит, существует два брата, две самых глубоких вершины, и это ровно те символы, у которых cnt минимален. Осталось сделать переход по индукции к задаче того же вида: заменим два самых глубоких символа-брата на их отца, и будем искать код отца. Длина кода отца домножается как раз на сумму частот братьев. ■

22.1.1. Хранение кодов

Есть несколько способов сохранить коды.

1. Сохранить не коды, а частоты. Декодер строит по ним коды так же, как кодер.
2. Рекурсивный код дерева: если лист, пишем бит 0 и символ, иначе пишем бит 1. Длина = $2 + \lceil \log \Sigma \rceil$ бит на каждый ненулевой символ.
3. Сделать dump памяти = `sizeof(Node) * (2Σ - 1)` байт.

Какой бы способ мы не выбрали, можно попробовать рекурсивно сжать коды Хаффманом. Чтобы рекурсия не была бесконечной добавим в код один бит – правда ли, что сжатый текст меньше. Если этот бит ноль, вместо кода запишем исходный текст.

22.2. Компаратор и сортировки

Решим несколько простых задач.

1. Сортировка по времени
Даны n заданий, у каждого есть время выполнения t_i. К моменту T выполнить как можно больше заданий. Решение: выполняем в порядке возрастания t_i. Доказательство: не важен порядок $\Rightarrow$ выполняем в порядке возрастания t_i. Пусть на первом месте не t_{min}, поставим $t_{min} \Rightarrow$ сумма уменьшилась $\Rightarrow$ сумма всё ещё $\leqslant T \Rightarrow$ перейдём по индукции к аналогичной задаче, но без задания t_{min} и временем $T - t_{min}$

2. Сортировка по дедлайну
Даны n заданий, у каждого есть время выполнения t_i и дедлайн момента сдачи d_i. Нужно успеть выполнить все задания. Решение: выполнять в порядке возрастания дедлайна. Способ придумать решение: посмотрим, кого мы можем выполнить последним? Любого, у кого $d_i \geqslant \sum_j t_j \Rightarrow$ выполним в конце d_{max}. Доказательство: можем выполнить множество заданий I, выполнив последним $d_{last} \Rightarrow$ (можем выполнить $I_1 = I \backslash \{d_{max}\}) \wedge (d_{max} \geqslant d_{last} \geqslant \sum_j t_j \Rightarrow$ можем выполнить последним $d_{max}) \Rightarrow$ возьмём по индукции решение для I_1, допишем к нему d_{max}.
На практике разобраны похожие задачи – с решением "сортировка по сумме".

Ход мыслей в (1) и (2): как только мы показали, кого ставить на первое/последнее место, отсортируем в таком порядке, получим решение за $\mathcal{O}(sort)$.

3. Сортировка по частному

Даны файлы размеров s_i и частоты обращения к ним cnt_i. Лента – хранилище данных, позволяющее к файлу, начинающемуся на позиции pos обратиться за время pos.
Расположить файлы на ленте, минимизировать суммарное время обращения $\sum_i pos(i)cnt_i$.
Решение для случая $s_i = 1$: сортируем по убыванию cnt_i.
Решение для случая $cnt_i = 1$: сортируем по возрастанию s_i.
Общий случай: сортируем по убыванию $\frac{cnt_i}{s_i}$. Чтобы придумать такое решение предположим, что задача решается сортировкой $\Rightarrow$ у сортировки есть компаратор и компаратор решает задачу для $n = 2 \Rightarrow$ чтобы придумать компаратор полезно исследовать решение при $n = 2$. В порядке $\{1, 2\}$ получаем $cnt_2 s_1$, в порядке $\{2, 1\}$ получаем в $cnt_1 s_2 \Rightarrow$ `less(1,2)`: $cnt_2 s_1 < cnt_1 s_2 \Leftrightarrow \frac{cnt_1}{s_1} > \frac{cnt_2}{s_2}$.
Решение придумано, доказываем: для оптимального ответа верно, что отличимые компаратором `less` элементы идут в том же порядке, что задаёт `less`; осталось понять, что неотличимые ($\frac{cnt_1}{s_1} = \frac{cnt_2}{s_2}$) можно расставить в любом порядке, ответ $\sum_i pos(i)cnt_i$ не изменится.

Алгоритм 22.2.1. *Перестановочный метод: π – искомая перестановка, $F(\pi) \to \min$.*
В поисках оптимальной π пробуем поменять соседние элемента: $F(\pi) \lesseqgtr F(\pi[swap(i, i{+}1)])$.
Полученное условие используем, как компаратор.
Часто это условие совпадает с более простым: $\texttt{less(i, j)} = (F([i, j]) < F([j, i]))$

Необходимые условия к компаратору
Когда можно пользоваться перестановочным методом Algo 22.2.1 из (3)?
Проще сформулировать это в терминах компаратора `lessOrEquals`: он должен задавать линейный предпорядок (total preorder), то есть быть транзитивным и любые два элемента должны быть сравнимы хотя бы в одну сторону.

Если говорить в терминах отношения компаратора `less`, то он должен задавать strict weak ordering, условия на который немного сложнее: он должен быть антирефлексивным, транзитивным, и несравнимость должна быть транзитивной. Поясним последнее: если $\neg less(a, b)$ и $\neg less(b, a)$, то будем говорить, что a и b несравнимы и писать $a \sim b$. Такое отношение $\sim$ должно быть транзитивным.
Проблемы обычно только с транзитивностью и транзитивностью несравнимости. Чтобы показать транзитивность в (3), мы переписывали компаратор: $(cnt_2 s_1 < cnt_1 s_2) \to (\frac{cnt_1}{s_1} > \frac{cnt_2}{s_2})$.

Чтобы сортировка по такому компаратору минимизировала $F(\pi)$, достаточно выполнения следующего условия: для всех перестановок π и для каждого i, если $\texttt{lessOrEquals}(\pi_\texttt{i}, \pi_\texttt{i+1})$, то $F(\pi) \le F(\pi')$, где π' получена из π операцией $\pi_i \leftrightarrow \pi_{i+1}$. (Упражнение: проверьте, что этого действительно достаточно.)

22.2.1. Задача про 2 станка

Задача: даны 2 станка, n деталей, время обработки деталей на первом станке a_i и время обработки деталей на втором станке b_i. Выбрать порядок обработки деталей π, обработать каждую сперва на первом, затем на втором станке. Минимизировать время окончания работ.

Решение: разбить на $P_1 = \{i \colon a_i \leqslant b_i\}$ и $P_2 = \{i \colon a_i > b_i\}$, отсортировать P_1 по возрастанию a, P_2 по убыванию b. $\pi = P_1 P_2$. Оригинальная статья (S.M.Johnson'1953).

- (*) **Вспомогательные рассуждения**

Время окончания выполнения на первом станке $A_i = A_{i-1} + a_{\pi_i}$
Время окончания выполнения на втором станке $B_i = \max(A_i, B_{i-1}) + b_{\pi_i}$

Пусть x_i – ожидание второго станка перед обработкой i-й детали $\Rightarrow B_k = \sum_{i\leqslant k}(x_i + b_i)$.

Обозначим $F(k) = \sum_{i\leqslant k} x_i$. $F(n)$ – суммарный простой второго станка. Задача: $F(n) \to \min$.

При этом $x_k = \max(0, A_k - B_{k-1}) = \left(\sum\limits_{i=1}^{k} a_i - \sum\limits_{i=1}^{k-1} b_i\right) - \sum\limits_{i=1}^{k-1} x_i = D_k - F_{k-1}$, где $D_i = \sum\limits_{j\leqslant i} a_i - \sum\limits_{j<i} b_j$.

Докажем по индукции, что $F(k) = \max(0, \max\limits_{i=1..k} D_i)$.

База: $k = 0$. Переход: $x_k > 0 \Rightarrow x_k = D_k - F_{k-1} \wedge F_k = D_k$; $x_k = 0 \Rightarrow F(k) = F(k-1) \wedge D_k \leqslant F_{k-1}$.

• (*) **Корректность алгоритма**

Осталось показать, что для минимизации $F(n) = \max_{i=1..n} D_i$, нужно 3 вещи:
1. Если рядом стоят $a_i > b_i$ и $a_{i+1} \leqslant b_{i+1}$, можно поменять местами, $F(n) \searrow$
2. Если рядом стоят $a_i \leqslant b_i$ и $a_{i+1} \leqslant b_{i+1}$: $a_i > a_{i+1}$, можно поменять местами, $F(n) \searrow$
3. Если рядом стоят $a_i > b_i$ и $a_{i+1} > b_{i+1}$: $b_i < b_{i+1}$, можно поменять местами, $F(n) \searrow$

• **Неверное решение**

Выведем компаратор, как в Algo 22.2.1. $F(1,2) = a_1 + \max(a_2, b_1) + b_2 = (a_1+a_2+b_1+b_2) - \min(a_1, b_2)$.
Получили "`less(i, j): return min(a[i], b[j]) > min(a[j], b[i])`".
Получили транзитивное отношение меньше, которое не является строгим линейным порядком.
Если его передать функции `std::sort`, получится почти всегда рабочий алгоритм.
Стресс-тестом можно найти контрпримеры. Подробнее на codeforces и на github.

• **Внешние ссылки**

Можно ознакомиться с e-maxx и ИТМО-конспектами. По ссылкам рассуждения (доказательства) не полны. При этом по обоим ссылкам рабочая реализация (хоть и переусложнённая).

22.2.2. Выбор максимума

Пример задачи («сортировка по дедлайну»): есть n задач, у каждой есть время, требуемое для выполнения t_i и дедлайн d_i, к которому нужно завершить выполнение задачи.

Мы уже умеем проверить, можно ли выполнить все задачи: отсортировать по d_i и выполнять в таком порядке. Теперь мы хотим успеть выполнить максимальное количество задач.

Замечание 22.2.2. Полученный в этом разделе метод часто позволяет перейти от «выполнить все» сортировкой за $\mathcal{O}(n \log n)$ к «выполнить как можно больше» жадностью за $\mathcal{O}(n \log n)$.

Решение за $\mathcal{O}(n^2)$: если мы зафиксируем множество заданий, которое хотим выполнить, выполнять их нужно в порядке $d_i\uparrow$ $\Rightarrow$ отсортируем по d_i и напишем динамику `T[i,k]` – минимальное суммарное время, нужное, чтобы выполнить какие-то k заданий из i первых. Переходы:

1. `relax(T[i+1,k], T[i,k]);`
2. `if d[i] >= T[i,k] + t[i] then relax(T[i+1,k+1], T[i,k] + t[i]);`

Решение за $\mathcal{O}(n \log n)$: будем жадно брать в порядке $d_i\uparrow$, если не получается взять, попытаемся замениться на работу с максимальным t_j.

```
1.  sumTime = 0
2.  for пары ⟨d_i, t_i⟩ в порядке d_i↑:
3.      if sumTime + t_i ⩽ d_i then берём: heap.add(t_i), sumTime += t_i
4.      else if t_i < heap.max then делаем замену:
5.          sumTime += t_i − heap.max, heap.delMax, heap.add(t_i)
```

В строке 4, когда условие выполнено, верно и $d_i \geqslant d_j \Rightarrow$ мы лучше и по t, и по $d \Rightarrow$ после замены порядок выполнения оставим прежним, а себя вставим ровно на место удалённого.

• **Корректность решения за $\mathcal{O}(n \log n)$**

Пусть на момент рассмотрения пары $\langle d_i, t_i \rangle$ жадность считает, что нужно выполнять задания $t_{j_1} \leqslant t_{j_2} \leqslant \ldots \leqslant t_{j_m}$. Мы специально записали эти задания в порядке $t_j\uparrow$. Докажем по индукции, что, $\forall k$ `T[i,k]`$= t_{j_1} + \cdots + t_{j_k}$, более того множество этих k предметов есть $\{j_1, \ldots, j_k\}$.

База: $i = 0 \Rightarrow m = 0$.
Переход: пользуемся корректностью динамики `T[i+1,k]` $=$ `T[i,k]` $\vee$ `T[i,k-1]` $+ t_i$.
Заметим, что для всех таких k, что $t_{j_k} \leqslant t_i$ оптимально `T[i,k]` и наше решение как раз его выберет. А теперь заметим **TODO**, что для всех таких k, что $t_{j_k} > t_i$ второе возможно и оптимальнее первого.
Наше решение его и выберет. Для удобства считаем $t_{j_{m+1}} = +\infty$.
В сумме получаем, что $\forall k$ `T[i,k+1]` – префикс $t_{j_1} \leqslant t_{j_2} \leqslant \ldots \leqslant t_i \leqslant \ldots \leqslant t_{j_m}$.

Замечание 22.2.3. То же применимо к задаче про спортсменов:
$s_i + m_i \geqslant s_j + m_j$, $m_i < m_j \Rightarrow s_i > s_j \Rightarrow$ можно вместо j вставить i на то же место $\Rightarrow$ наше рассуждение (доказательство) выше работает и для этой задачи.

22.3. Жадность для гамильтонова пути. Варнсдорф

Варнсдорф решил задачу гамильтонова цикла для шахматного коня. Правило Варнсдорфа говорит: нужно идти в вершину минимальной остаточной степени, а при равных прижиматься к краю (то ли по $\min(dx, dy) \to \min$, то ли $|dx| + |dy| \to \min$). В какой вершине начать: в случайной точно подойдёт.

Варнсдорф, конечно, решал для 8×8, но его решение подходит и для $n \times n$.

Как это счастье применить к общему случаю?
1. Жадно идти в вершину минимальной остаточной степени!
2. Можно хитрее: random walk (тоже жадность), который пытается жадно идти в вершину минимальной остаточной степени, а из равных выбирает случайную.
3. Можно ещё хитрее: random walk (всё ещё жадность), который выбирает случайную вершину куда идти с вероятностью обратно пропорциональной остаточной степени. $p_v = \frac{1}{deg_v}$.

Лекция #23: Приближённые алгоритмы

23-я пара, 2024/25

Как мы знаем, некоторые задачи NP-трудны, вряд ли их возможно решить за полином.
Но на практике решать их всё равно нужно. В таких случаях часто ищут приближённое решение, причём ищут за полиномиальное или даже линейное время.

Def 23.0.1. *Алоритм M называется α-оптимальным (α-OPT) для задачи P, если на любом входе для P ответ, выдаваемый M, не более, чем в α раз хуже оптимального.*

23.1. Коммивояжёр

Задача коммивояжёра заключается в поиске гамильтонова цикла минимального веса.
Решим задачу в случае полного графа с неравенством $\triangle$.

Lm 23.1.1. $\forall \alpha > 1 \nexists$ полиномиального α-OPT алгоритма для задачи коммивояжёра.

Доказательство. Если такой существует, то он в частности ищет гамильтонов путь. ■

Замечание 23.1.2. Любой граф можно сделать полным, добавив вместо отсутствия ребра ребро веса $+\infty$. А вот неравенство $\triangle$ – важное свойство.

23.1.1. 2-OPT через MST

Оптимальный маршрут коммивояжёра = цикл = путь + ребро = остовное дерево + ребро $\Rightarrow$ MST $\leqslant$ OPT. Возьмём MST графа. Каждое ребро заменим на два ориентированных (туда и обратно), получили эйлеров орграф (число входящих равно числу исходящих).

Возьмём эйлеров обход, его вес в два раза больше MST. Если в полученном пути некоторая вершина встречается второй раз, удалим её второе вхождение.

По нер-ву $\triangle$ вес пути только уменьшится. Получили маршрут коммивояжёра веса $\leqslant 2 \cdot \text{MST}$.

23.1.2. 1.5-OPT через MST (Кристофидес)

Чтобы превратить MST в эйлеров граф, добавим к нему M – совершенное паросочетание минимального веса на вершинах нечётной степени. $w(MST) \leqslant \text{OPT}$, докажем что $w(M) \leqslant \frac{1}{2}\text{OPT}$.
$\exists X$ – маршрут коммивояжёра только на нечётных вершинах. По нер-ву $\triangle$ $w(X) \leqslant \text{OPT}$.
Цикл X можно разбить на два паросочетания – A (чётные рёбра) и B (нечётные рёбра).

$$w(A) + w(B) = w(X) \Rightarrow \min(w(A), w(B)) \leqslant \tfrac{1}{2} w(X) \leqslant \tfrac{1}{2}\text{OPT}$$

Lm 23.1.3. $\exists$ полиномиальный алгоритм для поиска паросочетания минимального веса в произвольном графе. Кстати, умеют это делать весьма быстро, за $\mathcal{O}(E\sqrt{V}\log(VN))$. PDF 2017.

23.2. Set cover

Задача: даны множества $A_1, A_2, \ldots, A_m$, хотим покрыть $U\colon |U| = n$ минимальным числом A_i.
Шаг жадности: добавить в ответ множество A_i, покрывающее максимальное число ещё не покрытых элементов U.

Упражнение 23.2.1. Придумать реализацию за $\mathcal{O}(n + \sum |A_i|)$.

Теорема 23.2.2. Получили $(\ln n)$-OPT алгоритм.

Доказательство. Пусть оптимальный ответ равен k.
За один шаг жадность покроет $\geqslant \frac{1}{k}n$ элементов $U \Rightarrow$
размер U уменьшится в $\frac{k-1}{k}$ раз $\Rightarrow$ за t шагов размер U уменьшится в $(\frac{k-1}{k})^t$ раз.
Осталось взять t: $(\frac{k-1}{k})^t < \frac{1}{n} \Leftrightarrow (\frac{k}{k-1})^t > n \Leftrightarrow t > \frac{1}{\ln k - \ln(k-1)} \ln n > k \ln n$. ■

Упражнение 23.2.3. Найдите $(\ln n)$-OPT приближение взвешенной задачи – вес A_i равен w_i.

23.3. Рюкзаки

Будем пользоваться англоязычными обозначениями задач:

1. *Partition* — разбиение множества предметов на два равных по весу.
   Даны a_i, выбрать I: $\sum_{i\in I} a_i = \frac{1}{2}\sum a_i$.
2. *Knapsack* — поместить в рюкзак размера S предметы максимальной суммарной стоимости.
   Даны $a_i > 0$, $p_i > 0$ и S, выбрать $0 \leqslant x_i \leqslant 1$: $\sum a_i x_i \leqslant S$, при этому $\sum p_i x_i \to \max$.
   $x_i \in \mathbb{R} \Rightarrow$ задача называется непрерывным рюкзаком, решается сортировкой по $\frac{p_i}{a_i}$.
   $x_i \in \mathbb{Z} \Rightarrow$ получили классический NP-трудный дисркетный рюкзак (knapsack).
3. *Bin packing* – уложить предметы в минимальное число рюкзаков размера 1.
   Даны $0 < a_i \leqslant 1$, выбрать $I_1, I_2, \dots I_k$: $\cup I_j = \{1..n\} \wedge \forall j \sum_{i\in I_j} a_i \leqslant 1$. При этом $k \to \min$.

• NP-трудность

Базовой NP-трудной у нас будет задача partition.
Knapsack труден, так как можно взять все $p_i = a_i$, $S = \frac{1}{2}\sum a_i$ и решить partition.
Bin packing труден, так как если научимся отличать ответы 2 и 3, решим partition.

Следствие 23.3.1. $\forall \varepsilon > 0$ для bin packing $\nexists (1.5-\varepsilon)$-OPT алгоритма.

23.4. Схемы приближений

Для некоторых задач доступны сколь угодно точные приближения за полиномы. PTAS/FPTAS.

Def 23.4.1. *Алгоритм $A(\varepsilon, x)$ – PTAS (polynomial-time approximation scheme) для задачи M, если $\forall \varepsilon > 0$ он отработает за полином от $|x|$ и выдаст решение F такое, что:*

$$\begin{cases} F \geqslant (1-\varepsilon)OPT & \text{если } M \text{ – задача максимизации} \\ F \leqslant (1+\varepsilon)OPT & \text{если } M \text{ – задача минимизации} \end{cases}$$

Def 23.4.2. *FPTAS (fully PTAS) обязывает A работать за полином и от $|x|$, и от ε^{-1}.*

23.5. Knapsack

Здесь можно почитать про PTAS и FPTAS для knapsack.
Задача: даны $a_i > 0, p_i > 0$, выбрать $0 \leqslant x_i \leqslant 1$: $\sum a_i x_i \leqslant S \wedge \sum p_i x_i \to \max$.
Если $x_i \in \mathbb{R}$ задача называется непрерывной и решается жадно сортировкой по $\frac{p_i}{a_i}$.
Если $x_i \in \mathbb{Z}$ задача называется дискретной или 0-1 и жадное решение ничего не приближает.
Пример: $a_1 = 1$, $p_1 = 2$, $a_2 = S$, $p_2 = S \Rightarrow$ жадность возьмёт первый предмет, оптимально брать второй. Ошибка в $S/2$ раз $\Rightarrow$ не ограничена.

Замечание 23.5.1. Во всех решениях дальше будем считать, что мы сразу выкинули предметы, которые даже по отдельности не помещаются в рюкзак, то есть у которых $x_i > S$.

На практике разобрали 2-приближение: выбираем лучшее из жадности и первого предмета, который уже не влез.
Но можно приблизить и PTAS, и FPTAS.

• **PTAS:** переберём, какие k предметов возьмём за $\binom{n}{k}$, оставшиеся наберём жадностью.

Теорема 23.5.2. Мы получили $(1-\frac{1}{k})$-OPT решение, работающее за $\mathcal{O}(n^{k+1})$

Доказательство. Обозначим оптимальное решение I. Рассмотрим момент, когда мы угадали k max по p_i предметов, присутствующих в I: $i_1 \dots i_k$. Посмотрим теперь, как жадность лихо добавляет лучшие по $\frac{p_i}{a_i}$ предметы и остановимся на первом событии «жадность не взяла предмет $j \in I$». В этот момент остановим жадность, полученное решение обозначим I'. $I' \cup \{j\}$ не хуже OPT: $i_1, \dots, i_k, j$ – общая часть, остальные предметы I' мажорируют I по $\frac{p_i}{a_i}$. Осталось заметить, что $p_j \leqslant \frac{1}{k}(p_{i_1} + \dots + p_{i_k}) \leqslant \frac{1}{k}\text{OPT} \Rightarrow P(I') \geqslant (1 - \frac{1}{k})P(I)$, где $P(I) = \sum_{i \in I} p_i$. ■

Замечание 23.5.3. На самом деле, перебирать надо и меньшие множества тоже, поскольку в ответе может быть меньше, чем k предметов.

• **FPTAS:** возьмём псевдополиномиальную динамику `minWeight[i,sumP]`, и чтобы она работала за строгий полином. сделаем все $p_i \in \{0, 1, \dots, k\}$: $p'_i = \lfloor \frac{p_i}{\max_j p_j} k \rfloor = \lfloor \frac{1}{x} p_i \rfloor$, где $x = \frac{\max_j p_j}{k}$.
Решение работает за $\mathcal{O}(n^2 k)$, оценим ошибку.

$$\text{Пусть } P(set) = \sum_{i \in set} p_i, \text{ а } P'(set) = \sum_{i \in set} p'_i = \sum_{i \in set} \lfloor \frac{p_i}{x} \rfloor.$$

Сразу заметим, что $P'(set) \leqslant \frac{P(set)}{x} \leqslant P'(set) + n$, так как мы теряем на округлении, но не более, чем единицу на каждый предмет.
Пусть *opt* это оптимальное решение, а мы нашли *sol*. Тогда $P'(sol) \geqslant P'(opt)$, так как мы нашли самое лучшее решение в профитах p'.

$$P(sol) \geqslant xP'(sol),\ P'(sol) \geqslant P'(opt),\ P'(opt) \geqslant \frac{P(opt) - nx}{x}$$

$$\Longrightarrow\ P(sol) \geqslant xP'(sol) \geqslant xP'(opt) \geqslant P(opt) - nx$$

Получили решение, которое хуже не более чем на nx. Хотим: $nx \leqslant \varepsilon P(opt)$.
Знаем, что $P(opt) \geqslant \max_j p_j$. Выберем x, так что $\varepsilon(\max_j p_j) \geqslant nx$, тогда получаем требуемое.

$$x \leqslant \frac{\varepsilon(\max_j p_j)}{n} \Longrightarrow\ k \geqslant \frac{n}{\varepsilon}$$

Замечание 23.5.4. Аналогичная динамика `maxProfit[i,sumW]` не дала бы нам хорошее приближение: чтобы решение оставалось подходящим веса нужно округлять вверх, после минимального округления вверх предметы могут перестать влезать ⇒ ответ ухудшился в 2 раза.

23.6. Partition

Задача NP-трудна. Эквивалентная ей – subset sum.
Оптимизационная версия partition – *balanced partition*: разбить на два максимально близких по сумме множества.
Обе задачи мы умеем решать за $\mathcal{O}^*(2^{n/2})$ методом meet-in-the-middle.

$\nexists$ приближённых по величине $D = |\sum a_i - \sum b_i|$ решений (NP-трудно отличить $D = 0$ от $D > 0$). Но делать что-то надо, поэтому строят приближённые решения по величине $\max(A, B)$, $A = \sum a_i$, $B = \sum b_i$.

Тривиальное жадное решение за $\mathcal{O}(n \log n)$: *перебирать предметы в порядке $a_i \downarrow$, класть очередной предмет в множество с меньшей суммой.*

Lm 23.6.1. При $n \leqslant 4$ решение даст оптимальный ответ

Доказательство. Пусть $a_1 > a_2 > a_3 > a_4$, тогда варианты $a_1 + a_2$ и $a_1 + a_3$ точно хуже $\max(a_1 + a_4, a_2 + a_3)$, а наша жадность за первые три хода как раз получит $\{a_1\}, \{a_2, a_3\}$, четвёртое положит оптимально. ■

Пусть $n \geqslant 5$. Будем следить за величиной $D = |A - B|$. Заметим, что если после обработки $k \geqslant 4$ предметов D стало не больше a_5, то и дальше оно будет всегда не больше a_5. Тогда уменьшить величину $\max(A, B)$ можно не больше чем на $d = \frac{1}{2}a_5$. Обозначим $X = \max(A, B)$, $X \geqslant 3a_5 \Rightarrow d \leqslant \frac{1}{6}X \Rightarrow X \leqslant \text{OPT} + \frac{1}{6}X \Rightarrow X \leqslant \frac{6}{5}\text{OPT}$.
Однако, могло случиться так, что D так и не стало достаточно маленьким (после обработки четвёртого предмета и дальше). В этом случае все предметы, начиная с пятого, мы клали в одну и ту же коробку, пусть это B, и всё равно $B_n < A_n$. Тогда $\max(A_n, B_n) = A_n = A_4 = \max(A_4, B_4)$, и при этом для любого размещения предметов $A_n \geqslant A_4$. Мы знаем, что A_4 было минимально, поэтому A_n тоже минимально. То есть в этом случае жадность находит оптимальный ответ.
Таким образом, наша жадность $\frac{6}{5}$-оптимальна.

Замечание 23.6.2. Более аккуратным анализом можно показать, что это $\frac{7}{6}$-OPT решение.

PTAS схема: переберём, куда кладём $2k$ максимальных по весу предметов, остаток разложим, следуя предыдущей жадности. Получили $(1 + \frac{1}{k})$-OPT решение. Время $\mathcal{O}(2^k n) \Rightarrow$ не FPTAS.

Упражнение 23.6.3. Доказательство осталось на практику

23.6.1. Алгоритм Кармаркара-Карпа (LDM)

[wiki]

Задача partition: разбить предметы на два рюкзака, минимизировать $D = |\sum a_i - \sum b_i|$. Мы уже умеем решать её жадностью Lm 23.6.1.

Алгоритм Кармаркара-Карпа (1982)

```
1 def solve(a):
2     while size(a) > 1:
3         x = a.extractMax()
4         y = a.extractMax()
5         a.push(x - y)
6     return a.extractMax()
```

Описанный алгоритм ещё называют LDM (largest difference method). Он даёт $\frac{7}{6}$-приближение (как и жадность).

Теперь посмотрим на величину $D = |A - B|$. На n случайных равномерно распределённых величинах в $\mathbb{R} \cap [0, 1]$ жадность даст $E(D) = \Theta(\frac{1}{n})$, Кармаркар-Карп $n^{-\Theta(\log n)}$. Ту же асимптотику

матожидания разности Кармаркар-Карп даст на любых независимых одинаково распределённых случайных величинах.

[Yakir'1996] – доказательство.
[Mertens'1999], [Boettcher,Mertens'2008] – более глубокий анализ.

Интуиция, почему Кармаркар-Карп даёт $n^{-\Theta(\log n)}$.
Представим a в сортированном порядке. Сейчас там n чисел из $[0,1]$, за первые $\frac{n}{2}$ шагов мы заменим их на разности соседних, получим $\frac{n}{2}$ чисел из $[0, \frac{1}{n} + o(\frac{1}{n})]$.
Повторим процесс $\log n$ раз, последнее число будет $\approx \frac{1}{n} \cdot \frac{1}{n/2} \cdot \frac{1}{n/4} \cdots = n^{-\Theta(\log n)} = 2^{-\Theta(\log^2 n)}$.

23.6.2. (*) Применение для $\langle P, M\rangle$-антихеш-теста

Найдём 2 строки длины n над алфавитом $\{a, b\}$ с равными полиномиальными $\langle P, M\rangle$-хешами.

$hash(s) = (s_0 + s_1 P^1 + s_2 P^2 + \dots) \bmod M$
$hash(s) = hash(t) \Leftrightarrow hash(s) - hash(t) = \sum x_i P^i \bmod M = 0$, где $x_i \in \{0, \pm 1\}$

Итого: нам даны числа $P^i \bmod M$, хотим некоторые из них разбить на два равных рюкзака.
Применяем Кармаркар-Карпа, пока в множестве a не встретится 0.

Оценка чисел в a в конце Кармаркар-Карпа:
$M \cdot 2^{-\Theta(\log^2 n)} \Rightarrow \log\log n = \Theta(\sqrt{\log M})$ возьмём $n = 2^{\Theta(\sqrt{\log M})}$, на практике константа будет 1.5.
Для $M \approx 10^{18}$ имеем $\log M = 64$, $\sqrt{\log M} = 8$, $n = 2^{1.5 \cdot 8} = 4096$.

23.7. Bin packing

[Garey,Johnson'1997]	Оценки на first fit (FFD), best fit (BFD, BBF).
[PTAS]	Более-менее случайная презентация.
[Hoberg'2015]	$OPT + \mathcal{O}(\log OPT \log\log OPT)$ приближение.
[Korf'1999]	Решение задачи на практике.
[Jansen et al.'2010]	FPT-решение за $\mathcal{O}(2^{k \log^2 k} + n)$, ошибающееся $\leqslant$ чем на 1 корзину.

Напомним, формулировку задачи Section 23.3 и, что отличить ответы 2 и 3 уже трудно Cons 23.3.1.
Распределять по k корзинам сложнее, чем по двум (partition).
Динамикой мы умеем решать задачу лишь для $\mathbb{Z}$ весов, и то за $\mathcal{O}(n \cdot W^{k-1})$.

• First fit

Перебираем предметы в произвольном порядке.
Для каждого предмета перебираем корзины, кладём предмет в первую подходящую.

Теорема 23.7.1. First fit – 2-OPT алгоритм для bin packing

Доказательство. Если наш ответ m, то $m-1$ корзина заполнена больше чем на половину. Итого $\sum a_i > \frac{1}{2}(m-1)$. С другой стороны $\mathrm{OPT} \geqslant \sum a_i \Rightarrow 2\mathrm{OPT} > m - 1 \Rightarrow 2\mathrm{OPT} \geqslant m$. ■

First fit ещё можно оценить как $1.7\mathrm{OPT} + 2$.
Конечно, есть более удачные порядки перебора предметов чем "произвольный".
И в теории, и на практике хорошо работает порядок "по убыванию".

Теорема 23.7.2. First fit decreasing (FFD) для задачи bin packing даёт ответ $\leqslant \frac{11}{9}\mathrm{OPT} + 1$

Доказательство. Можно прочесть по ссылкам выше. Кстати, это была PhD Джонсона [Johnson'73]. Чуть более сильный $(\frac{11}{9}OPT + \frac{6}{9})$ результат [Dosa'2007].

При этом в Cons 23.3.1 мы уже говорили, что нет $(1.5-\varepsilon)$-OPT алгоритма.

• Best fit

Эвристика: класть в корзину, в которой после добавления предмета останется поменьше места.

BBF – перебирать при этом предметы в произвольном порядке даёт $1.7\mathrm{OPT} + 1$.
BFD – перебирать при этом предметы в убывающем порядке даёт $\frac{11}{9}\mathrm{OPT} + 1$.

• Практически эффективное решение

Обе жадности уже хороши тем, что работают за $\mathcal{O}(n \log n)$.
Есть чуть менее быстрое, но более эффективное решение:

- зафиксируем число корзин k (если задача минимизации k, сделаем бинпоиск по ответу);
- начинём со случайного распределения предметов по k корзинам
- минимизируем максимальный размер корзины локальными оптимизациями двух типов –
  пытаемся перекинуть предмет из одной корзины в другую;
  пытаемся поменять два предмета в двух корзинах.
- для эффективной реализации второй оптимизации перебираем предметы в порядке $a_i \uparrow$ и пытаемся менять местами только соседние – a_i и a_{i-1}.

```
1 // p[i] - номер корзины по предмету, sumA[j] - суммарный вес в корзине
2 while optimizing // одна фаза оптимизации работает за O(n)
3     min_index <-- номер корзины с минимальным весом
4     for i = 1..n
5         if sumA[p[i]] - a[i] > sumA[min_index]
6             p[i] = min_index, recalc sumA
7     for i = 2..n // a[i] возрастают
8         if sumA[p[i]] - a[i] > sum[p[i-1]] - a[i-1]
9             swap(p[i], p[i-1]), recalc sumA
```

23.8. PTAS для bin packing

Напомним задачу: даны $0 < a_i \leqslant 1$, разбить их на min число корзин размера 1.

• План решения

1. Сперва рассмотрим ситуацию, когда все $a_i \geqslant \varepsilon$ и различных a_i не более K
2. Отбросим ограничение "не более K" (округлим все a_i к одному из K значений).
3. Отбросим ограничение $a_i \geqslant \varepsilon$ (в начале выкинем $a_i < \varepsilon$, в конце жадно добавим).

1 Итак, все $a_i \geqslant \varepsilon$ и различных a_i не более $K \Rightarrow$ в каждую корзину влезает не более $M = \frac{1}{\varepsilon}$ предметов $\Rightarrow$ различных типов корзин не более $R = K^M$. Нам не нужно больше n корзин, есть не более $P = \binom{n+R-1}{R}$ вариантов выбрать n корзин R типов.

Заметим, что $1/\varepsilon, K, M, R$ – константы. Огроменные, но константы.
$P \leqslant \frac{1}{R!}(n+R)^R \Rightarrow P$ – полином от n, переберём все P вариантов, выберем оптимальный.

2 Выберем $K = \lceil \frac{1}{\varepsilon^2} \rceil$. Отсортируем a_i, разобьём их на K групп, в каждой не более $\lfloor n\varepsilon^2 \rfloor$ элементов. Изменим в каждой группе все числа на максимальное, теперь различных $\leqslant K$.
Исходная задача – A, новая – A'. A' поэлементно больше $A \Rightarrow \forall$ распределение для A' является распределением для A. Рассмотрим A'', полученную из A' удалением $\lfloor n\varepsilon^2 \rfloor$ максимальных.

A'' поэлементно меньше $A \Rightarrow k(A'') \leqslant k(A) \Rightarrow k(A') \leqslant k(A'') + \lfloor n\varepsilon^2 \rfloor \leqslant k(A) + \lfloor n\varepsilon^2 \rfloor$.

$\forall i\ a_i \geqslant \varepsilon \Rightarrow OPT \geqslant n\varepsilon \Rightarrow k(A') \leqslant k(A) + \varepsilon \cdot OPT = (1+\varepsilon)OPT$.

3 I – исходная задача, I' – задача без $a_i < \varepsilon$. Мы уже получили решение для I' не хуже $(1+\varepsilon)k(I')$. Добавим, используя любую жадность, например FF, все $a_i < \varepsilon$.
Если ответ не изменился, отлично $k(I') \leqslant k(I) \Rightarrow (1+\varepsilon)k(I') \leqslant (1+\varepsilon)k(I)$.
Если ответ изменился, то все корзины кроме последней заполнены больше чем на $1-\varepsilon \Rightarrow$
$(1-\varepsilon)(k(I')-1)+1 \leqslant \sum_{\text{все}} a_i \leqslant k(I) \ \Rightarrow\ k(I') \leqslant \frac{1}{1-\varepsilon}(k(I)-1)+1 \leqslant (1+2\varepsilon)k(I)+1$.

Замечание 23.8.1.
В части (1) мы получили точное решение.
В части (2) мы получили $(1+\varepsilon)$ приближении, и лишь
В части (3) приближение стало $(1+2\varepsilon)$-OPT $+1$.

23.9. Надстрока

Гипотеза о 2-приближении к надстроке.
Сведение надстроки через TSP.
Простое жадное решение надстроки.
Сведение надстроки к взвешенной версии SetCover.
Всё это в разборе [практики]

23.10. Литература

[Книга по приближённым алгоритмам]. Approximation algorithms, Vazirani '2001.
[Knapsack | PTAS FPTAS]. Лекция от Goeman-a.
[Euclidian TSP | PTAS]. Sanjeev Arora '1998.
[Permanent FPRAS]

Лекция #23: Центроидная декомпозиция

23-я пара, 2024/25

23.11. Построение и минимум на пути дерева

Пусть дано дерево с весами на рёбрах или в вершинах. Рассмотрим три задачи вида «построить структуру данных, которая умеет за $\mathcal{O}(1)$ вычислять минимум...»

1. на пути от любой вершины до корня;
2. на любом пути, проходящем через корень;
3. на любом пути

Для решения первой задачи поиском в глубину по дереву предподсчитаем все минимумы. Вторую задачу сведём к первой, разделив корнем путь на две половины.

Def 23.11.1. *Центроид – вершина, при удалении которой, размеры всех компонент* $\leqslant \frac{n}{2}$.

Чтобы решить третью задачу найдём за $\mathcal{O}(n)$ центроид v (задача с практики); все пути, проходящие через v обработаем, используя (2); удалим v из дерева и рекурсивно построим структуру от оставшихся компонент.

В итоге каждая вершина ровно один раз будет найдена, как центроид. У каждого центроида v есть центроид-предок p_v в дереве рекурсии, глубина в дереве рекурсии d_v и компонента связности $C(v)$, в которой он был найден, как центроид.

Def 23.11.2. *Центроидной декомпозицией будем называть два массива –* p_v *и* d_v.
Первый из массивов задаёт так называемое "дерево центроидной декомпозиции".

Замечание 23.11.3. Зная только глубины d_v, можно восстановить компоненты $C(v)$. Для этого нужно от центра v запустить dfs, который проходит только по вершинам большей глубины. Поэтому $C(v)$ хранить **не нужно**.

Lm 23.11.4. Для центроидной декомпозиции нужно $\Theta(n)$ памяти.

Lm 23.11.5. Глубина дерева центроидной декомпозиции не более $\log_2 n$.

Lm 23.11.6. Суммарный размер компонент $\mathcal{O}(n \log n)$.

Следствие 23.11.7. В задаче поиска min на пути суммарный размер предподсчёта $\mathcal{O}(n \log n)$.

Если декомпозиция уже построена, можем посчитать все минимумы от u до вершин $C(u)$ с помощью `dfs`:

```
1 // c - центроид
2 void calc_min(int c, int v, int parent = -1, int curMin = INT_MAX) {
3     curMin = min(curMin, w[v]); // пусть веса в вершинах
4     f[d[c]][v] = curMin; // на уровне d[c] каждая вершина v встретится ⩽ 1 раза
5     for (int x : graph[v])
6         if (x != parent && d[x] >= d[c])
7             calc_min(c, x, v, curMin);
8 }
9 calc_min(u, u);
```

Lm 23.11.8. Для любого пути $[a..b]$ на дереве, есть такой центр v: $v \in path[a..b]$ и $a, b \in C(v)$.

Чтобы найти такой центр v рассмотрим пути от a и b вверх в дереве центроидной декомпозиции и возьмём наименьшего общего предка (LCA) a и b – самый нижний такой центр, что $a, b \in C_v$.

$$\text{минимум на пути}[a..b] = \min(f[d_v, a], f[d_v, b])$$

Сейчас мы умеем LCA искать только за $\mathcal{O}(\text{глубины дерева})$. В нашем случае это $\mathcal{O}(\log n)$. На практике научимся за $\mathcal{O}(\log \log n)$, а в скором будущем вообще за $\mathcal{O}(1)$.

23.12. Реализация

```
// Эта функция нужна только, чтобы посчитать размер компоненты связности
int calc_size(int v, int parent = -1) {
    int size = 1;
    for (int x : graph[v])
        if (x != parent && d[x] == -1)
            size += calc_size(x, v);
    return size;
}
// Зная размер дерева n, считаем размеры поддеревьев, параллельно ищем центроид
int get_centroid(int v, int parent, int n, int &centroid) {
    int size = 1;
    for (int x : graph[v])
        if (x != parent && d[x] == -1)
            size += get_centroid(x, v, n, centroid);
    if (size * 2 >= n && centroid == -1)
        centroid = v;
    return size;
}
// Главная функция построения, v - любая вершина компоненты
void build(int v, int parent, int depth) {
    int centroid = -1;
    get_centroid(v, -1, calc_size(v), centroid);
    d[centroid] = depth, p[centroid] = parent;
    precalc_min(depth, centroid); // собственно насчитали минимумы, см. ниже
    for (int x : graph[centroid])
        if (d[x] == -1)
            build(x, centroid, depth + 1);
}
d = vector<int>(n, -1);
p = vector<int>(n, -1);
build(0, -1, 0);
```

На самом деле, чтобы решать задачи, нам не нужная прям центроидная декомпозиция, нам нужно выбирать вершину `centroid` так, чтобы глубина дерева декомпозиции была $\leqslant \log_2 n$. Благодаря этому, можно обойтись без `calc_size`:

```
void build(int v, int parent, int depth, int n) {
    int centroid = -1;
    get_centroid(v, -1, n, centroid); // n - оценка сверху на размер компоненты
    // если оценка наглая, if мог не сработать, но тогда v подходит
    if (centroid == -1) centroid = v;
    ...
            build(x, centroid, depth + 1, (n + 1) / 2);
}
```

И, наконец, часть, которая ищет минимум (немного иначе написанная, чем было выше). Нужно для каждого центроида насчитать dfs-ом минимумы на путях до корня.

```
f = vector<vector<int>>(LOG, vector<int>(n));
// depth - глубина центроида
void precalc_min(int depth, int v, int parent = -1, int curMin = INT_MAX) {
    curMin = min(curMin, weight[v]); // пусть веса будут в вершинах
    f[depth][v] = curMin; // на каждой глубине v встретится не более раза
    for (int x : graph[v])
        if (x != parent && d[x] == -1) // не ходим в уже помеченные вершины
            calc_min(depth, x, v, curMin);
}
int get_min(int a, int b)  {
    int lca = get_lca(a, b); // в дереве центроидной декомпозиции, заданной p[]
    return min(f[d[lca]][a], f[d[lca]][b]);
}
```

Лекция #24: Модели вычислений

24-я пара, 2024/25

24.1. RAM

Какие операции разрешены? `move/add/sub/mul/div/jz/jump` и, конечно, `halt/in/out`
Остальное выражается через них, например цикл `for` n раз:

```
i = n
:loop
    ...
    sub i 1
    jz end // Jump if Zero: если результат последей операции 0, выйти из цикла
    jump loop // перейти в начало цикла
:end
```

Важно, что память неограничена, к любой ячейке с целым номером i обращение за $\mathcal{O}(1)$. Обозначать можно, как $[i]$. Например $x = [i]$. В ячейках хранятся неограниченные целые числа $\Rightarrow$ возможна оптимизация: можно взять два массива целых чисел, закодировать из в два длинных целых числа, и сделать за $\mathcal{O}(1)$ операцию `+/-/OR/AND/XOR/</>` с массивами за $\mathcal{O}(1)$.

• **Флойд в RAM за $\mathcal{O}(V^2)$**

```
for k=1..n // V^3/64
    for i=1..n
        if d[i][k]
            for j=1..n
                d[i][j] |= d[k][j] // bitset-ы
```

Можно заменить на

```
for k=1..n // V^2
    for i=1..n
        if ((d[i] >> k) & 1) == 1 // операция с числами, O(1)
            d[i] |= d[k] // операция с n-битными числами, O(1)
```

• **bfs в RAM за $\mathcal{O}(V)$**

```
while queue.size(): // n шагов
    v = queue.pop() // O(1)
    while g[v] AND ((2^n-1) - used) != 0: // O(1)
        x = lowerbit(g[v] AND ((2^n-1) - used)) // O(1)
        used |= 2^x // O(1)
        queue.push(x) // O(1)
```

Аналогично `dfs` можно реализовать за $\mathcal{O}(V)$. См. практику.

24.2. P = NP

В модели вычислений RAM оказывается P = NP. Если коротко, можно распараллелить проверку всех $2^{P(n)}$ подсказок y, используя массивы длины $w \cdot 2^{P(n)}$, где w – битовый размер слова.

Теорема 24.2.1. SAT $\in$ P (в модели RAM)

Сначала напишем код для проверки одной подсказки y. Подсказка $= n$ бит, значения x_i.

```
result = 1
for i=1..m: // проверяем каждый клоз (скобку)
    f0 = ((y >> a[i][0]) & 1) ^ e[i][0] ^ 1 // мы хотим x_{a[i,0]} = e[i,0]
    f1 = ((y >> a[i][1]) & 1) ^ e[i][1] ^ 1 // или то же, для [1]
    f2 = ((y >> a[i][2]) & 1) ^ e[i][2] ^ 1 // или то же, для [2]
    result &= f0 | f1 | f2 // специально пишем операции для int-ов, пригодится
return result
```

Далее пусть $w = 3$. Теперь создадим число $Y = 000, 001, 002, 003\ldots$ (все 2^n y-ков в одном $w2^n$-битном числе). И число $E = 2^n$ групп 001 (единица). И то, и то можно сделать за $\mathcal{O}(\log(2^n))$ последовательным удвоением. Пусть $n = 2^k$. Научимся переходить $k \to k{+}1$ (удваивать).

$$Y \to ((Y + E2^k) << w2^k) + Y$$

$$E \to (E << w2^k) + E$$

Теперь можем смело писать код:

```
result = E
for i=1..m: // проверяем каждый клоз (скобку)
    F0 = ((Y >> a[i][0]) & E) ^ (e[i][0] ? E : ZERO) ^ E
    F1 = ((Y >> a[i][1]) & E) ^ (e[i][1] ? E : ZERO) ^ E
    F2 = ((Y >> a[i][2]) & E) ^ (e[i][2] ? E : ZERO) ^ E
    RESULT &= F0 | F1 | F2 // int-ы!
return RESULT != 0 // хотя бы в одном из y-ков результат проверки 1.
```

E и Y строим за $\mathcal{O}(n)$, код работает за $\mathcal{O}(m)$. Итого $\mathcal{O}(n + m)$.

24.3. Более сложные операции с массивами

• Сумма в массиве из нулей и единиц за $\mathcal{O}(\log n)$

Пусть массив = целое число $A \Rightarrow$ можем проделать тот же трюк, что и при подсчёте числа единичных бит: $A \to y = (A$ `AND` $01010101) + ((A$ `>>` $1)$ `AND` $01010101) \to z = (y$ `AND` $00110011) + ((y$ `>>` $2)$ `AND` $00110011) \to \ldots$ После i операций в группе из 2^i бит у нас записана их сумма. Индукционный шаг: сложить параллельно две соседние группы. Для подсчёта единичных бит трюк даёт $\mathcal{O}(\log w)$ битовых операций, для сложения чисел в массиве длины n, закодированным целым числом, $\mathcal{O}(\log n)$ арифметических операций в RAM.

• Число ненулевых элементов в массиве целых чисел

```
R = 0
for i=0..w-1:
    R |= (A >> i) & E // всё то же E, что и выше.
return sum(R) // как и в массиве из нулей и единиц, только группы нулей/единиц длины w.
```

24.4. RAM-w

Мы же с вами программируем в word-RAM (RAM-w). Это RAM с ограничением «за $\mathcal{O}(1)$ арифметические операции происходят лишь с числами из $[0, 2^w)$, и ячейки памяти также w-битовые». w-битовое число = word = машинное слово. Мы привыкли к 64-битным компьютерами, которые, как раз RAM-w для $w = 64$. На самом деле мы всегда предполагаем, что операции с числами $1..n$ происходят за $\mathcal{O}(1) \Rightarrow n < 2^w \Rightarrow \log n < w \Rightarrow w$ – не константа. Можно писать

$\mathcal{O}(Time(n)/\log n)$, можно более точно $\mathcal{O}(Time(n)/w)$.

• **Флойд/bfs/dfs в RAM-w**

Аналогичный код из раздела про RAM будет работать за $\mathcal{O}(V^3/w)$, так как n-битное число в RAM-w = bitset длины n, в котором биты разбиты на группы по w, все операции за $\mathcal{O}(n/w)$. Аналогично bfs/dfs = $\mathcal{O}(n)$ операций с битовыми векторами длины $n = \mathcal{O}(n^2/w)$.

24.5. Немного Ассемблера

C++ компилируют в asm
jz, jc, jo (zero, carry, overflow)
вычитание даёт == (jz) и даёт < (jo)
div даёт <

24.6. Машина Тьюринга

[wiki] [итмо-конспекты]

Пример наиболее простого языка программирования, в котором нет ничего лишнего. Про другие языки говорят «Тьюринг-полные», т.е. умеют всё то же, что и машина Тьюринга. В некотором смысле машина Тьюринга – RAM-w с указателем на текущую ячейку памяти, которую можно двигать только на `+-1` и без арифметики ($a + b$ нужно программировать самим).

Def 24.6.1. *Есть* **бесконечная** *лента A, в каждой ячейке символ из алфавита Σ. Есть текущий указатель на ячейку ленты p (целое число). Есть набор возможных состояний S, начальное состояние программы $s \in S$, конечное $t \in S$ (останавливается). И есть правила переходов M: смотрим на то в каком состоянии $s \in S$ мы находимся, смотрим на $A[p] \in \Sigma$ и делаем вывод: какой символ с записать в $A[p]$, в какое состояние s' перейти, куда сместиться (`L`/`R`). То есть, M: $\langle s, A[p]\rangle \to \langle s', c, p'\rangle$ причём $|p-p'| = 1$.*

Входные и выходные данные можно хранить на той же ленте, можно на отдельных.

• **Пример программы: reverse массива** .

Ставим пометку L, идём до конца, ставим пометку R, запоминаем a[R-1], идём до L, ставим в a[L+1] и запоминаем, что там было, идём до R...
TODO

• **Пример программы: +1 к длинному числу** .

Разобрано в практике.

Лекция #25: BST и AVL

25-я пара, 2024/25

25.1. 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 25.1.1. *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>

`std::set<int>` – какое-то BST (на самом деле RB-tree).
`std::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 «больше 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.

25.2. Немного кода

```
Node* next(Node* v) {
    if (v->r != 0) { // от правого сына спускаемся до упора влево
        v = v->r;
        while (v->l)
            v = v->l;
        return v;
    } else { // поднимаемся вверх, пока не окажемся левым сыном
        while (v->p && v->p->r == v)
            v = v->p;
        return v->p; // возвращает null, если next не существует
    }
}
```

• Добавление

```
Node* add(Node* v, int x) { // v -- корень поддерева
    if (!v)
        return new Node(x); // единственная новая вершина
    if (x < v->x)
        v->l = add(v->l, x);
    else
        v->r = add(v->r, x);
    return v;
}
```

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 25.2.1. *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.

• **Для любителей экзотики**

Реклама `Node**`. Полюбуйтесь на нерекурсивные `find`, `del`, `add`:

```
Node** find(Node** root, int x) {
    while (*root && (*root)->x != x)
        root = (x < (*root)->x ? &(*root)->l : &(*root)->r);
    return root;
}
bool isIn(Node *root, int x) {
    return *find(&root, x) != 0;
}
void add(Node** root, int x) { // предполагает, что x точно нет в дереве
    *find(root, x) = new Node(x);
}
void del(Node **root, int x) { // предполагает, что x точно есть в дереве
    Node **v = find(root, x);
    if (!(*v)->r) {
        *v = (*v)->l; // подцепили левого сына v к её отцу
        return;
    }
    Node **p = &(*v)->r;
    while ((*p)->l)
        p = &(*p)->l;
    swap((*v)->x, (*p)->x);
    *p = (*p)->r; // подцепили правого сына p к её отцу
}
```

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

Def 25.3.1. *Дерево будем называть сбалансированным iff глубина дерева* $\mathcal{O}(\log n)$.

Замечание 25.3.2. 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 25.3.3. *Высота вершины* v *– максимальное из расстояний от* v *до листьев в поддереве* v.

Def 25.3.4. *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 25.3.5. 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 $\geqslant 1.618^h \Rightarrow h \leqslant \log_{1.618} n$. ■

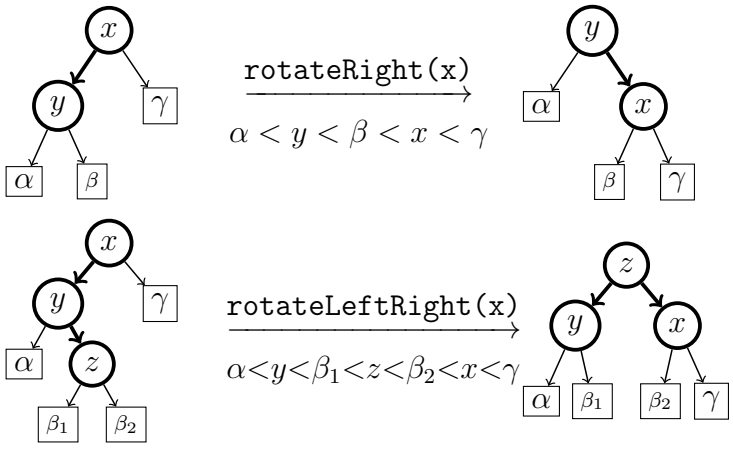
25.3.1. Добавление в AVL-дерево

• 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 $|v.l.h - v.r.h| = 2$, то AVL-свойство нарушено. Что делать? На обратном ходу рекурсии, поднимаясь снизу вверх, если видим некошерную вершину, «повернём» дерево (см. картинки).

(a) $h(v.l.l) \geqslant h(v.l.r)$.
В этом случае достаточно сделать малое вращение (single rotation) по ребру $(v \to v.l)$.

(b) $h(v.l.l) = h(v.l.r) - 1$.
В этом случае нужно делать большое вращение (double rotation) по рёбрам $(v \to v.l \to v.l.r)$.



Греческими буквами обозначены поддеревья, мы ничего не предполагаем про их непустоту. Расставляя высоты поддеревьев, убеждаемся, что AVL-свойство теперь выполнено.

• **Немного кода**

```
Node* rebalance(Node* v) {
    if (!v)
        return v;

    int hl = height(v->l), hr = height(v->r), d = hl - hr;
    assert(abs(d) <= 2);
    if (d == 2)
        if (height(v->l->l) >= height(v->l->r)) // v->l существует, так как hl ⩾ 2
            v = rotateRight(v); // Малое правое вращение
        else
            v = rotateLeftRight(v); // Большое правое вращение
    else if (d == -2)
        ...

    return v;
}

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 @!rebalance(v)!@;
}
```

Остаётся вопрос поддержки актуальных высот поддеревьев. Его мы затронем в Section 25.6.

• **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 убедиться, что если в вершине v происходит перебалансировка, то высота поддерева вершины v и до добавления, и после *добавления и вращения* равна $h(\gamma) + 2 \Rightarrow$
сразу же после первого вращения операцию перебалансировки можно прервать.
Однако это верно только для операции `add` (например, после операции `del` может понадобиться много перебалансировок на пути).

Lm 25.3.6. При добавлении в AVL-дереве происходит $\mathcal{O}(1)$ присваиваний указателей.

К сожалению, высоты могут поменяться у $\mathcal{O}(\log n)$ вершин.
На практике мы покажем, что тем не менее амортизированное число изменений высот $\mathcal{O}(1)$.

• Удаление

Удаление из AVL-дерева происходит также, как удаление из обычного BST.
На обратном ходу рекурсии от удалённой вершины происходит перебалансировка.
Подробнее про удаление из AVL на практике.

• Компактный код (персистентных) поворотов

Замечание 25.3.7. 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 25.3.8. Раз отцы не нужны, дерево можно сделать персистентным (см. Section 25.5).
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) { // персистентная версия
3      return new Node {v->l->x, v->l->l, new Node {v->x, v->l->r, v->r}};
4  }
5
6  Node* rotateLeftRight(Node* v) { // не персистентная версия
7      v->l = rotateLeft(v->l); // для персистентности здесь нужно создать новую вершину
8      return rotateRight(v);
9  }
10
11 Node* rotateLeftRight(Node* v) { // персистентная версия
12     return rotateRight(new Node {v->x, rotateLeft(v->l), v->r});
13 }
```

25.4. Split/Merge

Def 25.4.1. *split*(t, x) *делит* t *на* $l = \{a \mid a < x\}$ *и* $r = \{a \mid a \geqslant x\}$.

Def 25.4.2. *Пусть нам даны деревья* l, r: *все ключи в* l *меньше всех ключей в* r,
merge(l, r) – *BST, полученное объединением ключей* l *и* r.

Замечание 25.4.3. *merge* – операция, обратная *split*.

В разборе практики показано, как можно в AVL дереве сделать split и merge за $\mathcal{O}(\log n)$.

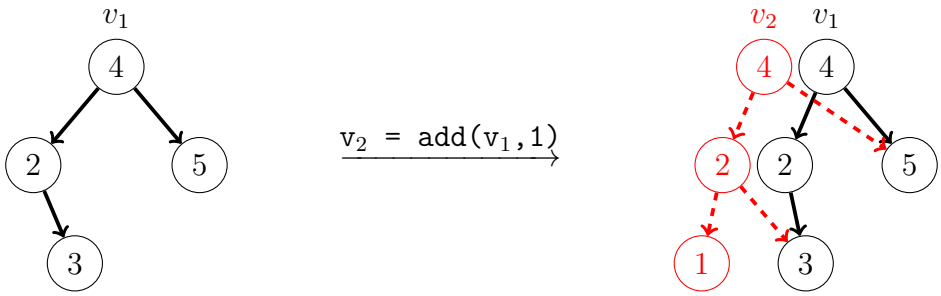
Замечание 25.4.4. По умолчанию split и merge ломают деревья, получаемые на вход.
Но их, как и все другие процедуры работы с BST, можно сделать персистентными.

25.5. Персистентность

Def 25.5.1. *Персистентной называется структура данных, которая при модификации сохраняет доступ ко всем предыдущим версиям.*

У нас уже есть персистентный `add` в обычное BST Algo 25.2.1. Отличие от не персистентной версии в том, что на обратном ходу рекурсии вместо того, чтобы менять старые вершины, мы создаём новые. По аналогии можно реализовать `del`.

Замечание 25.5.2. Каждая отдельная версия персистентного дерева – дерево.
Все версии в совокупности образуют ацикличный граф.



Замечание 25.5.3. У вершины теперь несколько "отцов", поддерживать их не получается.

Замечание 25.5.4. Есть несколько способов сделать структура персистентной. Рассмотренный нами называется path copying. Также полезна fat nodes.

25.6. Дополнительные операции, групповые операции

Можно поддерживать в вершине v размер поддерева `v->size`.
Всегда, когда поддерево вершины v меняется, считаем `v->size = v->l->size+v->r->size+1`
Чтобы не обрабатывать отдельно случай "`v->l` $= \varnothing$", будем как пустое поддерево использовать специальную фиктивную вершину `Node *nullNode = {size:0, l:nullNode, r:nullNode}`.

BST используется для хранения упорядоченного множества. Структура дерева задаёт на элементах порядок, поэтому у каждого элемента дерева есть позиция (номер).
Теперь, когда у каждой вершины хранится размер поддерева, мы умеем делать 4 операции:

`getValue(i)` – получать значение ключа по номеру
`getPosition(x)` – по ключу находить его номер в дереве
`add(i, data)` – вставить вершину с записью *data* на i-ю позицию в дерево
`del(i)` – удалить вершину, находящуюся на i-й позиции.

В процедурах `getValue(i)`, `add(i,data)`, `del(i)` используется спуск по дереву,
в котором сравнение идёт не по ключу, а по размеру поддерева:

```
1 Node* getValue( Node* v, int i ) { // `i - нумерация с нуля`
2     if (v->l->size == i)
3         return v;
4     if (v->l->size > i)
5         return getValue(v->l, i);
6     return getValue(v->r, i - v->l->size - 1);
7 }
```

• Операции на отрезке и групповые операции

Пусть в каждой вершине дерева кроме ключа x_v хранится некое число y_v.
Научимся для примера обрабатывать запрос "`getMin(l,r)` $= \min\{y_v \mid l \leqslant x_v \leqslant r\}$"
Также, как мы поддерживали размер поддерева, поддерживаем минимальный y_v в поддереве.

Каждой вершине BST соответствует поддерево, которому соответствует отрезок значений.
Сделаем `getMin(l,r)` за $\mathcal{O}(\log n)$ спуском по дереву.
По ходу спуска, находясь в вершине v, будем поддерживать отрезок значений $[vl, vr]$.

```
int getMin( Node* v, int vl, int vr, int l, int r ) {
    if (v == nullNode || r < vl || vr < l) // `[vl,vr] ∩ [l,r] = ∅`
        return INT_MAX;
    if (l <= vl && vr <= r) // `[vl,vr] ⊂ [l,r]`
        return v->min_y;
    return min(
        getMin(v->l, vl, v->x - 1, l, r),
        getMin(v->r, v->x + 1, vr, l, r),
        (l <= v->x && v->x <= r) ? v->y : INT_MAX
    );
}
int result = getMin(root, INT_MIN, INT_MAX, l, r);
```

Таким образом можно считать любую ассоциативную функцию от y_v в $\{y_v \mid l \leqslant x_v \leqslant r\}$.

• Модификация на отрезке

Хотим со всеми $\{y_v \mid l \leqslant x_v \leqslant r\}$ сделать y_v `+=` Δ (групповая операция).
Если мы захотим честно обновить все нужные y_v, это займёт линейное время.
Чтобы получить время $\mathcal{O}(\log n)$ на запрос, воспользуемся отложенными операциями:
у каждой вершины v будем хранить `v->add` – число, которое нужно прибавить ко всем вершинам в поддереве v. Запрос "прибавления на отрезке" (`rangeInc`) теперь обрабатывается по аналогии с `getMin`, есть три случая: (1) `[vl,vr]` $\cap$ `[l,r]` $= \varnothing$, (2) `vl,vr` $\subset$ `l,r` и (3) "отрезки пересекаются".

Операция называется отложенной, потому что в любом запросе (`add`, `del`, `getMin`, `rangeInc`), проходя через вершину v, у которой `v->add` $\neq 0$, мы проталкиваем эту операцию вниз детям:

```
void pushTo( Node* v, int x ) {
    if (v != nullNode) // `фиктивная вершина всегда должна оставаться в исходном состоянии`
        v->add += x, v->min += x;
}
void push( Node* v ) { // `push = проталкивание вниз`
    pushTo(v->l, v->add);
    pushTo(v->r, v->add);
    v->add = 0; // `при этом минимум не изменился`
}
```

25.7. Неявный ключ

Теперь наша цель – на основе BST сделать структуру данных для хранения массива, которая умеет делать вставку в середину `insert(i,x)`, удаление из середины `del(i)`, считать функцию на отрезке массива и, конечно, как и положено массиву, обращаться к ячейке `a[i]=z`, `z=a[i]`.

Возьмём дерево по ключу "индекс в массиве": $x_v = i,\ y_v = a_i$.
Тогда, если в каждой вершине хранить размер поддерева и минимум y_v в поддереве, у нас уже

есть операции `getValue(i)`, `add(i, data)`, `del(i)`, `getMin(l,r)`.
Нужно только после `add(i,data)` и `del(i)` пересчитывать ключи (индексы в массиве)...

Но зачем их пересчитывать и вообще хранить, если мы ими *нигде* не пользуемся?
Идея дерева по неявному ключу: можно просто не хранить ключ.

По неявному ключу также можно делать операции `split(v,i)` и `merge(l,r)`,
из них легко выразить циклический сдвиг массива = один `split` + один `merge`.

25.8. Reverse на отрезке

Если у нашего дерева есть split и merge, можно реализовать функции `getMin(l,r)`, `rangeInc(l,r)` и другие чуть проще: высплитим отрезок $[l, r)$; в его корне будет содержаться минимум на $[l, r)$, к этому корню можно за $\mathcal{O}(1)$ применить отложенную операцию; сделав с корнем $[l, r)$ всё, что хотели, смерджим деревья обратно. Пример:

```
1 Node* rangeInc( Node *v, int l, int r, int value ) {
2     Node *a, *b, *c;
3     split(v, b, c, r);
4     split(b, a, b, l);
5     b.add += value; // a = [0, l), b = [l, r), c = [r,end)
6     return merge(merge(a, b), c);
7 }
```

Техника "высплитим отрезок $[l, r)$" позволяет простой отложенной операцией `v->isReversed` реализовать `reverse(l, r)` (перевернуть отрезок массива).

Замечание 25.8.1. `getMin`, `rangeInc` можно было делать спуском по дереву, сделать `reverse` без выспличивания отрезка не получится.

25.9. Итоги

Идея отложенных операций и идея неявного ключа применимы к любым BST.

Операции `getValue(i)`, `getPosition(x)`, `add(i, data)`, `del(i)`, `getMin(l,r)`, `rangeInc(l,r)`, применимы к любому BST по явному ключу и неявному ключу.

Операция `reverse(l,r)` осмысленна только в дереве по неявному ключу, так как принципиально меняется порядок вершин дерева. В любом BST, где уже есть операции `split` и `merge`, можно получить `reverse(l,r)`.

Не в стандарте есть `tree` и `rope`. **TODO** ссылки

25.10. Персистентность: итоги

Любое дерево поиска, где не обязательно нужны отцы, можно сделать персистентным. В частности, AVL. Время работы останется пропорциональным глубине дерева, для AVL это $\mathcal{O}(\log n)$.

Следствие 25.10.1. $\exists$ персистентное дерево по неявному ключу с операциями за $\mathcal{O}(\log n)$.

Итого у нас есть персистентный массив с обращением к i-й ячейке и кучей ништяков (вставка в середину, `getMin`, `rangeInc`, `reverse` на отрезке и т.д.), который всё это умеет за $\mathcal{O}(\log n)$.

Следствие 25.10.2. Любую структуру данных можно сделать персистентной, с замедлением в $\mathcal{O}(\log n)$ раз. Просто поменяем все массивы на персистентные массивы.

Основной минус персистентных массивов – $\forall$ операция создаёт $\mathcal{O}(\log n)$ новых вершин дерева.

Лекция #26: Декартово дерево

26-я пара, 2024/25

26.1. Treap (Seidel,Aragon'1989)

Def 26.1.1. *Декартово дерево (cartesian tree) от множества пар $\langle x_i, y_i\rangle$ – структура, являющаяся BST по x_i и кучей с минимумом в корне по y_i.*

Lm 26.1.2. Если все y_i различны, декартово дерево единственно.

Доказательство. Корень выбирается однозначно – минимальный y_i. Левое поддерево – декартово дерево от всех $x_j < x_i$, оно единственно по индукции. Аналогично правое поддерево. ■

Def 26.1.3. *Случайное дерево поиска (RBST) от множества $\{x_i\}$.*
Каждый элемент с равной вероятностью $\frac{1}{n}$ может стать корнем.
Аналогичные условия выполнены рекурсивно сверху вниз во всех поддеревьях.

Def 26.1.4. *Случайное дерево поиска (RBST) от множества $\{x_i\}$ – результат добавления случайной перестановки x_i в пустое BST (спустились от корня вниз до упора, вставили лист).*

Lm 26.1.5. Если x_i различны, определения эквивалентны. (Напомним, что ключи можно всегда сделать различными, например, записав пары $\langle x_i, i\rangle$).

Доказательство. Корень – первый элемент перестановки, выбирается равновероятно.
Смотрим на левое поддерево. Элементы, попавшие в левое поддерево, также образуют случайную перестановку $\Rightarrow$ далее по индукции. ■

Мы хотим понять, что RBST является в каком-то смысле сбалансированным. В идеале мы бы хотели доказать следующее утверждение:

Lm 26.1.6. Матожидание высоты RBST (то есть *максимальной* глубины вершин) – $\mathcal{O}(\log n)$.

Это утверждение верно, но доказывать мы его не будем (есть в допах в теордз). Вместо этого докажем более простое утверждение, которого нам хватит для оценки времени работы:

Lm 26.1.7. Пусть RBST построено по набору ключей $\{x_1, x_2, \ldots, x_n\}$. Для каждого i обозначим за v_i узел, содержащий ключ x_i.
Тогда для каждого i матожидание глубины узла v_i — $\mathcal{O}(\log n)$.

Доказательство. Посмотрим на путь от корня до v_i. При каждом спуске с вероятностью хотя бы 1/3 размер поддерева уменьшался хотя бы в полтора раза: рассмотрим отсортированное множество ключей, с вероятностью 1/3 корень будет выбран из второй трети, тогда в поддерево, содержащее v_i, попадёт не больше 2/3 от всех ключей.
Тогда матожидание количества спусков, при которых размер поддерева уменьшится хотя бы в полтора раза, не больше трёх. Тогда матожидание количества спусков, при которых размер поддерева станет равным единице, не больше $3\log_{1.5} n = \mathcal{O}(\log n)$. (Возможно, размер поддерева с корнем в v_i даже больше единицы — нам только лучше.) ■

Замечание 26.1.8. Из этого, в частности, следует, что матожидание *средней* глубины вершин в RBST – $\mathcal{O}(\log n)$.

Def 26.1.9. *Декартово дерево (treap) от множества ключей* $\{x_i\}$ *– декартово дерево (cartesian tree) пар* $\langle x_i, random \rangle$.

Lm 26.1.10. Treap, как BST, является RBST

Следствие 26.1.11. Матожидание высоты Treap – $\mathcal{O}(\log n)$

Следствие 26.1.12. Матожидание глубины любой вершины в Treap – $\mathcal{O}(\log n)$

Следствие 26.1.13. Матожидание средней глубины вершины в Treap – $\mathcal{O}(\log n)$

Доказательство. В нашем курсе идёт без доказательства. Доказывать можно разными способами. Вариант через неравенство Йенсена. Более детский вариант. ■

26.2. Операции над Treap

В основе декартова дерева лежат операции Split и Merge, Add и Del выражается через них. И Split, и Merge – спуск вниз по дереву. Время обеих операций – глубина вершины, до которой мы спускаемся, т.е. $\mathcal{O}(\log n)$ (подробнее на практике).

Напомним, что `Split(x)` разделяет структуру данных на две того же типа – содержащую элементы "$< x$" и содержащую элементы "$\geqslant x$". `Merge(L, R)` – обратная к `Split` операция. В частности `Merge` требует, чтобы все ключи в L были меньше всех ключей в R.

Lm 26.2.1. Пусть над некоторым BST определены Split и Merge за $\mathcal{O}(\log n)$. Тогда можно доопределить: `Add(x) = Split + Merge + Merge`, `Del(x) = Split + Split + Merge`.

Доказательство. `Add`: разделим старое дерево на меньшие и большие x элементы. Итого, включая сам x, у нас три дерева. Склеим их. `Del`: разделим строе дерево на три – меньше x, сам x, больше x. Склеим крайние. ■

`Split` в декартовом дереве состоит из двух случаев – в какую из половин попал корень:

```
1 // чтобы вернуть пару, удобно получить в параметры две ссылки
2 void Split(Node* t, Node* &l, Node* &r, int x ) {
3     if (!t)
4         l = r = 0; // база индукции
5     else if (t->x < x)
6         Split(t->r, t->r, r, x), l = t;
7     else
8         Split(t->l, l, t->l, x), r = t;
9 }
```

Подробно объясним строку (5). Если корень `t->x` попал в левую половину, то элементы меньшие x – это корень, всё его левое поддерево и какая-то часть правого. Поэтому вызовемся рекурсивно от `t->r`, левую из образовавшихся частей подвесить к корню, правую запишем в `r`.

`Merge(L,R)` по определению должен сделать корнем `Node`-у с минимальным y. Это или корень `L`, или корень `R`. Пусть это `L.y`, тогда рекурсивно дёргаем `Merge(L.R, R)`.

26.3. Сборка мусора (garbage collection)

Если наше BST персистентно, а нам нужны не все версии, появляются бесполезные вершины, память от которых неплохо бы очистить. Простейший способ: `shared_ptr`, или рукописная «ссылочная сборка мусора» (работает быстрее стандартного `shared_ptr`):

```
struct Node {
    Node *l, *r;
    int cnt;
    void dec() { if (!--cnt) l->dec(), r->dec(), delete this; }
}
```

Более эффективный способ — не париться, пока память не закончится, как только закончилась, очистить разом всё лишнее. Тут можно поступить двумя способами:

• **Списковый аллокатор**

Поддерживаем список свободных `Node`-ов. В `Node` добавим бит `used`. В момент «список свободных меньше 100 (больше глубины, запас, которого хватит на 1 операция с деревом)»

1. Пометим сперва вообще все `Node` как `used=0`.
2. Запустим `dfs` от нужных корней.
3. Переберём все `Node`-ы и `used=0` добавим в список свободных.

Для организации списка удобно поле `Node* l` использовать как `next`.

• **Стековый аллокатор** ♡

Выделим стек на столько нод, сколько у нас есть памяти. Выделить `Node*`: вернуть кусок памяти `mem+mpos`, сдвинуть `mpos`. Когда память стека близка к концу (осталось <500 нод), сделаем rebuild: dfs-ом выпишем содержимое текущего дерево в массив a ($+4n$ байт памяти); сместим указатель `mpos` на 0 (освободим все ноды); заново построим дерево по массиву a.

Как технически реализовать на `C++`?
Или перегрузить `void* Node::operator new(size_t n);` (перегрузить new только для нужного нам типа). Или писать `Node* p = (mem + mpos)new Node(x, L, R);` – подсовывать оператору new уже выделенную память, где нужно разместить объект.

26.4. Дополнение о персистентности

Мы умеем представлять любую структуру данных в виде множества массивов и деревьев поиска. Дерево поиска само по себе может быть персистентным. Сделать массив персистентным можно двумя способами:

1. Уже умеем. Дерево поиска по неявному ключу. Оно кроме всего прочего будет Rope.
2. Научимся позже. Дерево отрезков с запросами сверху.

Когда нам от массива не нужно ничего кроме присваивания "`a[i] := x`" и чтения "`x = a[i]`", второй способ предподчтительней – его реализация проще.

26.4.1. Offline

Если все запросы к персистентной структуре известны заранее, мы можем построить дерево версий и обойти его поиском в глубину. При выходе из рекурсии нам нужно откатывать операции $\Rightarrow$ «амортизация» убивается (пример: persistent offline CHM будет работать за $\mathcal{O}(\log n)$).

26.4.2. Персистентная очередь за $\mathcal{O}(1)$

В прошлом семестре мы прошли "очередь с минимумом без амортизации". Там использовались две идеи: (а) очередь = два стека, (б) переворачивать стек можно лениво. Из последнего до-

машнего задания мы знаем, что персистентный стек с операциями push/pop/size/copy за $\mathcal{O}(1)$ существует и представляет собой дерево. Осталось собрать все знания в алгоритм.

Пусть наша очередь = `L|R` – стек `L` для `pop` и стек `R` для `push`. В момент `L.size == R.size` начнём ленивое копирование: `L|R` → `LR|`. Для этого создадим ко `L1 = L.copy(), R1 = R.copy()`, сделаем `R = empty`, и будем продолжать использовать `L/R` для `pop/push`, а новые нам нужно лишь для копирования. Вся процедура копирования выглядит так:

```
while (R1.size()) result.push(R1.pop()) // R1 попал в result в обратном порядке
while (L1.size()) tmp.push(L1.pop()) // переворачиваем
for (k = 0; k < L.size(); k++) // мы хотим не весь L1, а L, часть, не скушанную pop-ами
    result.push(tmp.pop()) // нужная часть L1 попал в result в таком же порядке
```

Если изначально n == `L.size` == `R.size`, нам нужно $3n$ шагов для копирования ⇒ за 1 `pop/push` достаточно делать 3 шага, чтобы точно успеть, пока `L` не опустел и пока `R` не вырос в 2 раза.

Далее можно прочесть полную версию кода на 5 стеках.

```
struct Queue {
    Stack L, R; // L для pop, R для push
    Stack L1, R1, tmp; // вспомогательные стеки, итого 5 стеков
    int state, copied;

    Queue Copy() const {
        return Queue(L.copy(),R.copy(),L1.copy(),R1.copy(),tmp.copy(),state,copied);
    }
    int Front() const {
        return L.front(); // очередь не пуста => L не пуст!
    }
    pair<Queue, int> Pop() const {
        Queue res = Copy();
        int data = res.L.pop();
        forn(i, 3) res.Step();
        return make_pair(res, data);
    }
    Queue Push(int data) const {
        Queue res = Copy();
        res.R.push(data);
        forn(i, 3) res.Step();
        return res;
    }
    void Step() { // основной шаг переворачивания; этот метод не const!
        if (state == DO_NOTHING) {
            // Если у нас достаточно большой pop-стек, рано волноваться
            if (L.size > R.size) return;
            // Иначе L.size == R.size, и мы начинаем копирование
            R1 = R, R = new Stack(), tmp = new Stack();
            state = REVERSE_R;
        }
        if (state == REVERSE_R) {
            tmp.push(R1.pop())
            if (R1.size == 0) L1 = L.copy(), state = REVERSE_L;
        } else if (state == REVERSE_L) {
            R1.push(L1.pop());
            if (L1.size == 0) copied = 0, state = REVERSE_L_AGAIN;
        } else { // state == REVERSE_L_AGAIN
```

```
            if (L.size > copied)  copied++, tmp.push(R1.pop());
            if (L.size == copied) L = tmp, state = DO_NOTHING;
        }
    }
};
```

У структуры `Stack` операции `pop` и `push` меняют стек.
Персистентность стека используется только в момент вызова метода `Copy`.

26.4.3. Простой персистентный дек (pairing)

PairingDeque — рекурсивная структура данных, которая шепчет нам «давайте хранить не более одного элемента слева, не более одного справа, а остальные разобьём на пары и рекурсивно сделаем с ними то же самое».

Время работы операций `push_back`, `pop_back`, `push_front`, `pop_front` – $\mathcal{O}(\log n)$.

Мы уже умели получать такое время (BST, дерево отрезков), но у данная структура проще и имеет меньшую константу. Из крутого: если мы делаем много `push_back` подряд, амортизированное время – $\mathcal{O}(1)$ на операцию. Ниже представлен код на псевдоплюсах.

```
struct Deque<T> {
    T l; // элемент или null
    Deque<pair<T,T>> m; // все остальные разбиты на парах
    T r; // элемент или null

    Deque<T> push_back(T x) { // возвращаем новую версию
        if (r == null) return {l, m, x};
        return Deque(l, m.push_back(pair(r, x)), null);
    }
    pair<T, Deque<T>> pop_back() { // возвращаем то, что достали, и новую версию
        if (r != null) return pair(r, Deque(l, m, null));
        if (m == null) return pair(l, Deque(null, null, null));
        <x, m1> = m.pop_back();
        return pair(x.second, Deque(l, m1, x.first));
    }
}
```

26.4.4. Treap и избавление от Y

В структуре Treap мы пользовались игреками для того, чтобы оценить полученную структуру как RBST. Попробуем избавиться от хранения y, но всё ещё сохранить случайность дерева.

Обсудим только операции Split и Merge, раз остальные выражаются через них. Функция Split не использовала y, поэтому не поменяется. Как поменять функцию Merge?

Если мы хотим смержить дерево A и дерево B, то кинем монетку: с вероятностью $\frac{|A|}{|A|+|B|}$ выберем в качестве корня корень A, с вероятностью $\frac{|B|}{|A|+|B|}$ выберем в качестве корня корень B.

Почему именно такие вероятности? Каждый из ключей в A является корнем A с вероятностью $1/|A|$, каждый из ключей B является корнем B с вероятностью $1/|B|$.
Получается, что каждый ключ объединения является корнем с вероятностью $\frac{1}{|A|+|B|}$.
По индукции, корни всех оставшихся поддеревьев тоже будут выбраны случайно. Конец.

Плюсы: сэкономили память, теперь всегда можно делать Treap персистентным.

Минусы: работает медленней, random – нетривиальное $\mathcal{O}(1)$.

Замечание 26.4.1. Какая вообще была проблема с персистентностью? Если мы делаем `root = new Node; for 20 раз root = Merge(root, root)`, мы раньше получали дерево из 2^{20} одинаковых y-ов, которые благополучно выстраивались в бамбук.

Если мы не планируем никогда `Merge`-ить дерево с собой или пересекающиеся части одного дерева, у нас такой проблемы не будет и можно спокойно использовать y-и.

26.5. (*) Частичная персистентность: fat nodes

Def 26.5.1. *Структура данных частично персистентна, если допускает модификацию последней версии и get-запрос к любой.*

Замечание 26.5.2. Частично персистентный массив тривиально сделать за $\mathcal{O}(1)$ памяти/времени на модификацию и $\mathcal{O}(\log n)$ времени на обращение: храним независимо для каждой ячейки вектор пар $\langle$время модификации, значения$\rangle$; обращение = бинпоиск.

Теорема 26.5.3. Если у нас есть ссылочная структура (BST, список, двусвязный список) с ограниченной входящей степенью (во всех перечисленных структурах $\leqslant 2$), можно получить амортизированную оценку $\mathcal{O}(1)$ времени и на модификацию, и на обращение.

Доказательство. Техника fat nodes.
Можно было бы воспользоваться тривиальным решением и в каждой вершине хранить вектор $\langle time_i, version_i\rangle$, при каждом обращении к каждой вершине пришлось бы делать бинпоиск.
Оптимизация: будем для каждой вершины хранить не более k версий. Когда у вершины приходится создавать $(k+1)$-ую версию, создадим копию вершины из одной новой версии, всем предкам вершины рекурсивно создадим новую версию, направив исходящее ребро в копию.

```
1 # Vertex :  parents, versions
2 def newVersion(t, v, newVersion):
3     if len(v.versions) < k:
4         v.versions.append((t, newVersion))
5         return
6     u = Vertex(v.parents, [(t, newVersion)])
7     for p in v.parents:
8         newVersion(t, p, redirectLink(p, v, u))
```

Осталось выбрать число k. Возьмём любое, чтобы $\forall v\ |v.parents| < k$. Время работы и память оцениваются одинаково. Потенциал $\varphi = \sum |v.versions|$, где сумма берётся только по вершинам, содержащим последнюю версию. Когда мы создаём новую вершину, мы уменьшаем потенциал на k и увеличиваем на $|v.parents| \Rightarrow$ уменьшаем. ■

• **Примеры:**

◦ Treap. Обычная операция split. Создаст $\mathcal{O}(\log n)$ новых вершин, отработает за $\mathcal{O}(\log n)$.
Её fat-node-персистентный аналог создаст асимптотически столько же вершин за то же время.

◦ AVL по явному ключу. Операция add модифицирует ссылки у $\mathcal{O}(1)$ вершин. Высоты – промежуточная информация, её можно смело портить у старых версий. Значит n персистентных операций add в AVL используют $\mathcal{O}(n)$ памяти и $\mathcal{O}(n \log n)$ времени.

◦ AVL по неявному ключу. Нужно хранить размеры $\Rightarrow$ поменять $\Theta(\log n)$ вершин.

◦ Дерево отрезков. Одна модификация в точке изменяет $\log n$ вершин $\Rightarrow$
n модификаций изменят $n \log n$ вершин $\Rightarrow$ создадут $\mathcal{O}(n \log n)$ новых.

Следствие 26.5.4. Если взять scanline с деревом отрезков или, мы не получили преимущества перед обычной персистентностью. Если взять scanline с BST, то важно, какое именно BST использовать, и какие операции с ним. Пусть используем AVL с явным ключом, операции только add $\Rightarrow$ profit.

Пример задачи: online проверка, лежит ли точка внутри невыпуклого многоугольника (обобщение: в какой грани планарного графа лежит точка). Научились её решать за $\langle \mathcal{O}(n), \mathcal{O}(\log n) \rangle$.

26.6. B-дерево (Bayer, McCreight'1972)

Зафиксируем число k. Вершиной дерева будем называть множество от $k-1$ до $2k-2$ ключей. Если в вершине хранится i ключей $x_1, x_2, \dots x_i$, то детей у неё $i+1$: $T_1, T_2, \dots, T_{i+1}$. При этом верно $T_1 < x_1 < T_2 < \dots < x_i < T_{i+1}$. Вспомним бинарное дерево поиска: $T_1 < x_1 < T_2$, один ключ, два ребёнка.

Def 26.6.1. *B-дерево: для всех вершин кроме, возможно, корня количество ключей* $[k-1, 2k-1)$*, соответственно количество детей* $[k, 2k)$*, все листья имеют одинаковую глубину.*

Lm 26.6.2. При $k \geqslant 2$ глубина B дерева, хранящего n ключей, – $\Theta(\log_k n)$

При больших k встаёт вопрос: как хранить множество ключей и детей? В основном используют два варианта: в виде отсортированного массива или в виде дерева поиска.

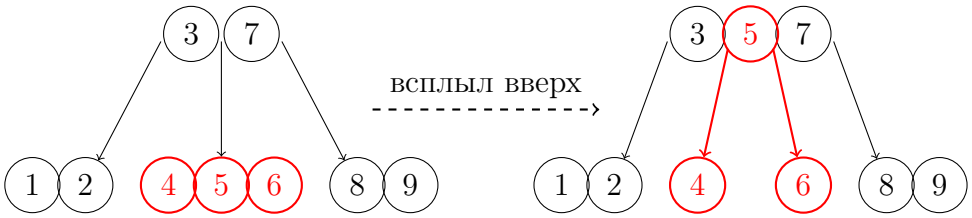
26.6.1. Поиск по B-дереву

Спуск вниз по дереву. Выбор направления, куда спускаться, происходит за $\mathcal{O}(\log k)$, – бинпоиск или поиск в BST. Итого $\log_k n \ \log k = \log n$ процессорных операций.

Преимущество B-дерева над другими BST – количество операций чтения с диска. Пусть мы храним в дереве большую базу данных на $10^9..10^{10}$ записей. Тогда в оперативную память такая база не помещается, и при чтении из базы происходит чтение с жесткого диска. При чтении с жесткого диска за одну операцию чтения мы читаем не 1 байт, а сразу блок, например из 4 килобайт. B-дерево, не ухудшая процессорное время, минимизирует число обращений к диску.

26.6.2. Добавление в B-дерево

Спустимся до листьев, добавим в листе v в пачку из $[k{-}1..2k{-}2]$ ключей ещё один ключ. Если листьев в пачке стало $2k{-}1$, скажем "вершина переполнилась" и разделим её на две вершины по $k{-}1$ ключу, соединённые общим ключом-отцом (итого $(k{-}1)+(k{-}1)+1 = 2k{-}1$ ключ). Этот ключ-отец вставим в соответствующее место отца v. Можно представлять эту операцию, как будто средний из ключей всплыл вверх на один уровень.



На уровень выше опять могла возникнуть проблема "слишком много ключей в вершине", тогда рекурсивно продолжим "толкать ключи вверх". Пример такой ситуации для $k = 2$ смотрите на картинке. При $k = 2$ в каждой вершине должно быть от 1 до 2 ключей.

При проталкивании ключа вверх важно, как мы храним $\Theta(k)$ ключей. Если в сортированном массиве, вставка происходит за $\mathcal{O}(k \log_k n)$. Если в дереве поиска, то за $\mathcal{O}(\log k \log_k n) = \mathcal{O}(\log n)$.

26.6.3. Удаление из B-дерева

Как обычно, сперва найдём вершину, затем, если она не лист, swap-нем её с соседним по значению ключом справа. Теперь задача свелась к удалению ключа из листа. Удалим. Сломаться могло то, что в вершине теперь меньше $k{-}1$ ключей. Тогда возьмём вершину-брата, и общего ключа-отца для этих двух вершин и "спустим ключ-отец вниз". Это операция обратная изображённой на картинке выше. В результате операции мы могли получить слишком толстую вершину, тогда обратно разобьём её на две (получится, что мы просто сколько-то ключей забрали от более толстого брата себе). У вершины-отца могла уменьшиться число ключей, тогда рекурсивно пойдём вверх от отца.

Итого. Мы сперва переходим к листу. Затем удаляем ключ из листа (из вершины по середине дерева нельзя просто взять и удалить ключ). И дисбаланс ключей мы исправляем подъёмом от листа к корню дереву.

Замечание 26.6.3. В удалении и добавлении за $\mathcal{O}(\log n)$ мы пользуемся тем, что дерево поиска умеет делать split и merge за логарифмическое время.

26.6.4. Модификации

B*-tree предлагает уплотнение до $[k, \frac{3}{2}k]$ ключей в вершине. Разобрано на практике.
B$^+$-tree предлагает хранить ключи только в листьях. Можно почитать в wiki.

26.6.5. Split/Merge

Делаются за высоту дерева. Будут в практике, как упражнения. У деревьев должно быть одинаковое k, но могут быть произвольные высоты.

26.7. Производные B-дерева

26.7.1. 2-3-дерево (Hopcroft'1970)

При $k = 2$ у каждой вершины от 2 до 3 детей. Такая конструкция называется 2-3-деревом.

26.7.2. 2-3-4-дерево и RB-дерево (Bayer'1972)

Можно ещё разрешить вершины, у которых ровно 4 сына. Такая конструкция будет называться 2-3-4-деревом. Поддерживать 2-3-4 свойство при добавлении и удалении также, как и у 2-3 дерева: слишком толстые вершины делим на две, тонкие соединяем с братьями.

2-3-4-дерево ничем не лучше 2-3-дерева. Зато оно эквивалентно красно чёрному дереву (RB-дерево). А RB-дерево это быстрая и популярная реализация сбалансированного дерева поиска. Например, используется в `C++.STL` внутри `set<>`.

Def 26.7.1. *Рассмотрим BST, в котором каждая вершина покрашена в красный или чёрный. Также в каждое место "отсутствия сына" мысленно добавим фиктивного сына. Дерево называется красно чёрным, если:*

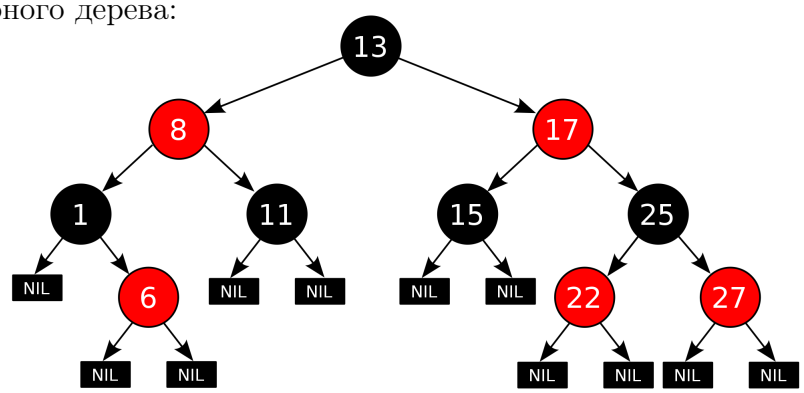
(a) На пути от корня до любой фиктивной вершины одинаковое число чёрных вершин.
(b) Сыном красной вершины не может быть красная вершина.
(c) Корень – чёрная вершина.

Lm 26.7.2. Есть преобразование между красночёрными и 2-3-4-деревьями.

Доказательство. Соединим красные вершины с их чёрными отцами, получим толстую вершину, в которой от 1 до 3 ключей и от 2 до 4 детей. В обратную сторону: у вершины 2-3-4-дерева с двумя ключами создадим правого красного сына, у вершины 2-3-4-дерева с тремя ключами создадим два красных сына. ■

Замечание 26.7.3. Это почти биекция. В случае "один красный сын" при сливании его с корнем теряется информация правый он или левый.

Пример красно-чёрного дерева:



26.7.3. AA-дерево (Arne Anderson'1993)

Из 2-3-4-дерева разделением толстой вершины на "чёрную с красными детьми" можно получить RB-дерево. Но мы знаем, что можно жить без 4-вершин: если 2-3-дерево по тому же принципу преобразовать в RB-дерево, получится так называемое AA-дерево.

Def 26.7.4. *RB-дерево называется AA-деревом, если любая красная вершина является правым сыном своего отца.*

RB-дерево и AA-дерево при аккуратной реализации имеют отличную скорость работы. Также заметим, что оба дерева кроме обычных любому BST ссылок на детей хранят лишь один бит – цвет вершины. Для сравнения их между собой приведём цитату из работы Андерсона:

The performance of an AA tree is equivalent to the performance of a red-black tree. While an AA tree makes more rotations than a red-black tree, the simpler algorithms tend to be faster, and all of this balances out to result in similar performance. A red-black tree is more consistent in its performance than an AA tree, but an AA tree tends to be flatter, which results in slightly faster search times.

AA-дерево известно тем, что у него есть простая/красивая/быстрая реализация.

Лекция #27: Splay и корневая оптимизация

27-я пара, 2024/25

27.1. Rope

Def 27.1.1. *Rope – интерфейс структуры данных, требующий, чтобы она умела производить с массивом следующие операции:*

(a) `insert(i)`, `erase(i)`
(b) `split(i)`, `merge(a, b)`, `rotate(k)`

Обычно подразумевают, что структура все выше перечисленные операции умеет делать быстро, например, за $\mathcal{O}(\log n)$. Всё выше перечисленное умеют сбалансированные деревья по неявному ключу со `split` и `merge`. У нас уже есть AVL-tree, RB-tree, Treap, сегодня ещё появится Splay-tree. Кроме того есть структуры, в основе которых лежат не деревья поиска, удовлетворяющие интерфейсу Rope. Из таких у нас сегодня появятся Skip-List и SQRT-decomposition.

27.2. Skip-list

Структура данных *односвязный список* хороша тем, что `split`, `merge`, `insert`, `erase` работают за $\mathcal{O}(1)$. За $\mathcal{O}(n)$ работает только операция поиска. Например `split(i) = find(i) + `$\mathcal{O}(1)$. Чтобы ускорить поиск можно добавить ссылки вперёд на 2^k шагов.

TODO: картинка с иллюстрацией структуры данных и нового `find`.

Правда после `insert` и `erase` такие ссылки неудобно пересчитывать.
Поэтому авторы Skip-List поступили хитрее. Skip-List – $\log_2 n$ списков. Нижний (нулевой) список – все данные нам элементы. Каждый элемент из i-го списка с вероятностью 0.5 содержится в $(i+1)$-м списке. Итого любой элемент в i-м списке содержится с вероятностью $2^{-i} \Rightarrow$
$E[|List_i|] = n2^{-i} \Rightarrow$ суммарный размер всех списков $2n$.

```
1  struct Node {
2      Node *down, *right;
3      int x, right_len;
4  };
5  const int LOG_N = 17;
6  vector<Node*> head[LOG_N+1]; // для удобства считаем, что log n не меняется
7  Node* find(int i):
8      Node *res;
9      int pos = -1; // в каждом списке голова - фиктивный (-1)-й элемент
10     for (Node *v = head[LOG_N]; v; res = v, v = v->down)
11         while (v->right && pos + v->right_len < i)
12             pos += v->right_len, v = v->right;
13     return res; // максимальный ключ меньший x в нижнем списке
```

Lm 27.2.1. Матожидание времени работы `find` – $\mathcal{O}(\log n)$.

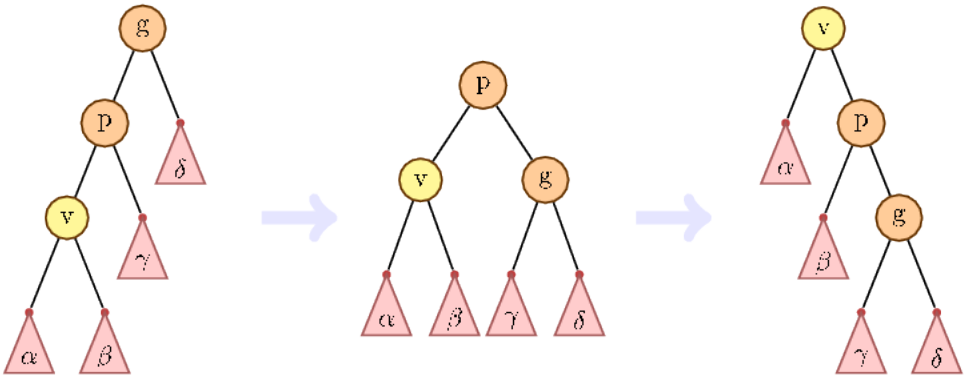
Доказательство. На каждом шаге `for` матожидание число шагов $\mathcal{O}(1)$. ■

Чтобы удалить i-элемент, сделаем `find(i)`, который в каждом списке пройдёт по элементу, предшествующему i. Для каждого списка: если i был в списке, удалим за $\mathcal{O}(1)$, сложим длины соседних рёбер. Если i не было, уменьшим длину ссылки вправо на 1.

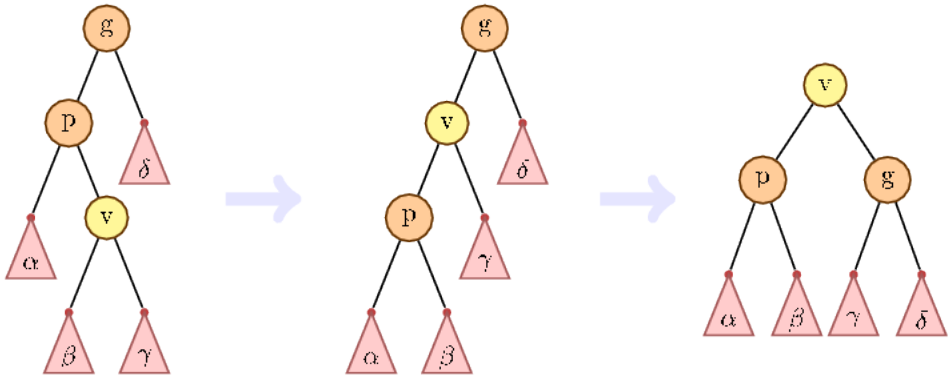
27.3. Splay tree

Splay-дерево — самобалансирующееся BST, не хранящее в вершине никакой дополнительной информации. В худшем случае глубина может быть линейна, но амортизированное время всех операций получится $\mathcal{O}(\log n)$. Возьмём обычное не сбалансированное дерево. При `add`/`del`. Модифицируем `add`: спустившись вниз до вершины v он самортизирует потраченное время вызовом `Splay(v)`, которая последовательными вращениями протолкнёт v до корня.

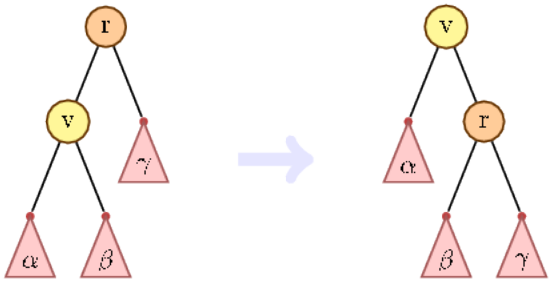
- **Zig-zig вращение**



- **Zig-zag вращение**



- **Если дедушки нет,** сделаем обычный single rotation (zig).



В частности из картинок видно, что все вращения выражаются через single rotation.
Любые другие операции со splay деревом делаются также, как и `add`: пусть v – самая глубокая вершина, до которой мы спустились $\Rightarrow$ вызовем `splay(v)`, который протолкнёт v в корень и самортизирует время работы. При этом всегда время `splay(v)` не меньше остальной полезной части $\Rightarrow$ осталось оценить время работы `splay`.

Lm 27.3.1. $x, y > 0, x + y = 1 \Rightarrow \log x + \log y \leqslant -2$

Доказательство. $\log x + \log y = \log x + \log(1 - x) = \log x(1 - x) \leqslant \log \frac{1}{2} \cdot 2 = -2$ ■

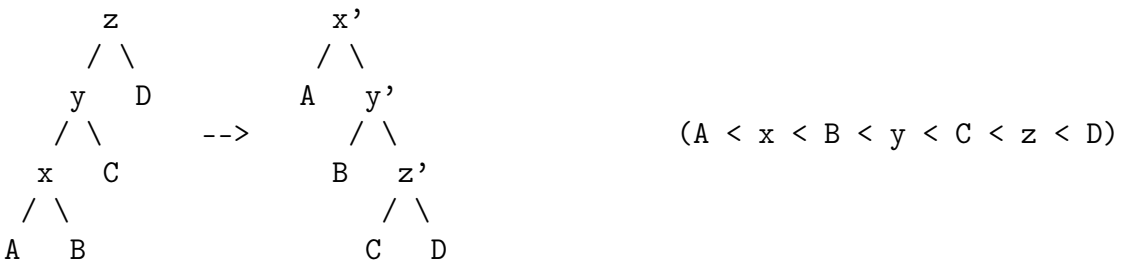
Lm 27.3.2. $x, y > 0, x + y = C \Rightarrow \log x + \log y \leqslant 2 \log C - 2$

Доказательство. $\log x + \log y = 2 \log C + \log \frac{x}{C} + \log \frac{y}{C} \leqslant 2 \log C - 2$ ■

Теперь введём потенциал. Ранг вершины $R_v = \log(size_v)$, где $size_v$ – размер поддерева. Потенциал $\varphi = \sum R_v$. Заметим сразу, что для пустого дерева $\varphi_0 = 0$ и $\forall$ момент времени $\varphi \geqslant 0$. Оценим амортизированное время операции `splay`, поднявшей v в u:

Теорема 27.3.3. $\forall v, u\ a_{v \to u} \leqslant 3(R_u - R_v) + 1 = 3 \log \frac{size_u}{size_v} + 1$

Доказательство. Полное доказательство доступно здесь. Мы разберём только случай zig-zig. Оставшиеся два аналогичны. +1 вылезет из случая zig (отсутствие деда). Итак, $a = t + \Delta\varphi = 2 + (R_{x'} + R_{y'} + R_{z'}) - (R_x + R_y + R_z) = 2 + R_{y'} + R_{z'} - R_x - R_y \leqslant 2 + R_{x'} + R_{z'} - 2R_x = F$.



Мы хотим показать $F \leqslant 3(R_{x'} - R_x) \Leftrightarrow R_{z'} \leqslant 2R_{x'} - R_x - 2 \Leftrightarrow R_{z'} + R_x \leqslant 2R_{x'} - 2$.
Теперь вспомним, что $R_{z'} = \log(C + D + 1), R_x = \log(A + B + 1) \overset{\text{лемма}}{\Rightarrow}$
$R_{z'} + R_x \leqslant 2 \log(A + B + C + D + 2) - 2 \leqslant 2R_{x'} - 2$. ■

Следствие 27.3.4. Среднее время одной операции в splay-дереве – $\mathcal{O}(\log n)$.

Доказательство. $\varphi_0 = 0, \varphi \geqslant 0 \Rightarrow \frac{1}{m} \sum t_i \leqslant a_i = \mathcal{O}(\log n)$. ■

Замечание 27.3.5. В теорему вместо $size_v$ можно подставить любой взвешенный размер: $size_v = w_v + size_l + size_r$, где w_v – вес вершины.
Если $w_v \geqslant 1$, то потенциал в каждый момент неотрицательный.

Полученная оценка на время работы операции Splay позволяет получать более крутые оценки на операции во многих случаях.

27.4. (*) Статическая оптимальность

Пусть дано множество пар $\langle x_i, p_i \rangle$. Здесь x_i – различные ключи, а p_i – вероятность того, что очередной запрос будет к ключу x_i. По ключам x_i можно построить различные BST. Новые ключи добавляться не будут. По ходу запросов менять структуру дерева нельзя. Время запроса к ключу x_i равно глубине соответствующего ключа, обозначим её d_i.

Def 27.4.1. *BST называется статически оптимальным, если минимизирует матожидание времени запроса:* $\sum_i p_i d_i \to min$.

Задачу построения статически оптимального BST решил Дональд Кнут в 1971-м году. Кстати, похожую задачу решает алгоритм Хаффмана, разница в том, что Хаффману разрешено менять порядок ключей, а здесь порядок фиксирован: $x_1 < x_2 < \cdots < x_n$.

Статически оптимальное BST можно построить динамикой по подотрезкам за $\mathcal{O}(n^3)$:

$$f[l,r] = min_{root}\big[(\sum_{i\in[l..r]} p_i) - p_{root} + f[l, root{-}1] + f[root{+}1, r]\big]$$

Сумма p-шек обозначает, что все ключи кроме ключа x_{root} имеют глубину хотя бы 1, и эту единицу мы можем учесть уже сейчас. После чего разобьёмся на две независимые задачи – построение левого и правого поддеревьев.

Если обозначить минимальный из оптимальных $root$ за $i[l,r]$, то будет верно,
что $i[l,r{-}1] \leqslant i[l,r] \leqslant i[l{+}1,r]$, что и доказал в своей работе Кнут.
Это приводит нас к оптимизации динамики до $\mathcal{O}(n^2)$:
будем перебирать $i[l,r]$ не от l до r, а между двумя уже посчитанными i-шками.

27.5. Splay Tree и оптимальность

Теорема 27.5.1. Пусть к splay-дереву поступают только запросы `find(v)`, k_v – количество обращений к вершине v, $m = \sum k_v$. Тогда суммарное время всех `find` равно $\mathcal{O}(m + \sum_v k_v \log \frac{m}{k_v})$.

Доказательство. Если покажем, что $(3\log\frac{m}{k_v}+1)$ – время на один запрос к вершине v, получимт нужную сумму. Эта оценка на время получается из Thm 27.3.3 подстановкой веса вершины $w_v = k_v$. Тогда $size_{root} = m$ и время $splay(v) \leqslant 3\log\frac{size_{root}}{size_v} + 1 \leqslant 3\log\frac{size_{root}}{k_v} + 1$. ■

Для задачи статически оптимального дерева поиска также известно жадное C-OPT решение. Из существования этого решения следует, что оценка на Splay дерево из предыдущей теоремы также отличается от статически оптимального BST всего лишь в константу раз.

Также есть гипотеза о *динамической оптимальности* Splay-дерева:

Пусть есть какой-то абстрактный алгоритм работы с BST. К нему поступают запросы поиска, которые работают за $d(key)+1$. Также между запросами к BST можно вращать, за 1 времени на каждый поворот. Не существует такого сценария запросов, в котором Splay дерево будет работать хуже, чем $\mathcal{O}(n + t_{opt})$.

В практике мы также доказали теорему о времени работы бора и Static Finger теорему.

В той же практике предлагается более быстрая `top-down` реализация splay-дерева.

27.6. SQRT decomposition

27.6.1. Корневая по массиву

Идея: разобьём массив на $\sqrt{n}$ частей (кусков) размера $k = \sqrt{n}$.

• Сумма на отрезке и изменение в точке

Решение: $\mathcal{O}(1)$ на изменение, $\mathcal{O}(\sqrt{n})$ на запрос суммы. $\forall i$ для i куска поддерживаем сумму `s[i]`.

```
void change(i, x):
    s[i/k] += (x-a[i]), a[i]=x
int get(l, r): // [l, r)
    int res = 0
    while (l < r && l % k != 0) res += a[l++]; // левый хвост
    while (l < r && r % k != 0) res += a[--r]; // правый хвост
    return accumulate(s + l / k, s + r / k, res); // цельные куски
```

Запрос на отрезке разбивается на два хвоста длины $\sqrt{n}$ и не более $\sqrt{n}$ цельных кусков.

Решение за $\mathcal{O}(\sqrt{n})$ на изменение и $\mathcal{O}(1)$ на запрос суммы: будем поддерживать частичные суммы для каждого куска и для массива s. При изменении пересчитаем за $\mathcal{O}(\sqrt{n})$ частичные суммы внутри куска номер `i/k` и частичные суммы массива s. Суммы на хвостах и на отрезке массива s считаются за $\mathcal{O}(1)$.

• **Минимум на отрезке и изменение в точке**

Решение за $\mathcal{O}(\sqrt{n})$ на оба запроса – поддерживать минимум в каждом куске. В отличии от суммы, минимум мы сможем пересчитать только за $\mathcal{O}(\sqrt{n})$.

27.6.2. Корневая через split/merge

Дополним предыдущие две задачи операциями `insert(i,x)` и `erase(i)`.
Будем хранить `vector` или `list` кусков. Здесь i-й кусок – это отрезок $[l_i, r_i)$ нашего массива, мы храним его как `vector`. Сам массив мы не храним, только его разбиение на куски.
Когда приходит операция `insert`/`erase`, ищем, про какой она кусок за $\mathcal{O}(\sqrt{n})$. Теперь сделаем эту операцию в найденном куске за $\mathcal{O}(\sqrt{n})$. При этом кусок мог уменьшиться/увеличиться. Кусок размера меньше $\sqrt{n}$ смерджим с соседним. Кусок размера $2\sqrt{n}$ посплитим на два.
Поскольку мы удерживаем размер куска в $[\sqrt{n}, 2\sqrt{n})$, количество кусков всегда $\Theta(\sqrt{n})$.
Время старых операций не изменилось, время новых – $\mathcal{O}(\sqrt{n})$.

27.6.3. Корневая через split/rebuild

Храним то же, что и в прошлой задаче.
Операция `split(i)` – сделать так, чтобы i-й элемент был началом куска (если это не так, кусок, в котором лежит i, нужно разделить на два). В задачах про сумму и минимум `split` $= \mathcal{O}(\sqrt{n})$.
Ещё удобнее, если `split` возвращает номер куска, началом которого является i-й элемент.
Любой запрос на отрезке $[l, r)$ теперь будем начинать со `split(r), split(l)`.
И вообще хвостов нигде нет, всегда можно посплитить.

```
vector<Part> p;
void insert(int i, int x):
    i = split(i);
    a[n++] = x; // добавили x в конец исходного массива
    p.insert(i, Part(n - 1, n));
void erase(int i):
    split(i + 1);
    p.erase(split(i));
int getSum(int l, int r): // [l,r)
    int sum = 0;
    for (r = split(r), l = split(l); l < r; l++)
        sum += p[l].sum;
    return sum;
```

Есть небольшая проблема – число кусков после каждого запроса вырастет на $\mathcal{O}(1)$.
Давайте, если число кусок $\geqslant 3\sqrt{n}$, просто вызовем `rebuild` – процедуру, которая выпишет все куски в один большой массив и от него с нуля построит структуру данных.
Время работы такой процедуры $\mathcal{O}(n)$, вызываем мы её не чаще чем раз в $\sqrt{n}$ запросов, поэтому среднее время работы `rebuild` – $\mathcal{O}(\sqrt{n})$ на запрос.

27.6.4. Применение

Задачи про минимум, сумму мы уже умели решать через BST, все операции за $\mathcal{O}(\log n)$.
Из нового мы пока научились только запросы сумма/изменение "перекашивать":
$\langle \mathcal{O}(\log n), \mathcal{O}(\log n) \rangle \longrightarrow \langle \mathcal{O}(\sqrt{n}), \mathcal{O}(1) \rangle$ и $\langle \mathcal{O}(1), \mathcal{O}(\sqrt{n}) \rangle$.
На самом деле спектр задач, решаемых корневой оптимизацией гораздо шире. Для примера приведём максимально ужасную задачу. Выполнять нужно следующие запросы:

1. `insert(i,x)`; `erase(i)`
2. `min(l,r)`
3. `reverse(l,r)`; `add_to_all(l,r,x)`
4. `sum(l,r,x,y)`; `kth_stat(l,r,k)`

Здесь `kth_stat(l,r,k)` – k-я статистика на отрезке. Бинпоиском по ответу такой запрос сводится к задаче вида `sum(l,r,x,y)` – число элементов на $[l, r)$ со значением от x до y. Чтобы отвечать на запрос `sum(l,r,x,y)` для каждого куска будем хранить его сортированную версию, тогда ответ на запрос – обработка двух хвостов за $\mathcal{O}(\sqrt{n})$ и $2\sqrt{n}$ бинпоисков. Итого $\sqrt{n} \log n$.

Чтобы отвечать на запросы `reverse(l,r)` и `add_to_all(l,r,x)` для каждого куска будем хранить две отложенных операции – `is_reversed` и `value_to_add`. Как пример, код `reverse(l,r)`:

```
1 void reverse(l, r):
2     r = split(r), l = split(l);
3     reverse(p + l, p + r);
4     for (; l < r; l++)
5         p[l].is_reversed ^= 1;
```

Единственное место, где будет использоваться `is_reversed` – `split` куска.
Если мы хотим решать задачу через `split/merge`, чтобы выполнять операцию `reverse`, всё равно придётся добавить `split(i)`. Теперь можно делать `reverse(l,r)` ровно, как описано выше, после чего при наличии слишком маленьких кусков, сделаем им "`merge` с соседним".

27.6.5. Оптимальный выбор k

Не во всех задачах выгодно разбивать массив ровно на $\sqrt{n}$ частей по $\sqrt{n}$ элементов.
Обозначим число кусков k. В каждом куске $m = \frac{n}{k}$ элементов.
На примере последней задачи оптимизируем k.

• split/rebuild

Время `inner_split` куска: $\mathcal{O}(m \log m)$[2], так как нам нужно сортировать.
Время `split(i)`: $\mathcal{O}(k) +$ `inner_split` $= \mathcal{O}(k + m \log m)$.
Время `reverse` и `add_to_all`: $\mathcal{O}(k) +$ `split(i)` $= \mathcal{O}(k + m \log m)$.
Время `insert` и `erase`: $\mathcal{O}(k) + \mathcal{O}(m)$.
Время `sum`: хвосты и бинпоиски в каждом куске $= \mathcal{O}(k \log m) + \mathcal{O}(m)$.

[2] при большом желании можно за $\mathcal{O}(m)$

Суммарное время всех запросов равно $\mathcal{O}((k+m)\log m)$.
В худшем случае нам будут давать все запросы по очереди $\Rightarrow$ эта асимптотика достигается.

Вспомним про `rebuild`! В этой задаче он работает за $\mathcal{O}(k(m\log m)) = \mathcal{O}(n\log n)$.
И вызывается он каждые k запросов (мы оцениваем только асимптотику, константу не пишем).

Итого: $\texttt{T(split)} + \texttt{T(insert)} + \texttt{T(sum)} + \cdots + \frac{1}{k}\texttt{T(rebuild)} = \Theta((k+m)\log m + \frac{1}{k}n\log n) \to \min$

При минимизации таких величин сразу замечаем, что "все log-и асимптотически равны".
$\frac{1}{k}n = m \Rightarrow$ минимизировать нужно $(\frac{n}{k} + k)\log n$. При минимизации $\frac{n}{k} + k$ мы не будем дифференцировать по k. Нас интересует только асимптотика, а $\Theta(f+g) = \Theta(\max(f,g))$.
Одна величина убывает, другая возрастает $\Rightarrow$ достаточно решить уравнение $\frac{n}{k} = k$.
Итого: $k = \sqrt{n}$, среднее время работы одного запроса $\mathcal{O}(\sqrt{n}\log n)$.

• **split/merge**

В этом случае всё то же, но нет `rebuild`.
Предположим, что мы умеем делать `inner_split` и `inner_merge` за $\mathcal{O}(m)$.
Тогда нам нужно минимизировать $\texttt{T(split)} + \texttt{T(insert)} + \texttt{T(sum)} + \cdots = \Theta(m + k\log m)$
Заменили $\log m$ на $\log n$, сумму на максимум $\Rightarrow$ решаем $\frac{n}{k} = k\log n$. Итого $k = \sqrt{n/\log n}$.

27.6.6. Корневая по запросам, отложенные операции

Задача: даны числа, нужно отвечать на запросы `lower_bound`. Самое простое и быстрое решение — отсортировать числа, на сортированном массиве вызывать стандартный `lower_bound`.

Задача: всё то же, но нужно ещё и добавлять новые числа.

Решение: отложенные операции, разобрано на 29-й странице осеннего конспекта.
Решение работает в online. Тем не менее, мы как будто обрабатываем запросы пачками по $\sqrt{n}$.

Другой пример на ту же тему – решение задачи dynamic connectivity в offline.
Задача: дан неорграф. Есть три типа запросов – добавить ребро в граф, удалить ребро, проверить связность двух вершин. Нужно в offline обработать m запросов.
Решение: обрабатывать запросы пачками по $\sqrt{m}$. Подробно описано в разборе практики.

27.7. (*) Другие деревья поиска

В мире есть ещё много деревьев поиска, которые не охватывает курс.
Отдельное внимание хочется обратить на

`[Finger tree]` – чисто функциональный rope, умеющий обращаться к концам за $\mathcal{O}(1)$.
`[Finger Search tree]` – дерево поиска, которое почти всё делает за амортизированное $\mathcal{O}(1)$.
`[Tango tree]` – $\mathcal{O}(\log\log n)$ динамически оптимальное дерево.
`[Size Balanced Tree]`.

Лекция #28: Дерево отрезков

28-я пара, 2024/25

Дерево отрезков (range tree) – это и структура данных, и мощная идея. Как структура данных, она не даёт почти ничего асимптотически нового по сравнению с BST. Основные плюсы дерева отрезков – малые константы времени и памяти, а также простота реализации.
Поэтому наш рассказ начнётся с самой эффективной реализации дерева отрезков – «снизу».

28.1. Общие слова

Дерево отрезков строится на массиве. Каждой вершине соответствует некоторый отрезок массива. Обычно дерево отрезков хранят, как массив вершин, устроенный, как бинарная куча:

`root = 1, parent[v] = v / 2, leftSon[v] = 2 * v, rightSon[v] = 2 * v + 1.`

Если вершине v соответствует отрезок $[vl, vr]$ [3], её детям будут соответствуют отрезки $[vl, vm]$ и $(vm, vr]$, где `vm = (vl+vr)/2` [4]. Листья дерева – вершины с $vl = vr$, в них хранятся элементы исходного массива. Сам массив нигде кроме листьев обычно не хранится.

В вершинах дерева отрезков хранится некая фукнция от отрезка массива. Простейшие функции – `min`, `sum` чисел на отрезке, но хранить можно совершенно произвольные вещи. Например, `set` различных чисел на отрезке. Единственное ограничение на функцию: зная только значения функции в двух детях, мы должны иметь возможность посчитать функцию в вершине.

Дерево отрезков используют, чтобы вычислять значение функции на отрезке.
Обычно (но не всегда!) дерево отрезков допускает модификацию в точке, на отрезке.

28.2. Дерево отрезков с операциями снизу

```
1  void build(int n, int a[]): // O(n)
2      t.resize(2 * n) // нам понадобится не более 2n ячеек
3      for (i = 0; i < n; i++)
4          t[i + n] = a[i] // листья дерева находятся в ячейках [n..2n)
5      for (i = n - 1; i > 0; i--)
6          t[i] = min(t[2 * i], t[2 * i + 1]) // давайте хранить минимум
7
8  int getMin(int l, int r): // O(log(r−l+1))
9      int res = INT_MAX; // нейтральный элемент относительно операции
10     for (l += n, r += n; l <= r; l /= 2, r /= 2):
11         // 1. Сперва спустимся к листьям: l += n, r += n
12         // 2. Вершины кроме, возможно, крайних, разбиваются на пары (общий отец)
13         // 3. Отрежем вершины без пары - нечётный l и чётный r
14         // 4. Перейдём к отрезку отцов [l/2, r/2]
15         if (l % 2 == 1) res = min(res, t[l++])
16         if (r % 2 == 0) res = min(res, t[r--])
17     return res;
```

Lm 28.2.1. Время работы `get` на отрезке $[l, r]$ равно $\mathcal{O}(1 + \log(r - l + 1))$

То есть, мало того, что `get` не рекурсивен, он ещё и на коротких отрезках работает за $\mathcal{O}(1)$.

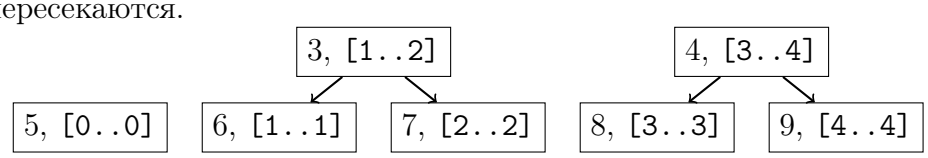
[3] Можно писать на полуинтервалах. Я сам пишу и рекомендую именно полуинтервалы.
[4] Если vl и vr могут быть отрицательными или больше $\frac{1}{2}$`INT_MAX`, вычислять vm следует, как `vl+(vr-vl)/2`.

```
int change(int i, int x) {
    t[i += n] = x; // обновили значение в листе
    // пересчитали значения во всех ячейках на пути до корня (все отрезки, содержащие i)
    for (i /= 2; i >= 1; i /= 2)
        t[i] = min(t[2 * i], t[2 * i + 1]);
}
```

Время работы – $\mathcal{O}(\log n)$. И снова без рекурсии.

Казалось бы `get` и `change` корректно решают задачу «минимум на меняющемся массиве». В чём подвох? Пора обратить внимание на то, что «реализация дерева отрезков снизу» не является деревом, она является лесом (см. картинку). Этот лес состоит из слоёв. Для корректности нужно доказать, что слои не пересекаются.

Дерево отрезков для $n = 5$:



Lm 28.2.2. Корректность.

Доказательство. Нижний слой – исходный массив, он хранится в ячейках $[l_0, r_0) = [n, 2n)$. Сразу заметим $r_0 \leqslant 2l_i$ и докажем по индукции $r_i \leqslant 2l_i$ Слой отцов слоя $[l_i, r_i)$ обозначим за $[l_{i+1}, r_{i+1})$. Читая `get`, видим что $[l_{i+1}, r_{i+1}) == \left[\lceil \frac{l}{2} \rceil, \lfloor \frac{r}{2} \rfloor\right)$. Слои не перескаются $\Leftrightarrow r_{i+1} \leqslant l_i \Leftrightarrow \lfloor \frac{r}{2} \rfloor \leqslant l_i \Rightarrow r_i \leqslant 2l_i$. Вторая часть: $r_{i+1} \leqslant 2l_{i+1} \Leftrightarrow \lfloor \frac{r}{2} \rfloor \leqslant 2\lceil \frac{l}{2} \rceil \Rightarrow r_i \leqslant 2l_i$. ■

Функция `change` будет иногда ходить по бесполезным ячейкам. Это ничему не мешает: главное, что все ячейки с полезной информацией change корректно пересчитал.

28.3. Дерево отрезков с операциями сверху

Дерево отрезков с операциями сверху – гораздо более естественная структура.
Корень – отрезок $[0..n)$, далее дерево строится рекурсивно по определению (Section 28.1).
Все функции работы с деревом, включая построение, – рекурсивные функции спуска. Пример:

```
int getMin( int v, int vl, int vr, int l, int r ) {
    if (vr < l || r < vl) return INT_MAX; // не пересекаются
    if (l <= vl && vr <= r) return t[v]; // вершина целиком внутри запроса
    int vm = (vl + vr) / 2;
    int fl = getMin(2 * v, vl, vm, l, r);
    int fr = getMin(2 * v + 1, vm + 1, vr, l, r);
    return min(fl, fr);
}
int result = getMin(1, 0, n - 1, l, r); // 1 = root
```

Здесь показана версия, в которой отрезок вершины $[vl, vr]$ не хранится, как свойство вершины, а вычисляется по ходу спуска сверху вниз. Версия с хранением ни чем не лучше – время пересчёта vl, vr сопоставимо с временем обращения к памяти для чтения уже посчитанной величине.

Операцию change можно сделать рекурсивно сверху.
А можно округлить n вверх до 2^k и сделать get сверху, change снизу. Правда преимуществ нет...

Чтобы пользоваться деревом «сверху» осталось его построить, а в построении – главное выделить массив нужной длины. При $n = 2^k$ получается полное бинарное дерево из $2^{k+1} = 2n$ вершин. Если не округлять n до 2^k, все индексы лишь уменьшатся.

Lm 28.3.1. $4n$ ячеек достаточно, более точно выделить $2^{1+\lceil \log_2 n \rceil}$ ячеек.

Теперь оценим время работы запроса `getMin`.

Lm 28.3.2. `getMin` посетит не более $4 \log n$ вершин дерева.

Доказательство. Уровней в дереве $\log n$. На каждом уровне мы посетим не более четырёх вершин, потому что только в двух вершинах предыдущего уровня мы ушли в рекурсию – в вершинах, которые содержали края отрезка $[l, r]$. ■

Lm 28.3.3. `getMin` разбивает любой отрезок $[l, r]$ на не более чем $2 \log n$ вершин дерева отрезков.

Доказательство. Уровней в дереве $\log n$, на каждом уровне мы выберем не более 2 вершин. ■

Последняя лемма верна и для реализации «снизу». По тем же причинам.
Решая задачи, часто удобно думать про дерево отрезков так:

Мы даём дереву отрезков $[l, r]$, а оно разбивает отрезок $[l, r]$
на $\leqslant 2 \log n$ вершин, для которых уже посчитана полезная функция.

Минусы по сравнению с реализацией «снизу»:

(a) Памяти нужно $4n$ вместо $2n$.
(b) `get` почти всегда за $\mathcal{O}(\log n)$, даже для отрезков длины $\mathcal{O}(1)$.
(c) Из-за рекурсии больше константа.

Зато есть много плюсов, главный из них: также, как и в BST, можно делать «модификацию на отрезке». «Все элементы на отрезке $[l, r]$ увеличить на x». «Всем элементам присвоить x».

Модификация на отрезке делается *отложенными операциями*. Если в вершине v хранится отложенная операция, проходя через v сверху вниз, важно не забыть эту операцию протолкнуть вниз. Проталкивание вниз назовём `push`. Пример функции присваивания на отрезке:

```
1  void push(int v) {
2      if (value[v] == -1) return; // нет отложенной операции
3      value[2 * v] = value[2 * v + 1] = value[v];
4      value[v] = -1;
5  }
6  void setValue(int v, int vl, int vr, int l, int r, int x) {
7      if (vr < l || r < vl) return INT_MAX; // не пересекаются
8      if (l <= vl && vr <= r) { // вершина целиком внутри запроса
9          value[v] = t[v] = x; // не забываем пересчитывать минимум в вершине
10         return;
11     }
12     push(v);
13     int vm = (vl + vr) / 2;
14     setValue(2 * v, vl, vm, l, r, x);
15     setValue(2 * v + 1, vm + 1, vr, l, r, x);
16     // 1. Сейчас в нашей вершине отложенной операции нет, мы её толкнули вниз
17     // 2. При написании кода важно заранее решить t[v] - минимум с учётом value[v] или без
18     t[v] = min(t[2 * v], t[2 * v + 1]); // у нас с учётом, т.к. под min ставится t[2 * v]
19 }
```

Ещё некоторые плюсы реализации сверху:

(a) Дерево отрезков сверху – реально дерево!
(b) Дерево отрезков сверху можно сделать динамическим (следующий раздел).
(c) Дерево отрезков сверху можно сделать персистентным (следующий раздел).

28.4. (*) Хаки для памяти к дереву отрезков сверху

• Проблема

Если $n = 2^k$ – проблем нет. Памяти $2n$.
Проблема, это, например, $n = 2^k + 2$, тогда дети корня имеют размер $2^{k-1} + 1 \Rightarrow$ глубину $k \Rightarrow$ правый из детей будет использовать ячейку $2^{k+1} + 2^k \approx 3n$.
Бывает ещё хуже. Пусть $n = 2^k + 4 \Rightarrow$ дети $2^{k-1} + 2 \Rightarrow$ внуки $2^{k-2} + 1 \Rightarrow$ нужна ячейка для правого внука $2^{k+1} + 2^k + 2^{k-1} \approx 3.5n$.

Lm 28.4.1. Обычная реализация сверху при выборе `vm = (vl + vr) / 2` и с округлением вверх, и с округлением вниз может привести к использованию ячеек с номером $4n - \Theta(\sqrt{n})$.

Доказательство. Рассмотрим $n = 2^k + 2^i \Rightarrow$ на нижнем уровне используем ячейку $2^{k+1} + 2^k + \cdots + 2^{k-i+1} = 2^{k+2} - 2^{k-i+1}$. Возьмём $i = k/2$, получим $n \to 4n - \Theta(\sqrt{n})$. ■

• Решение

Пусть мы v, тогда левый сын – $v + 1$. Правый сын? v+1+число вершин в левом поддереве. Пусть $[vl, vr) \to [vl, vm) + [vm, vr) \Rightarrow$ слева ровно $(vm - vl) \cdot 2 - 1 \Rightarrow$ правый сын v это $v + 2(vm - vl)$.

28.5. Динамическое дерево отрезков и сжатие координат

Пусть наш массив длины $M = 10^{18}$ и изначально заполнен нулями.
Есть два способа реализовать на таком массиве дерево отрезков.

Первый способ элегантный, не содержит лишнего кода, но резко увеличивает константу времени работы: давайте все массивы заменим на `unordered_map`-ы. При этом `t[]` заменим на `unordered_map<int, T>`, где `T` – специальный тип, у которого конструктор создаёт нейтральный относительно нашей операции элемент.

Замечание 28.5.1. Первый способ работает и для версии снизу, и для версии сверху.

Второй способ предлагает от хранения в массиве перейти к ссылочной структуре:

```
1 struct Node {
2     Node *l, *r;
3     int min;
4 };
```

Вершина дерева такая же, как и в BST. Вершины можно создавать лениво.
Изначально всё дерево отрезков состоит из `Node* root = NULL`.
Все запросы спускаются сверху вниз и, если попадают в `NULL`, создают на его месте `Node`.

Lm 28.5.2. Время работы любого запроса $\mathcal{O}(\log M)$.

Lm 28.5.3. После k запросов создано $\mathcal{O}(\min(M, k \log M))$ `Node`-ов.

Lm 28.5.4. После k запросов создано $\mathcal{O}(k \cdot \max(1, \log M - \log k)))$ `Node`-ов.

Динамические деревья отрезков незаменимы при решении таких задач, как «k-я статистика на отрезке за $\mathcal{O}(\log n)$». Также их удобно применять в качестве внешнего дерева в 2D-деревьях.

• Сжатие координат

Динамическое дерево отрезков засчёт большей глубины и ссылочной структуры медленней обычного. Использует больше памяти $\Rightarrow$ реже кешируется. Поэтому, когда есть возможность,

для решения исходной задачи применяют не его, а *сжатие координат*.

Задача: в offline обработать n запросов «изменение в точке i_j» и «сумма на отрезке $[l_j, r_j)$» над массивом длины M.

Сжатие координат.
Offline $\Rightarrow$ все запросы известны заранее, на $[0, M)$ не более $3n$ интересных точек: i_j, l_j, r_j.
Давайте, сложим их в массив, отсортируем и заменим на позиции в этом массиве.
То есть, перенумеруем числами из $[0, 3n)$. Свели исходную к задаче на массиве длины $3n$.

28.6. Персистентное дерево отрезков, сравнение с BST

Также, как динамическое дерево отрезков, пишется на указателях.

Также, как в персистентном BST, все запросы сверху. Также, как в персистентном BST, `Node`-ы нельзя менять, при желание менять всегда придётся создавать новую вершину.

Все операции, которые умело дерево отрезков, всё ещё работают. За то же время.

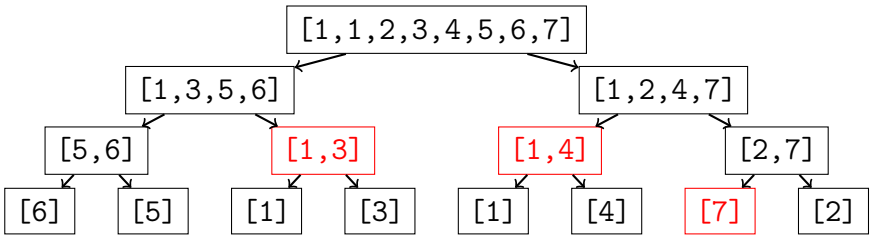
• Сравнение с персистентным BST

Персистентное BST по неявному ключу, конечно, умеет больше:
`insert`, `delete`, `split/merge/reverse`.

Если же всё, что мы хотим, – реализовать простейший персистентный массив ($a_i = x$, $x = a_i$), то у BST всё равно есть преимущество – оно хранит ровно n `Node`-ов, а дереву отрезков нужно от $2\log n$ до $4\log n$. Чтобы время работы у BST было не хуже, если мы не собираемся менять его структуру (нет `insert` и прочего), создадим его сразу идеально сбалансированным.

28.7. 2D-запросы, дерево сортированных массивов

Для начала попробуем сохранить в вершине дерева отрезков сортированную копию отрезка и посмотрим, что получится. На картинке дерево отрезков для массива `[6,5,1,3,1,4,7,2]`.



Как мы помним (Lm 28.3.3), дерево отрезков разбивает любой отрезок на $\mathcal{O}(\log n)$ вершин дерева отрезков. Красным выделены вершины, на которые распадётся отрезок $[2, 7)$.

Задача #1: дан массив длины n, отвечать на запросы `get(L,R,D,U)`: число элементов на отрезке $[L, R]$, значения которых от D до U, то есть, $\#\{i\colon L \leqslant i \leqslant R \wedge D \leqslant a_i \leqslant U\}$.

Задача #2: даны n точек (x_i, y_i) на плоскости, отвечать на запросы «число точек в прямоугольнике», то есть, $\#\{i\colon X_1 \leqslant x_i \leqslant X_2 \wedge Y_1 \leqslant y_i \leqslant Y_2\}$.

Мы описали так называемые 2D-запросы на массиве и на плоскости.

Теорема 28.7.1. Описанные выше задачи равносильны.

Доказательство. Если есть массив a_i, можно обозначить $(x_i, y_i) = (i, a_i)$. В другую сторону: отсортируем точки по x_i, теперь двумя бинпоисками условие вида $X_1 \leqslant x_i \leqslant X_2$ можно превратить в равносильное $L \leqslant i \leqslant R$. ■

Обе задачи решаются деревом отрезков сортированных массивов за $\mathcal{O}(\log^2 n)$ на запрос[5].
Решим задачу #1, вторая к ней сводится. Отрезок $[L, R]$ разделится на $\mathcal{O}(\log n)$ вершин, на каждой из них сделаем два бинпоиска, вернём $\sum$ по вершинам «`upper_bound(U)-lower_bound(D)`».

Время построения дерева отрезков сортированных массивов – $\mathcal{O}(n \log n)$, так как каждая вершина получается, как `merge` своих детей, который считается за линейное время.

• k-я статистика на отрезке за $\mathcal{O}(\log^3 n)$.

Запрос `get(L,R,k)` – вернуть `sorted(a[L..R])[k]`.
Сделаем бинпоиск по ответу. Внутри бинпоиска нам дают x и нужно узнать сколько чисел на $[L, R]$ не более x, что мы только что научились за $\mathcal{O}(\log^2 n)$. Заметим, что бинпоиск по ответу можно реализовать, как бинпоиск по `sorted(a)`, поэтому бинпоиск сделает $\mathcal{O}(\log n)$ итераций.

28.8. Многомерные деревья

• Д.О. of Д.О.

Задача. Даны x_i, y_i, запрос: на отрезке $[l, r]$ среди всех i: $min_x \leqslant x_i \leqslant max_x$ найти min y.

Решение. В дереве отрезков сортированных массивов в каждой вершине на сортированном по x_i массиве пар $\langle x_i, y_i \rangle$ построим дерево отрезков с минимумом по y_i. Ответ на запрос: обход внешнего дерева и в $\mathcal{O}(\log n)$ вершинах, делящих $[l, r]$, сперва бинпоиск, затем запрос к внутренному дереву отрезков (итого $\mathcal{O}(\log^2 n)$ на запрос). Память и время построения $\mathcal{O}(n \log n)$.

• Д.О. of treap

Дерево отрезков сортированных массивов — структура данных на статичном (не меняющемся) массиве. Если добавить запросы изменения массива, «`a[i] = x`», то нам в каждой вершине дерева отрезков нужен динамический аналог сортированного массива. Например, treap.
Получили «дерево отрезков декартовых деревьев» и $\mathcal{O}(\log^2 n)$ на запрос.

• 3D-запросы

В вершине дерева отрезков можно хранить вообще всё, что душе угодно. Например, 2D-дерево. Рассмотрим 3D-запрос на плоскости: даны n точек (x_i, y_i, z_i), нужно отвечать на запросы $\#\{i\colon X_1 \leqslant x_i \leqslant X_2 \wedge Y_1 \leqslant y_i \leqslant Y_2 \wedge Z_1 \leqslant z_i \leqslant Z_2\}$. Будем решать задачу также, как уже решили задачу про 2D-запросы: отсортируем точки по x_i, на полученном массиве построим дерево отрезков. Пусть вершине v дерева отрезков соответствует отрезок $[v_l, v_r]$. Чтобы ответить на исходный запрос, в вершине v нашего дерева отрезков достаточно иметь структуру данных, которая умеет отвечать на 2D-запросы для множества 2D-точек (y_i, z_i) из отрезка $[v_l, v_r]$. Итого $\mathcal{O}(\log^3 n)$ на запрос и $\mathcal{O}(n \log^2 n)$ памяти.

Аналогично можно на k-мерный запрос отвечать за $\mathcal{O}(\log^k n)$. Памяти будет использовано $\mathcal{O}(n \log^{k-1} n)$. При $k > 3$ это не эффективно.

[5] В Section 28.13 мы соптимизируем время до $\mathcal{O}(\log n)$

• Динамичный аналог

Пусть исходные координаты точек — целые от 0 до $C \approx 10^9$. Сейчас для решения задачи нам нужно постоянно сортировать массивы, жать координаты. Есть более простой, но дорогой по памяти подход: все деревья отрезков на всех уровнях делать динамическими.

Плюсы: можно легко добавлять новые точки, проще пишется, не нужны бинпоиски.

Минусы: деревья на указателях, $\mathcal{O}(n \log^k C)$ вместо $\mathcal{O}(n \log^{k-1} n)$ памяти.

28.9. Предверие сканирующей прямой

Давайте для каждого префикса массива насчитаем BST. $T_i = \{a_0, a_1, \dots a_{i-1}\}$. Если BST персистентно, нам хватит $\mathcal{O}(n \log n)$ памяти. А через FAT-nodes даже $\mathcal{O}(n)$ памяти. После этого мы умеем отвечать на запрос «сколько на отрезке $[l, r)$ чисел от $\leqslant x$» – ответ $= get(T_r, x) - get(T_l, x)$.

• НВП

Давайте возьмём простейшую квадратную динамику для НВП: $f[i] = 1 + \max_{j<i, a_j<a_i} f_j$.
Как за $\mathcal{O}(\log n)$ находить такой максимум? Давайте будем хранить BST по ключу a_j для префикса $[0, i)$. Такое BST умеет выбрать $\max f_j$ по всем $a_j < x$. Конец. Можно улучшить константу: сделать сжатие координат и использовать одно одномерное дерево отрезков на максимум. Обновляем на каждом шаге наше дерево так: $t[a_i] = f_i$, и пересчитать максимумы до корня.

28.10. Сканирующая прямая

Идея сканирующей прямой (scanline, sweep line, заметающая прямая) пришла из вычислительной геометрии и в самом общем виде звучит так: пусть на плоскости есть какие-то объекты, нарисуем вертикальную прямую и будем двигать её слева направо, обрабатывая по ходу движения события вида «объект начался», «объект закончился» и иногда «объект изменился».

Нам уже встречалась одномерная версия той же идеи: на прямой даны n точек и m отрезков, для каждого отрезка нужно узнать число точек внутри, для каждой точки узнать, сколькими отрезками она покрыта. Решение: идём слева направо, обрабатываем события «отрезок начался», «отрезок закончился», «точка».

2D случай. Даны n точек и m прямоугольников со сторонами параллельными осям координат.

Задача #1: для каждой точки посчитать, сколькими прямоугольниками она покрыта.
Решение: идём слева направо, встречаем и обрабатываем следующие события

(a) Прямоугольник начался: сделаем `count[y1..y2] += 1;`
(b) Прямоугольник закончился: сделаем `count[y1..y2] -= 1;`
(c) Встретили точку, тогда в `count[y]` хранится ровно число открытых ещё незакрытых прямоугольников, её покрывающих.

Дерево отрезков на массиве `count` справится с обеими операциями за $\mathcal{O}(\log n)$.
Итого время работы = сортировка + scanline = $\mathcal{O}((n+m)\log(n+m))$.

Задача #2: для каждого прямоугольника посчитать число точек внутри.
Сразу заметим, что прямоугольник можно разбить на два горизонтальных стакана:
количество точек в области $\{(x, y)\colon X_1 \leqslant x \leqslant X_2 \wedge Y_1 \leqslant y \leqslant Y_2\}$ равно разности
количеств в областях $\{(x, y)\colon x \leqslant X_2 \wedge Y_1 \leqslant y \leqslant Y_2\}$ и $\{(x, y)\colon x \leqslant X_1 - 1 \wedge Y_1 \leqslant y \leqslant Y_2\}$.
Итого осталось решить задачу для n точек и $2m$ горизонтальных стаканов.
Решение: идём слева направо, встречаем и обрабатываем следующие события:

(a) Встретили точку, сделаем `count[y] += 1;`
(b) Встретили конец стакана, посчитали $\sum_{y \in [y_1..y_2]}$ `count[y]`.

Дерево отрезков на массиве `count` справится с обеими операциями за $\mathcal{O}(\log n)$.

• Решение online версии.

Пусть теперь заранее известны только точки и в online приходят запросы-прямоугольники, для прямоугольника нужно посчитать число точек внутри. Возьмём решение задачи #2, дадим ему n точек и 0 стаканов. Теперь по ходу scanline-а мы хотим сохранить все промежуточные состояния дерева отрезков. Для этого достаточно сделать его персистентным. Асимптотика времени работы не изменилась (константа, конечно, хуже). Памяти теперь нужно $\mathcal{O}(n \log n)$.

Пусть `root[i]` – версия дерева до обработки события с координатой `x[i]`,
тогда запрос `get(x1, x2, y1, y2)` обработаем так:

```
return root[upper_bound(x, x + n, x2) - x].get(y1, y2) -
       root[lower_bound(x, x + n, x1) - x].get(y1, y2);
```

Итого: используя предподсчёт за $\mathcal{O}(n \log n)$, мы умеем за $\mathcal{O}(\log n)$ отвечать на 2D-запрос на плоскости. Из Thm 28.7.1 мы также умеем за $\mathcal{O}(\log n)$ обрабатывать 2D-запрос на массиве.

Важно запомнить, что любой «scanline с деревом отрезков» для решения offline задачи можно приспособить для решения online задачи, сделав дерево отрезков персистентным.

28.11. k-я порядковая статистика на отрезке

Мы уже умели бинпоиском по ответу искать k-ю статистику за $\mathcal{O}(\log^3 n)$.
Поскольку отвечаем на 2D-запросы мы теперь за $\mathcal{O}(\log n)$, это же решение работает за $\mathcal{O}(\log^2 n)$.
Перед тем, как улучшить $\mathcal{O}(\log^2 n)$ до $\mathcal{O}(\log n)$ решим вспомогательную задачу:

• Бинпоиск $\to$ спуск по дереву.

Задача: дан массив из нулей и единиц, нужно обрабатывать запросы «`a[i]=x`» и «k-я единица».

Решение за $\mathcal{O}(\log^2 n)$.
На данном нам массиве будем поддерживать дерево отрезков с операцией сумма. Чтобы найти k-ю единицу, сделаем бинпоиск по ответу, внутри нужно найти число единиц на префиксе $[0, x)$. Это запрос к дереву отрезков. Например, спуск сверху вниз.

Решение за $\mathcal{O}(\log n)$.
Спускаемся по дереву отрезков: если слева сумма хотя бы k, идём налево, иначе направо.

Мораль.
Внутри бинпоиска есть спуск по дереву $\Rightarrow$ скорее всего, от бинпоиска легко избавиться.

• k-я статистика на отрезке за $\mathcal{O}(\log n)$.

Сейчас у нас есть следующее решение за $\mathcal{O}(\log^2 n)$: возьмём точки (i, a_i), сделаем scanline с персистентным деревом отрезков. Теперь для ответа на запрос `get(l,r,k)`, делаем бинпоиск по ответу, внутри считаем `tree[r+1].get(x) - tree[l].get(x)`, где `tree[i].get(x)` обращается к i-й версии дерева отрезков и возвращает количество чисел не больше x на префиксе $[0, i)$.

Вместо бинпоиска по ответу будем параллельно спускаться по деревьям `tree[r+1]` и `tree[l]`. Пусть мы сейчас стоим в вершинах `a` и `b`. Обеим вершинам соответствует отрезок `[vl..vr]`, если (`a->l->sum - b->l->sum` $\geqslant$ `k`), есть хотя бы k чисел со значениями `[vl..vm]` и мы в обоих деревьях спустимся налево, иначе мы теперь хотим найти (`k - (a->l->sum - b->l->sum)`)-е

число и в обоих деревьях спустимся направо.

28.12. (*) Фенвик

[e-maxx]

Структура данных, умеющая делать изменения в точке и считать функцию на префиксе. Если функция обратима (плюс → минус), можно считать и на отрезке.

```
struct Fenwick {
    int n;
    vector<int> a;

    Fenwick(int n) : n(n), a(n) { }

    // O(logn)
    int get(int i) { // get = [0...i]
        int res = 0;
        // i&(i+1)-1  = 010101010100111111
        // i          = 010101010101011111
        // i+1        = 010101010101100000
        // i&(i+1)    = 010101010101000000
        // i&(i+1)-1  = 010101010100111111
        for (; i >= 0; i &= i + 1, i--) // [x=i&(i+1)...i]
            res += a[i];
        return res;
    }

    // O(logn)
    void change(int i, int d) { // += d
        // i|(i+1)    = 010101010101111111
        // i          = 010101010101011111
        // i+1        = 010101010101100000
        // i|(i+1)    = 010101010101111111
        for (; i < n; i |= i + 1)
            a[i] += d;

        // a[k]
        // k          = 010101010101011111
        // i          = 0101010101010*****
        // k&(k+1)    = 010101010101000000
    }
}
```

28.13. (*) Fractional Cascading

28.13.1. (*) Для дерева отрезков

У нас есть дерево отрезков сортированных массивов. Отвечать на запрос можно сверху или снизу. На скорость это не влияет, т.к. и там, и там мы делаем одни и те же $\mathcal{O}(\log n)$ бинпоисков.

Хотим отвечать на запрос $\#\{i\colon L \leqslant i \leqslant R \wedge a[i] < x\}$.
Будем отвечать на запрос сверху, но бинпоиск сделаем только в корне $root = 1$.
Вместо того, чтобы делать бинпоиски в детях, воспользуемся ответом для отца.
Для этого нам понадобится предподсчёт, который получается лёгкой модификацией `merge`:

```
1  void build( int n, int *a ) {
2      assert(n & (n - 1) == 0) // n = 2^k
3      vector<vector<int>> t(2 * n), l(2 * n);
4      for (int i = 0; i < n; i++)
5          t[n + i] = vector<int>(1, a[i]); // листья дерева
6      for (int i = n - 1; i > 0; i--) {
7          int A = t[2 * i].size(), B = t[2 * i + 1].size();
8          t[i].resize(A + B), l[i].resize(A + B);
9          for (size_t a = 0, b = 0, c = 0; a < L || b < R; ) { // собственно merge
10             l[i][c] = a; // среди первых c элементов t[i] ровно a ушли в левого сына
11             if (a == L || (b < R && t[2 * i][a] > t[2 * i + 1][b]))
12                 t[i][c++] = t[2 * i + 1][b++];
13             else
14                 t[i][c++] = t[2 * i][a++];
15         }
16     }
17 }
```

Если бинпоиск в вершине i вернул x, в детях нам вернут `l[i][x]` и `t[i].size() - l[i][x]`.

Следствие 28.13.1. $\forall k \geqslant 2$ мы улучшили время k-мерного ортогонального запроса с $\log^k n$ до $\log^{k-1} n$, добавив во внутреннее $2D$ дерево fractional cascading.

На самом деле fractional cascading – более общая идея: за $\mathcal{O}(k+\log n)$ сделать бинпоиск сразу по k сортированным массивам. Мы сейчас решили частный случай этой задачи для $\log n$ массивов специального вида. wiki даёт общее описание и ссылки. Общая идея, как в skip list – половину элементов «нижнего списка» толкать вверх в следующий список.

28.13.2. (*) Для k массивов

Задача: даны k отсортированных массивов длины $\leqslant n$, поступает запрос x, сделать на каждом `lowerbound(x)`.

Тривиальное решение: на каждом бинпоиск итого $\mathcal{O}(k \log n)$.

Техникой fractional cascading мы получим $\mathcal{O}(k + \log(nk))$: обозначим массивы $a_1, a_2, \ldots a_k$, каждый второй элемент a_1 добавим в a_2, после этого каждый второй элемент a_2 добавим в a_3 и так далее. Суммарное удлинение других массивов, которое вызовет a_1 равно $\frac{1}{2}|a_1| + \frac{1}{4}|a_1| + \frac{1}{8}|a_1| + \ldots \leqslant |a_1|$ и так для всех a_i. Построение структуры – k раз вызвать merge $\Rightarrow$ время работы $\mathcal{O}(\sum |a_i|)$. По ходу каждого merge запомним для каждой пары $\langle i, j\rangle$, сколько элементов

из $\texttt{a}_i\texttt{[0:j)}$ приехали из исходного $\texttt{a}_i$, а сколько добавлено из других массивов a_m.

Как ответить на запрос? Сделаем один бинпоиск, чтобы получить $\texttt{i = a}_k\texttt{.lowerbound(x)}$.

28.14. (*) КД-дерево

Даны n точек на плоскости, каждая задаётся тройкой (x_i, y_i, w_i) – координатами и весом. Построим за $\mathcal{O}(n \log n)$ структуру данных, которая будет занимать $\mathcal{O}(n)$ памяти и будет уметь для точек в прямоугольной области $(lx_j \leqslant x_i \leqslant rx_j,\ ly_j \leqslant y_i \leqslant ry_j)$ за $\mathcal{O}(\sqrt{n})$ делать все те же операции, что дерево отрезков на отрезке. Добавлять новые точки можно будет корневой по запросам. Удалять, если нет необратимых запросов типа «min на прямоугольнике», также.

Структура: бинарное дерево. В корне прямоугольник $[-\infty, +\infty]\times[-\infty, +\infty]$. На чётном уровне делим вертикальной прямой n точек на $\lfloor \frac{n}{2} \rfloor$ и $\lceil \frac{n}{2} \rceil$, на начётном делим горизонтальной прямой.

Построение за $\mathcal{O}(n \log n)$. Отсортировали и по x, и по y, передали в рекурсию два порядка – и по x, и по y. Время работы кроме исходной сортировки $T(n) = \mathcal{O}(n) + 2T(n/2)$.

Запрос за $\mathcal{O}(\sqrt{n})$. Как и в дереве отрезков 3 варианта – прямоугольник запрос и прямоугольник вершина-дерева могут не пересекаться, вкладываться, нетривиально пересекаться. Время работы $2^{\frac{1}{2}\log n}$, так как на каждом втором уровне рекурсии мы не будем ветвиться.

Все точки в прямоугольнике за $\mathcal{O}(k + \log n)$. k – размер ответа. Давайте в каждой вершине КД-дерева в процессе построения сохраним список всех точек, в порядке и по x, и по $y \Rightarrow$ если прямоугольник-запрос пересекает только одну сторону КД-вершины, мы можем сразу вывести ответ за $\mathcal{O}(k)$. Сколько будет КД-вершин, которые пересекает запрос иначе? На каждом уровне не более 4 (четыре угла запроса) $\Rightarrow$ всего $\mathcal{O}(\log n)$.

Лекция #29: LCA & RMQ

29-я пара, 2024/25

Если речь идёт о структуре данных, у которой есть функция построения (предподсчёт) и умение online отвечать на запросы, обозначение $\langle f(n), g(n)\rangle$ означает, что предподсчёт работает за время $\mathcal{O}(f(n))$, а ответ на запрос за $\mathcal{O}(g(n))$.

29.1. RMQ & Sparse table

Def 29.1.1. *RMQ = Range Minimum Query = запросы минимумов на отрезке.*

Задачу RMQ можно решать на не меняющемся массиве (static) и на массиве, поддерживающем изменения в точке (dynamic). Запросы минимума на отрезке будем обозначать "`get`", изменение в точке – "`change`". Построение структуры – "`build`".
Мы уже умеем решать задачу RMQ несколькими способами:

1. Дерево отрезков: `build` за $\mathcal{O}(n)$, `get` за $\mathcal{O}(\log n)$, `change` за $\mathcal{O}(\log n)$
2. Центроидная декомпозиция: `build` за $\mathcal{O}(n \log n)$, `get` за $\mathcal{O}(LCA)$, static.
3. Корневая: `build` за $\mathcal{O}(n)$, `get` за $\mathcal{O}(\sqrt{n})$, `change` за $\mathcal{O}(1)$.
4. Корневая: `build` за $\mathcal{O}(n)$, `get` за $\mathcal{O}(1)$, `change` за $\mathcal{O}(\sqrt{n})$.

Lm 29.1.2. $\nexists$ структуры данных, поддерживающей `build` за $\mathcal{O}(n)$, `change` и `get` за $o(\log n)$.

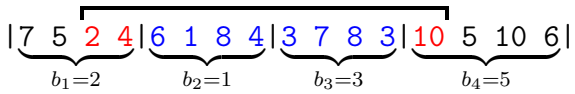
Доказательство. Построим структуру от пар $\langle a_i, i\rangle$, чтобы вместе с минимумом получать и его позицию. После этого n раз достанем минимум, и на его место в массиве запишем $+\infty$. Получили сортировку за $o(n \log n)$. Противоречие. ■

А вот static версию задачи (только get-запросы) мы скоро решим за время $\langle n, 1\rangle$.

• Sparse Table

Пусть `f[k,i]` – минимум на отрезке $[i, i+2^k)$. Массив `f[]` можно предподсчитать за $n \log n$:
`f[0]` – исходный массив, `f[k,i] = min(f[k-1,i], f[k-1,i+2^{k-1}])`.
Теперь, `get` на `[l,r) = min(f[k,l], f[k,r-2^k])`, где $2^k \leqslant r - l < 2^{k+1}$.
Чтобы за $\mathcal{O}(1)$ найти такое k, используем предподсчёт "`log`", теперь `k = log[r-l]`.
Итого получили решение static RMQ за $\langle n \log n, 1\rangle$.

• Sparse Table++



Разобьём исходный массив a на куски длины $\log n$, минимум на i-м куске обозначим b_i.
$\forall$ отрезок a, который содержит границу двух кусков, разбивается на отрезок b и два "хвоста". Минимум на хвосте – это минимум на префиксе или суффиксе одного куска, все такие частичные минимумы предподсчитаем за $\mathcal{O}(n)$. Чтобы, находить минимум на отрезке массива b, построим на b `Sparse Table`, который весит $\frac{n}{\log n} \log \frac{n}{\log n} \leqslant n$. Получили $\langle n, 1\rangle$ решение.

Осталось решить для отрезков, попадающих целиком в один из кусков.
Давайте на каждом куске построим структуру данных для решения RMQ.

Наивный цикл	$\langle n, \log n \rangle$	константа сверхмала
Sparse Table	$\langle n \log\log n, 1 \rangle$	$\frac{n}{\log n} \cdot (\log n \log\log n)$
Дерево отрезков	$\langle n, \log\log n \rangle$	
Рекрсивно построить себя	$\langle n \log^* n, \log^* n \rangle$	$T(n) = 1 + T(\log n)$, $M(n) = n + (n/\log n)(\log n + \dots)$

29.2. RMQ за $\mathcal{O}(1)$ одним махом

Возьмём разбиение на куски длины 32, построим сверху `Sparse Table`. Для кусков длины 32 сделаем предподсчёт: пройдём слева направо, поддерживая стек элементов, которые являются **минимумами** на суффиксах и маску, какие элементы принадлежат этому стеку, для каждого префикса запомним маску.

Пример: кусок $[7, 3, 5, 4, 6, 5, 2, 6]$.
Состояние стека: $[7] \to [3] \to [3, 5] \to [3, 4] \to [3, 4, 6] \to [3, 4, 5] \to [2] \to [2, 6]$. Маски для соответствующих стеков: `1`, `01`, `011`, `0101`, `01011`, `010101`, `0000001`, `00000011`.

Предподсчёт стеков и масок работает линейное время. Суммарное время на предподсчёт: $n + \frac{n}{32} \log n + \frac{n}{32} \cdot 32 = \mathcal{O}(n)$. Тем, что куски длины именно 32, мы пользовались только, чтобы на любой машине все операции с масками происходили за $\mathcal{O}(1)$.

Как теперь ответить на запрос? Любому подотрезку $[L, R]$ куска длины 32 соответствует маска префикса $[0, R]$, у которой нужно отбросить первые L бит и найти крайнюю левую единицу. Нахождение крайней левой единицы, это или предподсчёт, или одна процессорная инструкция.

29.3. LCA & Двоичные подъёмы

В дереве с корнем для двух вершин можно определить отношение "a – предок b".
Более того, на запрос `isAncestor(a, b)` легко отвечать за $\mathcal{O}(1)$.
Предподсчитаем времена входа выхода dfs-ом по дереву, тогда:

```
bool isAncestor(int a, int b):
    return t_in[a] <= t_in[b] && t_out[b] <= t_out[a];
```

Def 29.3.1. *$LCA(a, b)$ – общий предок вершин a и b максимальной глубины.*

LCA = least common ancestor = наименьший общий предок.

Мы уже умеем искать LCA за $\Theta(dist(a, b))$: предподсчитаем глубины всех вершин, при подсчёте LCA сперва уравняем глубины a и b, затем будем параллельно подниматься на 1 вверх.

> Соптимизируем эту идею – научимся $\forall v, k$ из вершины v прыгать сразу на 2^k вверх.
> `up[k,v] = up[k-1,up[k-1,v]]` – прыжок на 2^k равен двум пряжкам на 2^{k-1}.
> База: `up[0,v] = parent[v]`.
> Переход: если уже построен слой динамики `up[k-1]`, мы за $\Theta(n)$ насчитаем слой `up[k]`.
> Чтобы не было крайних случаев, сделаем `up[0,root] = root`.

LCA по-прежнему состоит из двух частей – уравнять глубины и прыгать вверх:

```
int K = ⌈log2 N⌉, up[K][N];
int LCA(int a, int b) {
    if (depth[a] < depth[b]) swap(a, b);
    a = jump(a, depth[a] - depth[b]);
    for (int k = K - 1; k >= 0; k--)
```

```
6         if (up[k][a] != up[k][b])
7             a = up[k][a], b = up[k][b];
8     return a == b ? a : up[0][a];
9 }
```

Здесь `jump(v,d)` за $\mathcal{O}(\log n)$ прыгает вверх из v на d, для этого d нужно представить, как сумму степеней двойки. Время и память предподсчёта – $\Theta(n \log n)$, время поиска LCA – $\Theta(\log n)$.
Можно уменьшить константу времени работы, используя `isAncestor`:

```
1 int LCA(int a, int b) {
2     for (int k = K - 1; k >= 0; k--)
3         if (!isAncestor(up[k][a], b))
4             a = up[k][a];
5     return isAncestor(a, b) ? a : up[0][a];
6 }
```

29.4. RMQ±1 за $\langle n, 1 \rangle$

Def 29.4.1. *Говорят, что массив обладает ±1 свойством, если* $\forall i \ |a_i - a_{i+1}| = 1$

Наша цель – решить RMQ на ±1 массиве.
Подкрутим уже имеющуюся у нас идею из Sparse Table++.
"*Разобьём исходный массив* `a` *на куски длины* $k = \frac{1}{2} \log n$, *минимум на* i*-м куске обозначим* b_i."
$\min(a_1, a_2 \dots, a_k) = a_1 + \min(0, a_2 - a_1, a_3 - a_1, \dots, a_k - a_1) = a_1 + X$.
Поскольку $\forall i \ |a_{i+1} - a_i| = 1$, X – минимум на одной из 2^{k-1} последовательностей.
$2^k = 2^{\log n/2} = \sqrt{n} \Rightarrow$ можно за $o(n)$ предподсчитать ответы для всех последовательностей.

29.5. LCA → RMQ±1 и Эйлеров обход

Напомним, эйлеров обход графа – цикл, проходящий по каждому ребру ровно один раз.
Если у дерева каждое ребро заменить на два ориентированных, мы получим эйлеров орграф.
Чтобы построить эйлеров обход дерева, пишем `dfs(v)`.

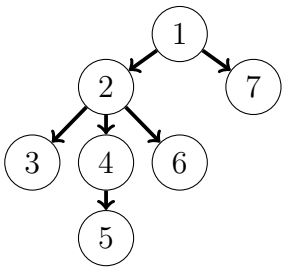
```
1 void dfs(int v) {
2     for (Edge e : graph[v]) { // пусть мы храним только рёбра, ориентированные вниз‘
3         answer.push_back(edge(v,x));
4         dfs(x);
5         answer.push_back(edge(x,v));
6     }
7 }
```

Такой обход назовём обычным или "*эйлеровым обхода дерева первого типа*".

Иногда имеет смысл хранить другую информацию после обхода:

```
1 void dfs2(int v) {
2     index[v] = answer.size(); // сохранили для вершины v любое её вхождение в answer‘
3     answer.push_back(v);
4     for (Edge e : graph[v]) {
5         dfs2(x);
6         answer.push_back(v); // поднимаемся, проходим через вершину v
7     }
8 }
9 void dfs3(int v) {
10    L[v] = answer.size(); // аналог времени входа
```

```
11      answer.push_back(v); // просто сохраняем порядок обхода вершин dfs-ом
12      for (Edge e : graph[v])
13          dfs3(x);
14      R[v] = answer.size(); // аналог времени выхода
15  }
```



Эйлеров обход 1-го типа (обычный)	(1,2) (2,3) (3,2) (2,4) (4,5) (5,4) (4,2) (2,6) (6,2) (2,1) (1,7) (7,1)
Эйлеров обход второго типа (±1)	Обход: 1 2 3 2 4 5 4 2 6 2 1 7 1 Высоты: 0 1 2 1 2 3 2 1 2 1 0 1 0
Обход третьего типа	Обход: 1 2 3 4 5 6 7

Второй обход по сути тоже эйлеров.
Мы сохраняем не рёбра, по которым проходим, а вершины, в которых оказываемся при обходе.
Третий обход уже слабо напоминает эйлеров, но мы в контексте задач, где нужно выбирать между 2-м и 3-м обходами, будем иногда называть его эйлеровым.
Зачем нужны 2-й и 3-й обходы будет понятно уже сейчас, 1-й нам пригодится для ETT.

Lm 29.5.1. После третьего обхода отрезок обхода $[L_v, R_v)$ задаёт ровно поддерево вершины v.

Следствие 29.5.2. Пусть у каждой вершины дерева есть вес w_v. Тогда мы теперь умеем за $\mathcal{O}(\log n)$ делать все операции на поддереве, которые ДО умело делать на отрезке.

Lm 29.5.3. Массив высот `h[i] = height[answer[i]]` второго обхода обладает ±1 свойством.

Lm 29.5.4. `LCA(a,b) = answer[h.RMQ[index[a],index[b]]]`

Доказательство. dfs по пути из a в b пройдёт через LCA. Это будет вершина минимальной высоты на пути, так как, чтобы попасть в ещё меньшие, dfs должен сперва выйти из LCA. ■

Замечание 29.5.5. Итого мы получили решение задачи LCA за $\langle n, 1\rangle$. Этот относительно свежий результат был получен в 2000-м Фарах-Колтоном и Бендером (два человека). Ссылка на статью.

Замечание 29.5.6. На практике популярен способ решения LCA: LCA → RMQ, а RMQ решим через Sparse Table. Это $\langle n \log n, 1\rangle$, причём у $\mathcal{O}(1)$ относительно небольшая константа.

29.6. RMQ → LCA

Чтобы свести задачу "RMQ на массиве a" к LCA, построим декартово дерево на парах (i, a_i).
Пары уже отсортированы по x, поэтому построение – проход со стеком за $\mathcal{O}(n)$.

Lm 29.6.1. RMQ на $[l, r]$ в исходном массиве равно $LCA(l, r)$ в полученном дереве.

Доказательство. Каждой вершине декартова дерева соответствует отрезок исходного массива, корнем поддерева выбирается минимум по $y_i = a_i$ на этом отрезке.
Будем спускаться от корня дерева, пока не встретим вершину, которая разделяет l и r.
Ключ, записанный в найденной вершине, обозначим i, отрезок вершины $[L_i, R_i] \Rightarrow$
$a_i = \min_{L_i \leqslant j \leqslant R_i} a_j$ и $L_i \leqslant l \leqslant i \leqslant r \leqslant R_i \Rightarrow a_i \geqslant \min_{l \leqslant j \leqslant r} a_j$ и $i \in [l, r]$.
Осталось заметить, что $i = \text{LCA}(l, r)$. ■

Следствие 29.6.2. Мы научились решать статичную версию RMQ за $\langle n, 1\rangle$. Победа!

29.7. LCA в offline, алгоритм Тарьяна

Для каждой вершины построим список запросов с ней связанных. Пусть i-й запрос – (a_i, b_i).

```
q[a[i]].push_back(i), q[b[i]].push_back(i)
```

Будем обходить дерево dfs-ом, перебирать запросы, связанные с вершиной, и отвечать на все запросы, второй конец которых серый или чёрный.

```
void dfs( int v) {
    color[v] = GREY;
    for (int i : q[v]) {
        int u = a[i] + b[i] - v;
        if (color[u] != WHITE)
            answer[i] = DSU.get(u);
    }
    for (int x : graph[v])
        dfs(x), DSU.parent[x] = v;
    color[v] = BLACK;
}
```

Серые вершины образуют путь от v до корня. У каждой серой вершины есть чёрная часть поддерева, это и есть её множество в DSU. LCA(v, u) – всегда серая вершина, то есть, нужно от u подниматься вверх до ближайшей серой вершины, что мы и делаем.

В коде для краткости используется DSU со сжатием путей, но без ранговой эвристики, поэтому время работы будет $\mathcal{O}((m+n)\log n)$. Если применить обе эвристики, получится $\mathcal{O}((m+n)\alpha)$, но нужно будет поддерживать в корне множества "самую высокую вершину множества".

29.8. LA (level ancestor)

Запрос `LA(v,k)` – подняться в дереве от вершины v на k шагов вверх.
Мы уже умеем решать эту задачу за $\langle n\log n, \log n\rangle$ двоичными подъёмами.
В offline на m запросов можно ответить dfs-ом за $\mathcal{O}(n+m)$: когда dfs зашёл в вершину v, у нас в стеке хранится весь путь до корня, и к любому элементу пути мы можем обратиться за $\mathcal{O}(1)$.

• Алгоритм Вишкина

Как и при сведении LCA $\to$ RMQ±1, выпишем высоты Эйлерова обхода второго типа.
`LA(v,k) = getNext(index[v],height[v]-k)`, где `index` – позиция в Эйлеровом обходе, а `getNext(i,x)` возвращает ближайший справа от i элемент $\leqslant x$.
Мы умеем отвечать на `getNext(i,x)` за $\langle n, \log n\rangle$ одномерным ДО снизу или сверху.

29.9. Euler-Tour-Tree

Задача: придумать структуру данных для хранения графа, поддерживающую операции

- `link(a, b)` – добавить ребро между a и b.
- `cut(a, b)` – удалить ребро между a и b.
- `isConnected(a, b)` – проверить связность двух вершин.

При этом в каждый момент времени выполняется условие отсутствия циклов (граф – лес).
По сути мы решаем *Dynamic Connectivity Problem* с дополнительным условием "граф – лес".

Решение: хранить обычный эйлеров обход дерева (ориентированные рёбра).

Эйлеров обход – массив. Заведём заведём на нём rope. Например, treap по неявному ключу.

```
map<pair<int,int>, Node*> node; // по орребру умеем получать вершину treap
vector<Node*> anyEdge; // для каждой вершины графа храним любое исходящее ребро
bool isConnected(int a, int b) {
    // взяли у каждой вершины произвольное ребро, проверили, что два ребра живут в одном дереве
    return getRoot(anyEdge[a]) == getRoot(anyEdge[b]);
}
void cut(int a, int b) {
    // по орребру получаем Node*, находим его позицию в эйлеровом обходе
    Node *node1 = node[make_pair(a, b)];
    Node *node2 = node[make_pair(b, a)];
    int i = getPosition(node1), j = getPosition(node2);
    if (i > j) swap(i, j); // упорядочили (i,j)
    Node *root = getRoot(node1), *a, *b, *c;
    Split(root, a, b, i);
    Split(b, b, c, j - i); // разделили дерево на три части: (a) (i b) (j c)
    Delete(b, 0), Delete(c, 0); // собственно удаление лишнего ребра
    Merge(a, c); // в итоге теперь есть два дерева: (a c) и (b)
}
```

С операцией `link(a,b)` чуть сложнее – нужно сделать циклические сдвиги обходов: если обходы называются X и Y, а добавляемые рёбра e_1 и e_2, мы хотим представить ответ, как Xe_1Ye_2.

```
void link(int a, int b) {
    Node *pa = anyEdge[a], *pb = anyEdge[b];
    Rotate(getRoot(pa), getPosition(pa)); // Rotate = Split + Merge
    Rotate(getRoot(pb), getPosition(pb));
    // теперь первые ребра эйлеровых обходов - исходящие из a и b соответственно
    Node *e1 = createEdge(a, b), *e2 = createEdge(b, a);
    Merge(Merge(getRoot(pa), e1), Merge(getRoot(pb), e2));
}
```

Можно реализовать `pair<Node*,int> goUp(Node*)`, которая заменит и `getRoot`, и `getPosition`.

29.10. (*) LA, быстрые решения

29.10.1. (*) Вишкин за $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$

В Section 29.8 у нас уже построен Эйлеров обход и мы знаем, что
`LA(v,k) = getNext(index[v],height[v]-k) = getNext(i,a[i]-k)`.
Осталось научиться вычислять `getNext` за $\mathcal{O}(1)$.

Давайте $\forall i$ предподсчитаем `getNext(i,a[i]-j)` для всех j до $3 \cdot 2^k$, где 2^k – максимальная степень двойки, которая делит i. Суммарный размер предподсчёта $= \sum_k \frac{2n}{2^k}(3 \cdot 2^k) = \Theta(n \log n)$, где $2n$ – длина Эйлерова обхода. Предподсчёт делается через LA-offline за то же время.

Ответ на запрос: выберем s: $2^s \leqslant k < 2^{s+1}$.
Поскольку `getNext(i,a[i]-k)` $\geqslant i + k \geqslant i + 2^s$, перейдём к $j = i - (i \bmod 2^s) + 2^s$.
Заметим, что `getNext(j,a[i]-k)` предподсчитан, так как $a[j] - a[i] \leqslant 2^s \wedge k \leqslant 2 \cdot 2^s$.

29.10.2. (*) Ladder decomposition + четыре русских

• **Longest-Path-Decomposition и** $\langle \mathcal{O}(n), \mathcal{O}(\sqrt{n}) \rangle$

Longest-Path-Decomposition – разбиение вершин дерева на пути, путь от вершины продолжается вниз в сторону самого глубокого сына. Внутри одного пути LA считается за $\mathcal{O}(1)$. Иначе будем прыгать вверх по путям. Прыжков $\mathcal{O}(\sqrt{n})$, так как, на k-м подъёме у нас должен быть путь длины $\geqslant k \Rightarrow$ всего в дереве $1 + \cdots + k = \Theta(k^2)$ вершин.

• **Ladder-Decomposition и** $\langle \mathcal{O}(n), \mathcal{O}(\log n) \rangle$

Возьмём пути из прошлого решения.
Пусть k_i – длина пути, насчитаем продолжение пути вверх на k_i.
Размер и время предподсчёта всё ещё линейны.
Прыгаем вверх. Пусть мы сейчас в вершине v и прошли снизу расстояние $x \Rightarrow$ из v есть путь вниз длины $\geqslant x \Rightarrow$ вверх тоже $\geqslant x \Rightarrow$ каждый раз расстояние удваивается.

• **Ladder-Decomposition и** $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$

Добавим двоичных подъёмов. Отсюда и предподсчёт $\mathcal{O}(n \log n)$. С их помощью, желая прыгнуть на k, за $\mathcal{O}(1)$ прыгнем на 2^s: $2^s \leqslant k < 2^{s+1}$. Из вершины, в которой мы оказались, путь ladder decomposition позволяет подняться за $\mathcal{O}(1)$ на оставшиеся $k - 2^s \leqslant 2^s$.

• **Эйлеров обход, четыре русских и** $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$

Нужно насчитать двоичные подъёмы не ото всех вершин, а лишь от $\Theta(\frac{n}{\log n})$ каких-то...
Предподсчитывать динамикой $f[i, v] = f[i-1, f[i-1, v]]$ не получится $\Rightarrow$ используем LA-offline.

Алгоритм:
Берём ± 1 эйлеров обход, выбираем m и от каждой m-й вершины считаем подъёмы.
Чтобы ответить на запрос $LA(v, k)$ берём $i = pos[v]$ в эйлеровом обходе, $j = i - (i \bmod m) + m$.
Если $h[j] \leqslant h[i] + k$, то отвечаем за $\mathcal{O}(1)$ от j, иначе ответ лежит на $(j - m, j]$.

Ответ зависит только от битовой строки длины m, чисел $j - i \leqslant m$ и $k \leqslant m$.
Итого $2^m m^2$ различных задач, ответы на которые мы предподсчитаем. Пусть $m = \frac{1}{2} \log n$.

Лекция #30: HLD & LCT

30-я пара, 2024/25

30.1. Heavy Light Decomposition

Задача. Дано дерево с весами в вершинах. Нужно считать функцию на путях дерева, и поддерживать изменение весов вершин.

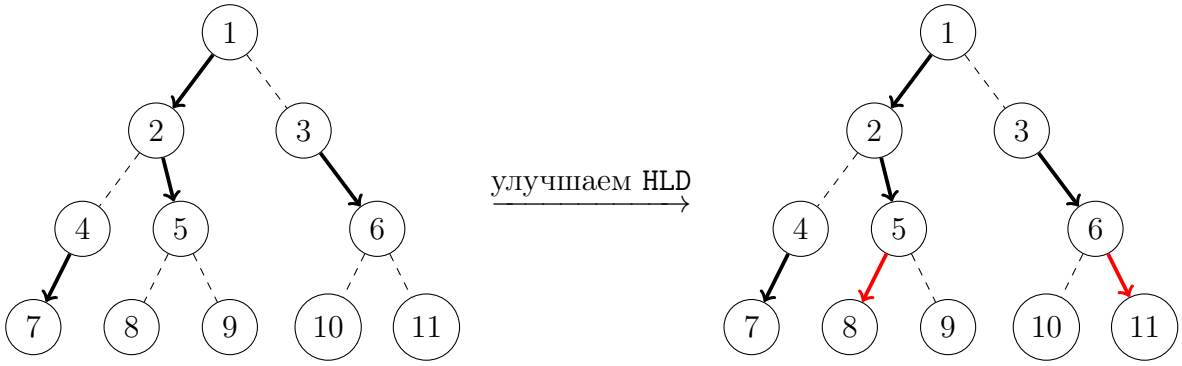
В частном случае "дерево = путь" задача уже решена деревом отрезков. Обобщим решение для произвольного дерева: разобьём вершины дерева на пути, на каждом построим дерево отрезков. Теперь любой путь разбивается на какое-то количество отрезков, на которых функцию можно посчитать деревом отрезков. Осталось выбрать такое разбиение на пути, чтобы количество отрезков всегда было небольшим. Подвесим дерево. Теперь каждое ребро или лёгкое, или тяжёлое, причём у каждой вершины не более одного тяжёлого сына.

Def 30.1.1. *Heavy-Light декомпозиция дерева (`HLD`) – пути образованные тяжёлыми рёбрами.*

Напомним, смысл в том, чтобы путь разбивался на как можно меньшее число отрезков. Поэтому если в пути можно включить больше рёбер, полезно это сделать. В частности каждую вершину выгодно соединить хотя бы с одним сыном.

Def 30.1.2. *Улучшенная `HLD`: для каждой вершины выбрали ребро в самого тяжёлого сына.*

Улучшенная `HLD` включает все те же рёбра, что и обычная, и ещё некоторые.



Построить разбиение на пути можно за линейное время.
Деревья отрезков на путях также строятся за линейное время.

```
void build_HLD(int v) { // внутри пути нумерация снизу вверх
    int ma = -1;
    size[v] = 1;
    for (int x : children[v]) {
        build_HLD(x);
        size[v] += size[x];
        if (ma == -1 || size[x] > size[ma])
            ma = x;
    }
    path[v] = (ma == -1 ? path_n++ : path[ma]); // номер пути, на котором лежит v
    pos[v] = len[path[v]]++; // позиция v внутри пути
    top[path[v]] = v; // для каждого пути помним верхнюю вершину
}
build_HLD(root); // за O(n) для каждой v нашли path[v], pos[v], для каждого p len[p], top[p]
```

Lm 30.1.3. В `HLD` на пути от любой вершины до корня не более $\log n$ прыжков между путями.

Доказательство. Все прыжки между путями – лёгкие рёбра. ■

• Подсчёт функции на пути.

Можно найти `LCA` и явно разбить путь $a \to b$ на два вертикальных: $a \to$ `LCA` $\to b$. Подсчёт функции на пути $a \to$ `LCA`: поднимаемся из a, если `path[a]` $=$ `path[LCA]`, можем посчитать функцию за одно обращение к дереву отрезков: `tree[path[a]].get(pos[a], pos[LCA])`.
Иначе поднимаемся до `top[path[a]]` и переходим в `parent[top[path[a]]]`.

Теорема 30.1.4. Время подсчёта функции на пути $\mathcal{O}(\log^2 n)$.

Доказательство. Не более $2 \log n$ обращений к дереву отрезков. ■

Теорема 30.1.5. Время изменения веса одной вершины $\mathcal{O}(\log n)$.

Доказательство. Вершина лежит ровно в одном дереве отрезке. ■

Замечание 30.1.6. Аналогично можно считать функции на рёбрах.
Например, используем биекцию "вершина $\leftrightarrow$ ребро в предка".

Замечание 30.1.7. Воспользуемся функцией `isAncestor`, тогда можно обойтись без подсчёта `LCA`: прыгать от вершины a вверх, пока не попадём в предка b, и затем от b вверх, пока не попадём в предка a. В конце a и b лежат на одном пути, учтём эту часть тоже. Более того, таким образом с помощью `HLD` можно найти `LCA` за время $\mathcal{O}(\log n)$, используя всего $\mathcal{O}(n)$ предподсчёта.

• Другие применения.

`HLD` позволяет выполнять любые запросы на пути дерева, которые умело выполнять на отрезке дерево отрезков. Например, перекрасить путь в некоторый цвет. Или делать `+=` на пути дерева и при этом поддерживать минимум на пути в дереве.

С помощью `HLD` можно считать даже гораздо более сложные вещи: например, в дереве с весами на рёбрах поддерживать изменение веса ребра и длину диаметра дерева.

• Функция на поддереве.

Сгенерируем пути чуть иначе.

```
int bound = 0; // левая граница текущего пути
int k = 0, a[n]; // Эйлеров обход дерева
void build_HLD( int v ) {
    left[v] = bound, pos[v] = k, a[k++] = v;
    if (children[v].empty())
        return;
    int i = (x in children[v] : size[x] = max);
    buildPos(i, v); // сперва идём в тяжёлого сына, продолжаем текущий путь
    for (int x : c[v])
        if (x != i)
            bound = k, build_HLD(x); // сдвинуть границу - начать новый путь
}
```

`pos[v]`, `left[v]` – позиции вершины и верхушки её пути в Эйлеровом обходе.
Получается, что каждый путь HLD – отрезок Эйлерова обхода `a[]`. Из этого два вывода:
- Можно параллельно применять массовые операции и на путях, и на поддеревьях.
- Можно хранить одно дерево отрезков на массиве `a[]`, содержащее сразу все пути.

30.2. Link Cut Tree

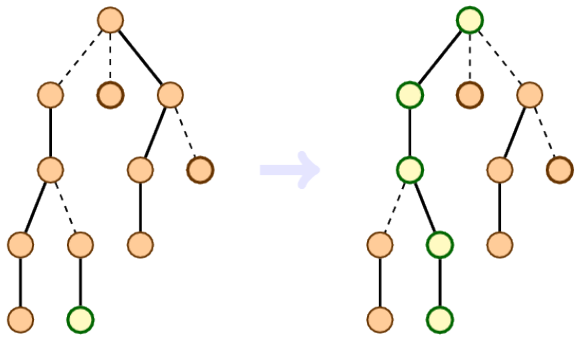
Мы уже очень много всего умеем делать с деревьями.
С помощью `HLD` умеем обрабатывать запросы `getMax(a,b)` и `changeWeight(v)`.
С помощью `ETT` умеем обрабатывать запросы `link(a,b)`, `cut(a,b)`, `isConnected(a,b)`.

Link-Cut-Trees структура данных, которая умеет всё вышеперечисленное и не только.
Основная идея = динамически меняющаяся декомпозиция дерева на пути + для каждого пути сохранить rope (например, treap по неявному ключу).
Изначально каждая вершина – отдельный путь.

• `GetRoot(v)`, `GetPos(v)`

В каждый момент любая вершина v лежит ровно в одном пути, а путь этот хранится в treap, который состоит из `Node*`. Давайте для v хранить ссылку на её `Node*` в treap её пути: `p[v]`. Пусть в treap есть ссылки на отцов $\Rightarrow$ от `p[v]` можно за $\mathcal{O}(\log n)$ подняться до корня её treap и параллельно насчитать "число вершин в treap левее `p[v]`", то есть, позицию v в её пути.
Если поступает любой из запросов {`getMax(a,b)`, `link(a,b)`, `cut(a,b)`, `isConnected(a,b)`}, сделаем сначала `Expose(a)`.

Def 30.2.1. *`Expose(v)` – операция, меняющая декомпозиция дерева на пути.*
Она все вершины на пути от v до корня объединяет в один большой путь.



Время работы `Expose(v)` равно $\mathcal{O}(k)$ вызовов `split` и `merge`, где k – число прыжков между путями при подъёме от v до корня. Мы докажем, что амортизированно $k \leqslant \log n$.

• `MakeRoot(v)`

Rope умеет делать `reverse`, для этого достаточно в каждой вершине treap хранить отложенную операцию `isReversed`. Если после `Expose(v)` отрезать часть пути под v и сделать "`GetRoot(p[v]).isReversed ^= 1`", вершина v станет новым корнем дерева.
Оценим время работы новой операции: $T(\texttt{MakeRoot(v)}) = T(\texttt{Expose(v)}) + \mathcal{O}(\log n)$.

• `getMax(a,b)`, `link(a,b)`, `cut(a,b)`, `isConnected(a,b)`

Красота происходящего в том, что все операции теперь коротко выражаются:

```
int getMax(int a, int b) {
    MakeRoot(a), MakeRoot(b);
    return GetRoot(a)->max;
}
void link(int a, int b) {
    MakeRoot(a), MakeRoot(b);
    parent[a] = b;
}
void cut(int a, int b) {
    MakeRoot(a), MakeRoot(b);
    split(GetRoot(b), 1); // путь состоит из двух вершин - a и b, разрежем на два
    parent[a] = -1; // a - нижняя из двух вершин
}
bool isConnected(int a, int b) {
    MakeRoot(a), MakeRoot(b); // a и b были в одной компоненте ⇒ теперь они в одном пути
    return GetRoot(a) == GetRoot(b);
}
```

Красота функции `getMax` в том, что после двух `MakeRoot` весь путь $a \rightsquigarrow b$ – ровно один treap в "покрытии путями". И максимум на пути хранится в корне соответствующего treap.

Теорема 30.2.2. Суммарное время m операций `Expose`/`MakeRoot` – $\mathcal{O}(n + m \log n)$.

Доказательство. Потенциал $\varphi =$ минус "число тяжёлых рёбер, покрытых путями".
$\varphi_0 = 0, \varphi \geqslant -n \Rightarrow \sum t_i = \sum a_i + (\varphi_0 - \varphi_m) \leqslant n + \sum a_i$. Осталось оценить a_i.

Число лёгких рёбер на пути будем обозначать "Л", число тяжёлых "Т".
`Expose`: $a_i = t_i + \Delta\varphi \leqslant t_i - \text{Т} + \text{Л} \leqslant k - (k - \log n) + \log n = \mathcal{O}(\log n)$.
`MakeRoot = Expose + Reverse`. `Reverse` меняет тяжёлость только рёбрам на пути $(v, root)$. И до reverse, и после лёгких на пути не более $\log n \Rightarrow \Delta\varphi \leqslant \log n \Rightarrow a_i = t_i + \Delta\varphi = \mathcal{O}(\log n)$. ■

Замечание 30.2.3. На практике мы также оценим собственно `Link`/`Cut` и добавим splay-дерево.

30.3. MST за $\mathcal{O}(n)$

Алгоритм построения MST от графа из n вершин и m рёбер обозначим $F(n, m)$. Алгоритм F:

1. Сделаем 3 шага алгоритма Борувки: $n \to \frac{n}{8}$. Время работы $\mathcal{O}(n+m)$.
2. Берём случайное множество A из $\frac{m}{2}$ рёбер.
   Построим MST от A рекурсивным вызовом $F(\frac{n}{8}, \frac{m}{2})$.
   Множество рёбер полученного MST обозначим T. Рёбра из $A \setminus T$ точно не входят в MST(E).
3. Переберём рёбра из $B = E \setminus A$, оставим из них $U \subseteq B$ – те рёбра, что могут улучшить T. Ребро (a, b) может улучшить T, если a и b не связны в T или максимум на пути в T между a и b больше веса ребра. Рёбра из $B \setminus U$ точно не входят в MST(E).
4. MST$(E) \subseteq T \cup U$. Сделаем рекурсивный вызов от $F(\frac{n}{8}, |T| + |U|)$.

Оценим общее время работы: $|T| \leqslant \frac{n}{8} - 1$. Скоро мы покажем, что матожидание $|U| \leqslant \frac{n}{8} - 1$. Суммарное время работы $F(n, m) \leqslant (n+m) + M(\frac{n}{8}, \frac{m}{2}) + F(\frac{n}{8}, \frac{m}{2}) + F(\frac{n}{8}, \frac{n}{4})$, где $M(n, m)$ – время поиска в offline минимумов на m путях дерева из n вершин. Оказывается $M(n, m) \leqslant n+m$. Сумма параметров в рекурсивных вызовах равна $\frac{n}{2}+\frac{m}{2} \Rightarrow$ вся эта прелесть работает за $\mathcal{O}(n+m)$.

Осталось всё последовательно доказать.

Lm 30.3.1. $A_1 = A \setminus T$ и $B_1 = B \setminus U$ точно не лежат в MST(E)

Доказательство. Представим себя Краскалом.
Встретив ребро $e_a \in A_1$, мы уже имеем путь между концами e_a (рёбра из T) $\Rightarrow e_a$ не добавим.
Встретив ребро $e_b \in B_1$ у нас уже есть путь между концами e_b (рёбра из T) $\Rightarrow e_b$ не добавим. ■

Как быстро посчитать минимумы на путях в дереве можно прочитать в работе Тарьяна. У нас на экзамене этого не будет. Лучшее, что мы сейчас умеем – $\mathcal{O}(m + n \log n)$: с помощью LCA разбили пути на вертикальные, перебираем верхние границы путей снизу вверх, считаем минимум за линию, но со сжатием путей. Собственно работа Тарьяна показывает, как в этом подходе использовать не только сжатие путей, но полноценное СНМ.

• Random Sampling Lemma.

Lm 30.3.2. Пусть p – вероятность включения ребра из E в A на втором шаге алгоритма F (в описанном алгоритме $p = \frac{1}{2}$) $\Rightarrow$ матожидание размера множества U не больше, чем $(\frac{1}{p}-1)(n-1)$.

Доказательство. Представим себя Краскалом, строящим MST от E. Основная идея доказательства: подбрасывать монетку, решающую попадёт ребро в A или в B не заранее, а прямо по ходу Краскала, когда встречаем "интересное" ребро. Упорядочим ребра по весу, перебираем их. Если Краскал считает, что очередное ребро надо добавить в остов, то с вероятностью p добавляем (событие X), а с $1-p$ пропускаем ребро (событие Y). Оценим матожидание числа просмотренных рёбер перед первым событием типа X, включая само X. $E = 1+(1-p)E \Rightarrow E = \frac{1}{p}$. Из этих $\frac{1}{p}$ рёбер первые $\frac{1}{p}-1$ идут в U, а последнее идёт в T. Поскольку в остов можно добавить максимум $n-1$ рёбер, событие X произойдёт не более $n-1$ раз, перед каждым X случится в среднем $\frac{1}{p} - 1$ событий типа Y. Итого $E\big[|U|\big] \leqslant (n-1)(\frac{1}{p} - 1)$. ■

30.4. (*) RMQ Offline

У нас есть алгоритм Тарьяна поиска LCA в Offline (dfs по дереву + CHM).
Мы умеем сводить RMQ-offline к LCA-offline построением декартова дерева.
Если две эти идеи объединить в одно целое, получится удивительно простой алгоритм:

```
void solve(int m, Query *q, int n, int *a) {
    vector<int> ids(n); // для каждой правой границы храним номера запросов
    for (int i = 0; i < m; i++)
        ids[q[i].right].push_back(i);
    DSU dsu(n); // инициализация CHM: каждый элемент - самостоятельное множество
    stack<int> mins;
    for (int r = 0; r < n; r++) {
        while (mins.size() && a[mins.top()] >= a[r])
            dsu.parent[mins.pop()] = r; // минимум достигается в r, объединим отрезки
        mins.push(r);
        for (int i : ids[r])
            q[i].result = dsu.get(q[i].left);
    }
}
```

В описанном выше алгоритме можно рассмотреть построение декартова дерево и параллельно обход полученного дерева dfs-ом. Проще воспринимать его иначе.
Когда мы отвечаем на запросы $[l, r]$ при фиксированном r, ответ зависит только от l.
Если $l \in [1, m_1]$, ответ – $a[m_1]$, если $l \in (m_1, m_2]$, ответ – $a[m_2]$ и т.д.
Здесь m_1 – позиция минимума на $[1, r]$, а m_{i+1} – позиция минимума на $(m_i, r]$.
Отрезки $(m_i, m_{i+1}]$ мы будем поддерживать, как множества в DSU.
Минимумы лежат в стеке: $a[m_1] \leqslant a[m_2] \leqslant a[m_3] \leqslant \dots$
Когда мы расширяем префикс: $r \to r{+}1$, стек минимумом обновляется, отрезки объединяются.

Лекция #31: Игры на графах

31-я пара, 2024/25

Тема подробно раскрыта в обзоре А. С. Станкевича «Игры на графах». В частности, там описана невошедшая в лекцию теория Смита для суммы игр с циклами.

31.1. Основные определения

Def 31.1.1. *Игра на орграфе: по вершинам графа перемещается фишка, за ход игрок должен сдвинуть фишку по одному из рёбер.*
Симметричная игра – *оба игрока могут ходить по всем рёбрам*
Несимметричная игра – *каждому игроку задано собственное множество рёбер*
Проигрывает игрок, который не может ходить.

Упражнение 31.1.2. Пусть по условию игры "*некоторые вершины являются выигрышными или проигрышными для некоторых игроков*". Такую игру можно вложить в определение выше.

Замечание 31.1.3. Чтобы не придумывать ничего отдельно для несимметричных игр, обычно просто вводят новый граф, вершины которого – пары $\langle v, who \rangle$ (вершина и, кто ходит).

Результат симметричной игры определяется графом G и начальной вершиной $v \in G$.
Игру будем обозначать (G, v), результат игры $r(G, v)$, или для краткости $r(v)$.
Классифицируем $v \in G$: в зависимости от $r(G, v)$ назовём вершину v
проигрышной (`L`), *выигрышной* (`W`) или *ничейной* (`D`).
Также введём множества: $\texttt{WIN} = \{v \mid r(v) = \texttt{W}\}$, $\texttt{LOSE} = \{v \mid r(v) = \texttt{L}\}$, $\texttt{DRAW} = \{v \mid r(v) = \texttt{D}\}$.

Замечание 31.1.4. В несимметричных играх, если вершина, например, проигрышна, ещё важно добавлять, какой игрок ходил первым: "проигрышная для 1-го игрока".

Lm 31.1.5. Если для вершины в условии задачи не указан явно её тип, то:

$$W \Leftrightarrow \exists \text{ ход в } L, \quad L \Leftrightarrow \text{ все ходы в } W$$

31.1.1. Решение для ациклического орграфа

Граф ацикличный $\Rightarrow$ вспомним про динамическое программирование.
`dp[v]` $= r(G, v)$; изначально "`-1`", т.е. "не посчитано".
База: пометим все вершины, информация о которых дана по условию.
Далее ленивая динамика:

```
int result( int v ) {
    int &r = dp[v];
    if (r != -1) return r;
    r = L; // например, если исходящих рёбер нет, результат уже верен
    for (int x : edges[v])
        if (result(x) == L) // ищем ребро в проигрышную
            r = W; // добавь break, будь оптимальнее!
    return r;
}
```

31.1.2. Решение для графа с циклами (ретроанализ)

Будем пользоваться Lm 31.1.5. Цель: как только есть вершина, которая по лемме должна стать W/L, делаем её такой и делаем из этого некие выводы. Процесс можно реализовать через dfs/bfs. Мы выберем именно bfs, чтобы в будущем вычислить *длину игры*. Итак, ретроанализ:

```
queue <-- все вершины, которые по условию W/L.
while !queue.empty()
    v = queue.pop()
    for x in inner_edges[v]: // входящие в v рёбра
        if lose[v]:
            make_win(x, d[v]+1)
        else:
            if ++count[x] == deg[x]: // в count считается число проигрышных рёбер
                make_lose(x, d[v]+1)
```

Функции `make_win` и `make_lose` проверяют, что вершину помечают первый раз, если так, добавляют её в очередь. Второй параметр – обычное для `bfs` расстояние до вершины.
Все помеченные вершины по Lm 31.1.5 помечены правильно.
Непомеченные вершины, чтобы им было не обидно, пометим D.

Теорема 31.1.6. Ничейные – ровно вершины с пометкой D.

Доказательство. Из вершины v типа D есть рёбра только в D и W (в L нет, т.к. тогда бы наш алгоритм пометил v, как W). Также из любой D есть хотя бы одно ребро в D. Мы игрок, мы находимся в D. Каков у нас выбор? Если пойдём в W, проиграем. Проигрывать не хотим $\Rightarrow$ пойдём в D $\Rightarrow$ вечно будем оставаться в D $\Rightarrow$ ничья. ■

Def 31.1.7. *$len(G, v)$ – длина игры, сколько ходов продлится игра, если выигрывающий игрок хочет выиграть максимально быстро, а проигрывающий максимально затянуть игру.*

Замечание 31.1.8. После ретроанализа `d[v]` $= len(G, v)$, так как ретроанализ:
1. Перебирал вершины в порядке возрастания расстояния.
2. Для выигрышной вершины брал наименьшую проигрышную.
3. Для проигрышной вершины брал наибольшую выигрышную.

Строго доказать можно по индукции. Инварианты: при обработке v все вершины со строго меньшей длиной игры уже обработаны; если посчитано $r(v)$, то посчитано верно.

Замечание 31.1.9. На практике разбиралась садистская версия той же задачи: проигрывающий хочет побыстрее выиграть и начать новую партию, а выигрывающий подольше наслаждаться превосходством (т.е. оставлять себе возможность выиграть). Описание решения: после первого ретроанализа поменять местами смысл L/W, запустить второй ретроанализ.

31.2. Ним и Гранди, прямая сумма

Def 31.2.1. *Прямая сумма графов $G_1 = \langle V_1, E_1\rangle$ и $G_2 = \langle V_2, E_2\rangle$ – граф с вершинами $\langle v_1, v_2\rangle \,|\, v_1 \in V_1, v_2 \in V_2$ и рёбрами $(a, b) \to (a, c) \,|\, (b, c) \in E_2$ и $(a, b) \to (c, b) \,|\, (a, c) \in E_1$.*

Def 31.2.2. *Прямая сумма игр:* $\langle G_1, v_1\rangle \times \langle G_2, v_2\rangle = \langle G_1 \times G_2, (v_1, v_2)\rangle$

По сути “у нас есть два графа, можно делать ход в любом одном из них”.

Def 31.2.3. $mex(A) = \min x \mid x \geqslant 0, x \notin A$.

Пример 31.2.4. $A = \{0,1,7,10\} \Rightarrow mex(A) = 2; \quad A = \{1,2,3,4\} \Rightarrow mex(A) = 0$

Def 31.2.5. *На ациклическом орграфе можно задать Функцию Гранди* $f[v]$*:*
Пусть из v *ведут рёбра в* $Out(v) = \{x_1, x_2, \dots, x_k\} \overset{def}{\Rightarrow} f[v] = mex\{f[x_1], \cdots, f[x_k]\}$.

Lm 31.2.6. $f[v] = 0 \Leftrightarrow v \in$ `LOSE` (доказывается элементарной индукцией)

Пример 31.2.7. *Игра "Ним".* На столе n камней, за ход можно брать любое положительное число камней. Заметим $f[n] = n$, выигрышная стратегия – взять всё.

Пока теория не выглядит полезной. Чтобы осознать полезность, рассмотрим прямые суммы тех же игр – есть n кучек спичек или n кучек камней, и брать можно, соответственно, только из одной из кучек. На вопрос "кто выиграет в таком случае" отвечают следующие теоремы:

Теорема 31.2.8. $f[\langle v_1, v_2\rangle] = f[v_1] \oplus f[v_2]$

Доказательство. Для доказательства воспользуемся индукцией по размеру графа.
Из вершины v в процессе игры сможем прийти только в $C(v)$ – достижимые из v вершины.
Для любого ребра $\langle v_1, v_2\rangle \to \langle x_1, x_2\rangle$ верно, что $|C(\langle v_1, v_2\rangle)| > |C(\langle x_1, x_2\rangle)|$.
База индукции: или из v_1, или из v_2 нет рёбер.
Переход индукции: для всех $\langle G_1, x_1, G_2, x_2\rangle$ с $|C(\langle x_1, x_2\rangle)| < |C(\langle v_1, v_2\rangle)|$ уже доказали

$$f[\langle x_1, x_2\rangle] = f[x_1] \oplus f[x_2]$$

Осталось честно перечислить все рёбра из $\langle v_1, v_2\rangle$ и посчитать mex вершин, в которые они ведут.

$$A = \{\underbrace{f[v_1] \oplus f[x_{21}], f[v_1] \oplus f[x_{22}], \dots}_{\text{игрок сделал ход из } v_2}, \underbrace{f[x_{11}] \oplus f[v_2], f[x_{12}] \oplus f[v_2], \dots}_{\text{игрок сделал ход из } v_1}\}.$$

Здесь $Out(v_1) = \{x_{11}, x_{12}, \dots\}, Out(v_2) = \{x_{21}, x_{22}, \dots\}$. Доказываем, что $mex\, A = f[v_1] \oplus f[v_2]$.

Во-первых, поскольку $\forall i\, f[x_{1i}] \neq f[v_1] \wedge \forall i\, f[x_{2i}] \neq f[v_2]$, имеем $f[v_1] \oplus f[v_2] \notin A$.
Докажем, что все меньшие числа в A есть. Обозначим $x = f[v_1]$, $y = f[v_2]$, $M = f[v_1] \oplus f[v_2]$.
Будем пользоваться тем, что из v_1 есть ходы во все числа из $[0, x)$, аналогично $v_2 \to [0, y)$.
Пусть k – старший бит M и пришёл он из $x \Rightarrow$
ходами $x \to [0, 2^k)$ мы получим 2^k в x различных k-битных чисел, т.е. все числа из $[0, 2^k)$.
Чтобы получить числа $[2^k, M)$ перейдём от задачи $\langle x, y, M\rangle$ к $\langle x - 2^k, y, M - 2^k\rangle$.
Воспользуемся индукцией. База: $M = 0$. ■

Следствие 31.2.9. Для суммы большего числа игр аналогично: $f[v_1] \oplus f[v_2] \oplus \cdots \oplus f[v_k]$.

Замечание 31.2.10. Рассмотрим раскраску части плоскости $[0, +\infty) \times [0, +\infty)$.
В клетку $[i, j]$ ставится минимальное неотрицательное целое число, которого нет левее и ниже.
Получится ровно $i \oplus j$, так как мы ставим ровно mex в игре Ним на кучках $\{i, j\}$.

31.3. Вычисление функции Гранди

Ациклический граф ⇒ динамика. Осталось быстро научиться считать mex:
используем "обнуление" массива за $\mathcal{O}(1)$ и получим $\mathcal{O}(deg_v)$ на вычисление mex вершины v:

```
int cc, used[MAX_MEX]; // изначально нули

cc++;
for (int x : out_edges[v])
    used[f[x]] = cc; // функция Гранди x уже посчитана
for (f[v] = 0; used[f[v]] == cc; f[v]++)
    ;
```

Итого суммарное время работы $\mathcal{O}(V+E)$.
Нужно ещё оценить `MAX_MEX`. Тривиальная оценка даёт $\mathcal{O}(\max_v \deg_v)$, можно точнее:

Lm 31.3.1. $\forall G = \langle V, E\rangle,\ \forall v\ f[v] \leqslant \sqrt{2E}$

Доказательство. ДЗ ■

31.4. Эквивалентность игр

Напомним, игра на графе – пара $\langle G, v\rangle$. Материал главы относится к произвольным орграфам.

Def 31.4.1. *Игры A и B называются эквивалентными, если $\forall C\ r(A \times C) = r(B \times C)$.*

Def 31.4.2. *Игры A и B называются эквивалентными, если $A + B$ проигрышна.*

На лекции вам дано второе определение, обычно используют первое...
В любом случая важно понимать, что “эквивалентность” – отдельная глава, которой мы не пользовались, выводя фукнцию Гранди от прямой суммы игр.

TODO