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

Собрано 12 ноября 2024 г. в 20:59

Содержание

1. Асимптотика 1
1.1. О курсе. Хорошие алгоритмы. 1
1.2. Асимптотика, $\mathcal{O}$-обозначния 2
1.3. Рекуррентности и Карацуба 3
1.4. Мастер Теорема 5
1.5. (*) Экспоненциальные рекуррентные соотношения 5
1.6. (*) Доказательства по индукции 6
1.7. Числа Фибоначчи 6
1.8. (*) $\mathcal{O}$-обозначения через пределы 6
1.9. (*) Замена сумм на интегралы 7
1.10. Примеры по теме асимптотики 8
1.11. Сравнение асимптотик 9

2. Структуры данных 10
2.1. C++ 11
2.2. Неасимптотические оптимизации 13
2.3. Частичные суммы 14
2.4. Массив 14
2.5. Двусвязный список 14
2.6. Односвязный список 15
2.7. Список на массиве 15
2.8. Вектор (расширяющийся массив) 16
2.9. Стек, очередь, дек 16
2.10. Очередь, стек и дек с минимумом 17

3. Структуры данных 17
3.1. Амортизационный анализ 18
3.2. Разбор арифметических выражений 19
3.3. Бинпоиск 20
3.3.1. Обыкновенный 20
3.3.2. Lowerbound и Upperbound 20
3.3.3. Бинпоиск по предикату 21
3.3.4. Вещественный, корни многочлена 22
3.4. Два указателя и операции над множествами 22
3.5. Хеш-таблица 23
3.5.1. Хеш-таблица на списках 23
3.5.2. Хеш-таблица с открытой адресацией 23
3.5.3. Сравнение 24
3.5.4. C++ 25

4. Структуры данных **26**
4.1. Избавляемся от амортизации . . . 26
4.1.1. Вектор (решаем проблему, когда случится) . . . 26
4.1.2. Вектор (решаем проблему заранее) . . . 26
4.1.3. Сравнение способов . . . 26
4.1.4. Хеш-таблица . . . 27
4.1.5. Очередь с минимумом через два стека . . . 27
4.2. Бинарная куча . . . 29
4.2.1. GetMin, Add, ExtractMin . . . 29
4.2.2. Обратные ссылки и DecreaseKey . . . 30
4.2.3. Build, HeapSort . . . 30
4.3. Аллокация памяти . . . 31
4.3.1. Стек . . . 31
4.3.2. Список . . . 31
4.3.3. Куча (кратко) . . . 32
4.3.4. (*) Куча (подробно) . . . 32
4.3.5. (*) Дефрагментация . . . 33
4.4. Пополняемые структуры . . . 33
4.4.1. Ничего → Удаление . . . 33
4.4.2. Поиск → Удаление . . . 33
4.4.3. Add → Merge . . . 34
4.4.4. Build → Add . . . 34
4.4.5. Build → Add, Del . . . 35

5. Сортировки **36**
5.1. Два указателя и алгоритм Мо . . . 36
5.2. Квадратичные сортировки . . . 37
5.3. Оценка снизу на время сортировки . . . 38
5.4. Решение задачи по пройденным темам . . . 38
5.5. Быстрые сортировки . . . 38
5.5.1. CountSort (подсчётом) . . . 38
5.5.2. MergeSort (сортировка слиянием) . . . 38
5.5.3. QuickSort (реально быстрая) . . . 40
5.5.4. Сравнение сортировок . . . 40
5.6. (*) Adaptive Heap sort . . . 41
5.6.1. (*) Модифицированный HeapSort . . . 41
5.6.2. (*) Adaptive Heap Sort . . . 41
5.7. (*) Timsort . . . 41
5.8. (*) Ссылки . . . 41
5.9. (*) 3D Мо . . . 43
5.9.1. (*) Применяем для mex . . . 43

6. Сортировки (продолжение) **44**
6.1. Quick Sort . . . 44
6.1.1. Оценка времени работы . . . 44
6.1.2. Introsort'97 . . . 45
6.2. Порядковые статистики . . . 45

6.2.1. Одноветочный QuickSort . . . 45
6.2.2. Детерминированный алгоритм . . . 46
6.2.3. `C++` . . . 46
6.3. Integer sorting . . . 47
6.3.1. CountSort . . . 47
6.3.2. Radix sort . . . 47
6.3.3. Bucket sort . . . 48
6.4. Van Embde Boas'75 trees . . . 49
6.5. (*) Inplace merge за $\mathcal{O}(n)$ . . . 50
6.6. (*) Kirkpatrick'84 sort . . . 51

7. Кучи **51**
7.1. Нижняя оценка на построение бинарной кучи . . . 52
7.2. Min-Max Heap (Atkison'86) . . . 53
7.3. Leftist Heap (Clark'72) . . . 54
7.4. Skew Heap (Tarjan'86) . . . 54
7.5. Quake Heap (потрясная куча) . . . 55
7.5.1. Списко-куча . . . 55
7.5.2. Турнирное дерево . . . 56
7.5.3. Список турнирных деревьев . . . 56
7.5.4. DecreaseKey за $\mathcal{O}(1)$, quake! . . . 57
7.6. (*) Pairing Heap . . . 58
7.6.1. История. Ссылки. . . . 59
7.7. (*) Биномиальная куча (Vuillemin'78) . . . 60
7.7.1. Основные понятия . . . 60
7.7.2. Операции с биномиальной кучей . . . 61
7.7.3. Add и Merge за $\mathcal{O}(1)$ . . . 61
7.8. (*) Куча Фибоначчи (Fredman,Tarjan'84) . . . 61
7.8.1. Фибоначчиевы деревья . . . 63
7.8.2. Завершение доказательства . . . 63

8. Рекурсивный перебор **64**
8.1. Перебор перестановок . . . 64
8.2. Перебор множеств и запоминание . . . 65
8.3. Перебор путей (коммивояжёр) . . . 66
8.4. Разбиения на слагаемые . . . 67
8.5. Доминошки и изломанный профиль . . . 67

9. Динамическое программирование 1 **69**
9.1. Базовые понятия . . . 69
9.1.1. Условие задачи . . . 69
9.1.2. Динамика назад . . . 69
9.1.3. Динамика вперёд . . . 70
9.1.4. Ленивая динамика . . . 70
9.2. Ещё один пример . . . 71
9.3. Восстановление ответа . . . 71
9.4. Графовая интерпретация . . . 72

9.5. Checklist . . . 73
9.6. Рюкзак . . . 73
9.6.1. Формулировка задачи . . . 73
9.6.2. Решение динамикой . . . 73
9.6.3. Оптимизируем память . . . 73
9.6.4. Добавляем `bitset` . . . 74
9.6.5. Восстановление ответа с линейной памятью . . . 74
9.7. Квадратичные динамики . . . 75
9.8. Оптимизация памяти для НОП . . . 76
9.8.1. Храним биты . . . 76
9.8.2. Алгоритм Хиршберга (по wiki) . . . 76
9.8.3. Оценка времени работы Хиршберга . . . 76
9.8.4. (*) Алгоритм Хиршберга (улучшенный) . . . 77
9.8.5. Область применение идеи Хиршберга . . . 77

10. Формула включения-исключения **78**
10.1. (*) Разминочные задачи . . . 78
10.2. (*) Задачи с формулой Мёбиуса . . . 78

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. $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_kx^k$. По леммам Rem 1.2.12, Lm 1.2.18 имеем, что все слагаемые кроме a_kx^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}`.

```
1 for (i = 0; i < an; i++) // an = 5
2     for (j = 0; j < bn; j++) // bn = 4
3         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:

```
1 // Перемножаем числа без переносов, как многочлены
2 for (i = 0; i < an; i++) // an = 5
3     for (j = 0; j < bn; j++) // bn = 4
4         c[i + j] += a[i] * b[j];
5 // Делаем переносы, массив c = [45, 76, 94, 100, 70, 40, 19, 6, 0]
6 for (i = 0; i < an + bn; i++)
7     if (c[i] >= 10)
8         c[i + 1] += c[i] / 10, c[i] %= 10;
9 // 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 $T_2(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585\dots})$.
Данный алгоритм применим и для умножения многочленов, и для умножения чисел.

Псевдокод алгоритма Карацубы для умножения многочленов:

```
1 Mul(n, a, b): // n = 2^k, c(w) = a(w)*b(w)
2     if n == 1: return {a[0] * b[0]}
3     a --> a1, a2
4     b --> b1, b2
5     x = Mul(n / 2, a1, b1)
6     y = Mul(n / 2, a2, b2)
7     z = Mul(n / 2, a1 + a2, b1 + b2)
8     // Умножение на w^i - сдвиг массива на i вправо
9     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:
$T(n) = f(n) + aT(\frac{n}{b}) = f(n) + af(\frac{n}{b}) + a^2 f(\frac{n}{b^2}) + \dots = n^c + a(\frac{n}{b})^c + a^2(\frac{n}{b^2})^c + \dots$
Тогда $T(n) = f(n)(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \dots + \left(\frac{a}{b^c}\right)^k)$. При этом в сумме $k+1$ слагаемых.
Обозначим $q = \frac{a}{b^c}$ и оценим сумму $S(q) = 1 + q + \dots + q^k$.
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 множно 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

<u>Lm</u> 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. $T(n) = T(n-3) + T(n-3) \Rightarrow T(n) = 2T(n-3) = 4T(n-6) = \dots = 2^{n/3}$
3. $T(n) = T(n-1) + T(n-3)$. Применяем Thm 1.5.1, получаем $1 = \alpha^{-1} + \alpha^{-3}$, находим α бинпоиском, получаем $\alpha = 1.4655\dots$.

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:* $\lim\limits_{n\to+\infty} \frac{f(n)}{g(n)} = 0$

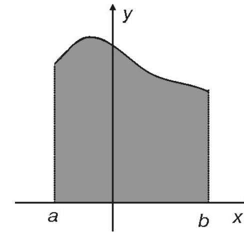
Def 1.8.2. $f = \mathcal{O}(g)$ *Определение через предел:* $\overline{\lim\limits_{n\to+\infty}} \frac{f(n)}{g(n)} < \infty$

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

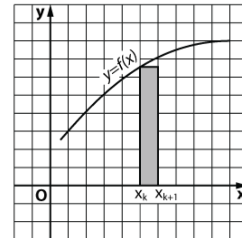
<u>Lm</u> 1.8.3. Определения o эквивалентны

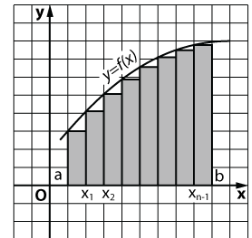
Доказательство. Вспомним, что речь о положительных функциях f и g.
Распишем предел по определению: $\forall C > 0 \quad \exists N \quad \forall n \geqslant N \quad \frac{f(n)}{g(n)} \leqslant C \Leftrightarrow f(n) \leqslant Cg(n)$. ■

1.9. (*) Замена сумм на интегралы



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





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

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

Доказательство. Сложили неравенства из Lm 1.9.2 ■

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

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

Теорема 1.9.5. *Замена суммы на интеграл #1*
$\forall f(x) \nearrow [1..\infty), f > 0, S(n) = \sum\limits_{i=1}^{n} f(i), I_1(n) = \int\limits_1^n, I_2(n) = \int\limits_1^{n+1}, I_1(n) = \Theta(I_2(n)) \Rightarrow \boxed{S(n) = \Theta(I_1(n))}$

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*
$\forall f(x) \nearrow [a..b], f > 0 \quad \int\limits_a^b f(x)dx \leqslant \sum\limits_{i=a}^{b} f(i) \leqslant f(b) + \int\limits_a^b f(x)dx$

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

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: $\int\limits_a^b f'(x)dx = f(b) - f(a)$

Пример: $\ln'(n) = \frac{1}{n} \Rightarrow \int\limits_1^n \frac{1}{x}dx = \ln n - \ln 1 = \ln n$

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 $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$

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

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: $[10..20) \to [20..40) \to [40..80) \to \dots$ ■

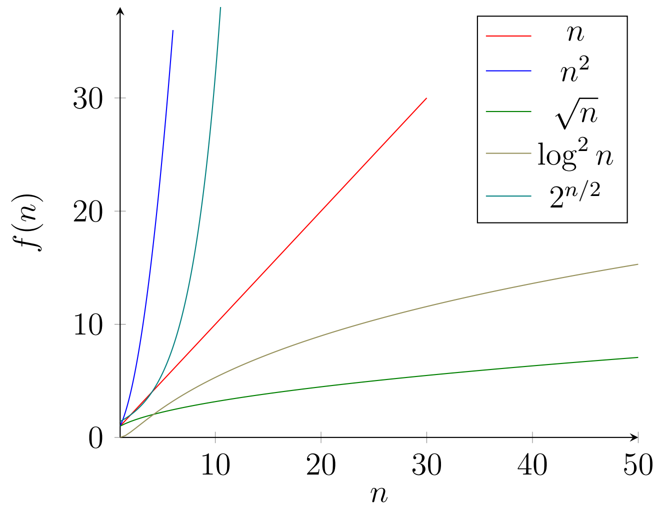
Следствие 1.11.7. $\forall x, y > 0, z > 1\colon\ \log^x n = \mathcal{O}(n^y), n^y = \mathcal{O}(z^n)$

Доказательство. Возьмём константу 1. ■

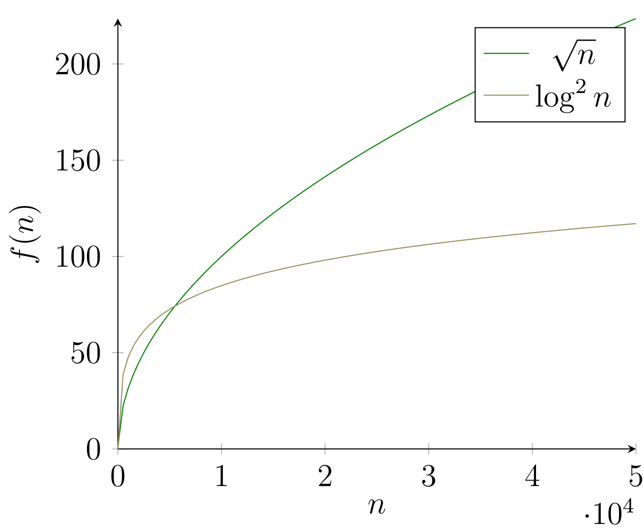
Следствие 1.11.8. $\forall x, y > 0, z > 1\colon\ \log^x n = o(n^y), n^y = o(z^n)$

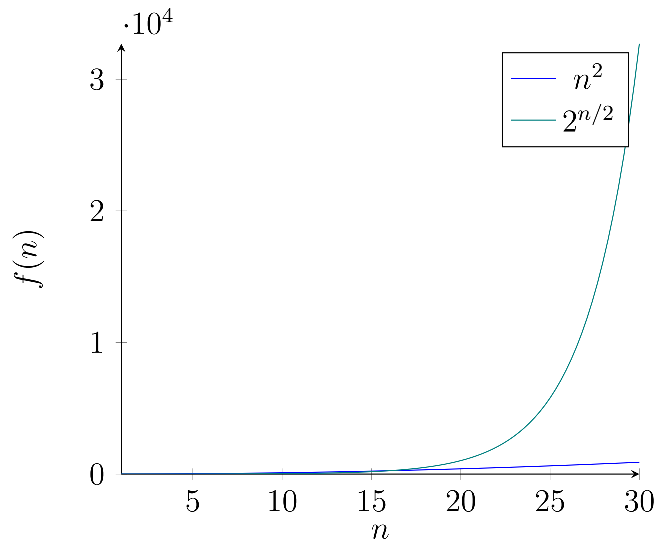
Доказательство. Достаточно перейти к чуть меньшим y, z и воспользоваться теоремой.
$\exists N\, \forall n \geqslant N\ \log^x n < n^{y-\varepsilon} = \frac{1}{n^\varepsilon} n^y,\ \frac{1}{n^\varepsilon} \underset{n\to\infty}{\to} 0 \Rightarrow \log^x = o(n^y)$.
$\exists N\, \forall n \geqslant N\ n^y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

• **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.

```
#define _GLIBCXX_DEBUG // должно быть до всех #include
// #define _LIBCPP_DEBUG 1 (аналог для компилятора clang)
#include <vector> // должно быть после
vector<int> a(3);
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)**

```
1 struct Point {
2   int x, y;
3 };
4 Point p, q = {2, 3}, *t = new Point {2, 3};
5 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.

```
1 int b = 3;
2 int *a = &b; // сохранили адрес b в пременную a типа int*
3 int c[10];
4 a = c; // указатель на первый элемент массива
5 *a = 7; // теперь c[0] == 7
6 Point *p = new Point {0, 0}; // выделили память под новый Point, указтель записали в p
7 (*p).x = 3; // записали значение в x
8 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`.

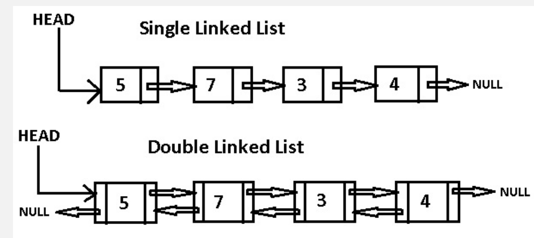
2.5. Двусвязный список

```
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.

```
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();
}
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 $\overline{\varphi} = y^2$.
$a_i = t_i + \Delta\overline{\varphi} = (y_{old} - y_{new} + 1) + (y^2_{new} - y^2_{old}) = \mathcal{O}(1)$
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.

```
1  stack<int> value; // уже посчитанные значения
2  stack<char> op; // ещё не выполненные операции
3  void make(): // выполнить последнюю невыполненную операцию
4      int b = value.top(); value.pop();
5      int a = value.top(); value.pop();
6      char o = op.top(); op.pop();
7      value.push(a o b); // да, не скомпилится, но смысл такой
8  int eval(string s): // пусть s без пробелов
9      s = '(' + s + ')' // при выполнении последней ')', выражение вычислится
10     for (char c : s)
11         if ('0' <= c && c <= '9') value.push(c - '0'); // просто добавили
12         else if (c == '(') op.push(c); // просто добавили, её приоритет меньше всех
13         else if (c == ')') { // закрылась? ищем парную открывающую на стеке
14             while (op.top() != '(') make();
15             op.pop();
16         } else { // пришла операция? можно выполнить все предыдущие большего приоритета
17             while (op.size() && priorioty[op.top()] >= priorioty[c]) make();
18             op.push(c);
19         }
20     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

```
1 int find(int l, int r, int x): // [l,r]
2     while (l <= r) {
3         int m = (l + r) / 2;
4         if (a[m] == x) return m;
5         if (a[m] < x) l = m + 1;
6         else r = m - 1;
7     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.

```
1 while (R - L > 1) {
2     int m = (L + R) / 2; // 0 ⩽ m < n ⇒ нет выхода за пределы a[]
3     if (a[m] == 0) // можно просто (!a[m] ? L : R) = m;
4         L = m;
5     else
6         R = m;
7 } // 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`.

```
1 int lower_bound(int l, int r, int x): // [l,r)
2     while (R - L > 1) {
3         int m = (L + R) / 2;
4         if (a[m] < x)
5             L = m;
6         else
7             R = m;
8     } // 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

```
1 int a[n]; // массив из n элементов
2 i = lower_bound(a, a + n, x) - a; // min i: a[i] >= x
3 i = upper_bound(a, a + n, x) - a; // min i: a[i] > x
4 vector<int> a(n);
5 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 любом 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`).

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

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

```
1 bool f(int i) { return a[i] >= x; }
2 int l = -1, r = n; // мысленно добавим a[-1] = −∞, a[n] = +∞
3 find_predicate(l, r); // f() будет вызываться только для элементов от l+1 до r-1
4 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

```
1 list<int> h[N]; // собственно хеш-таблица
2 void add(int x) { h[x % N].push_back(x); } // O(1) в худшем
3 auto find(int x) { return find(h[x % N].begin(), h[x % N].end(), x); }
4 // find работает за длину списка
5 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.

```
1  int h[N]; // собственно хеш-таблица
2  // h[i] = 0 : пустая ячейка
3  // h[i] = -1 : удалённый элемент
4  // h[i] > 0 : лежит что-то полезное
5  int getIndex(int x): // поиск индекса по элементу, требуем x > 0
6      int i = x % N; // используем хеш-функцию
7      while (h[i] && h[i] != x)
8          if (++i == N) // массив циклический
9              i = 0;
10     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 = $1+\sum\limits_{k=1}^{\infty}$ (вероятность того, что мы сделали k-й шаг) = $1+\alpha+\alpha^2+\alpha^3+\cdots = \frac{1}{1-\alpha}$. ∎

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

Доказательство. Тест: добавляем числа $0, 2, \ldots n$ и $r, r{+}1, r{+}2, \ldots r{+}n$, 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.

```
1  int *a, *b; // выделенные области памяти
2  int pos = -1; // разделитель скопированной и не скопированной частей
3  int n, size; // количество элементов; выделенная память
4  void push_back(int x):
5      if (pos >= 0) b[pos] = a[pos], pos--;
6      if (n == size):
7          delete [] a; // мы его уже скопировали, он больше не нужен
8          a = b;
9          size *= 2;
10         pos = n - 1, b = new int[size];
11     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.

```
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();
```

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.

```
1 Stack a, b, a1, b1;
2 void push(int x):
3     b1.push(x); // кидаем не в b, а в b1, копию b
4     STEP; // сделать несколько шагов копирования
5 int pop():
6     if (копирование завершено)
7         a = a1, b = b1, начать новое копирование;
8     STEP; // сделать несколько шагов копирования
9     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 $\sum\limits_{i=1}^{k} 2^{k-i} i = 2^k \sum\limits_{i=1}^{k} \frac{i}{2^i} \overset{(*)}{=} 2^k \cdot \Theta(1) = \Theta(n)$. `(*) доказано на практике.` ∎

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

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:

```
1 int pos = 0; // указатель на первую свободную ячейку
2 int new(uint32_t n): // push n bytes
3     pos += n;
4     assert(pos <= MAX_MEM); // проверить, что памяти всё ещё хватает
5     return pos - n;
6 void delete(uint32_t old_pos): // освободили всю память, выделенную с момента old_pos
7     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. *Особождение 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, получаещая 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 $N =$

$2^{a_1}+2^{a_2}+\dots+2^{a_k}$ элементов будем хранить k структур `S` из $2^{a_1}, 2^{a_2}, \dots, a^{a_k}$ 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.

```
1 for (i = 1; есть две структуры размера i; i *= 2)
2     Добавим S.Build(A.AllElements + B.AllElements). // A, B - те самые две структуры
3     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), ..., `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. $\forall k \geqslant 1, a_i > 0\colon (\sum a_i)^k \geqslant \sum a_i^k$ (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 $x\colon l \leqslant x \leqslant r$
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 ⇔ $\mathtt{I} = 0$

• **Selection sort** (сортировка выбором)

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

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

• **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.

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

```
1 int n;
2 vector<int> a(n), buffer(n);
3 for (int k = 0; (1 << k) < n; k++)
4     for (int i = 0; i < n; i += 2 * (1 << k))
5         Merge(i, min(n, i + (1 << k), min(n, i + 2 * (1 << k)), a, buffer)
6     swap(a, buffer); // O(1)
7 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

```
1 def QuickSort(a):
2     if len(a) <= 1: return a
3     x = random.choice(a)
4     b0 = select (< x).
5     b1 = select (= x).
6     b2 = select (> x).
7     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`.

```
1 void Partition(int l, int r, int x, int *a, int &i, int &j): // [l,r], x ∈ a[l,r]
2     i = l, j = r;
3     while (i <= j):
4         while (a[i] < x) i++;
5         while (a[j] > x) j--;
6         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:

```
1 void QuickSort(int l, int r, int *a): // [l, r]
2     if (l >= r) return;
3     int i, j;
4     Partition(l, r, a[random [l, r]], i, j);
5     QuickSort(l, j, a); // j < i
6     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 Мо, 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`) $\{x\colon$ `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.

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

Воспользуемся индукционным предположением (индукция же!)

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

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 ⩽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.

<u>Lm</u> 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 $i+1, i+2, \dots, j-1$ также содержатся (т.к. i и j — индексы в отсортированном массиве). i и j разделятся ⇒ `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:

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

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:

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

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:

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

Каждое число x_j попадает в отрезок номер $i_j = \lfloor\frac{x_j-\mathtt{min}}{\mathtt{max}-\mathtt{min}+1}n\rfloor$. 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. $\mathtt{max}-\mathtt{min} \leqslant n \Rightarrow$ и BB, и BI работают за $\Theta(n)$

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

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

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 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.

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

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 $[0, 2^{2^k})$.

```
1 struct Heap<k> {
2     int min, size; // отдельно храним минимальный элемент и размер
3     Heap<k-1>* notEmpty; // номера непустых кусков
4     unoredered_map<int, Heap<k-1>*> parts; // собственно куски
5 };
```

• Как добавить новый элемент?

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

```
1 void Heap<k>::add(int x): // size ⩾ 2, k ⩾ 2, разбираем только интересный случай
2     int i = index(x);
3     if parts[i] is empty
4         notEmpty->add(i); // появился новый непустой кусок, рекурсивный вызов
5         parts[i] = {x}; // O(1)
6     else
7         parts[i]->add(x); // рекурсивный вызов
8     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?

```
1 void Heap<k>::del(int x): // size ⩾ 2, k ⩾ 2, разбираем только интересный случай
2     int i = index(x);
3     if parts[i]->size == 1
4         notEmpty->del(i); // кусок стал пустым, делаем рекурсивный вызов
5         parts[i] = empty heap
6     else
7         parts[i]->del(i)
8     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`$(3, 4, z) \dots$ 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.

```
1  vector<int> Sort(int k, vector<int> &x) {
2      int n = x.size()
3      if (n == 1 || n >= 2^{2^k}) return CountSort(x) // за O(n)
4      vector<int> a(n), b(n), as, result;
5      unordered_map<int, vector<int>> BS; // хеш-таблица
6      for (int i = 0; i < n; i++) {
7          a[i] = старшие 2^{k-1} бит x[i];
8          b[i] = младшие 2^{k-1} бит x[i];
9          BS[a[i]].push_back(b[i]); // для каждого a[i] храним все парные с ним b[i]
10     }
11     for (auto &p : A) as.push_back(p.first); // храним все a-шки, каждую 1 раз
12     as = Sort(k - 1, as); // отсортировали все a[i]
13     for (int a : as) {
14         vector<int> &bs = BS[a]; // теперь нужно отсортировать вектор bs
15         int i = max_element(bs.begin(), bs.end()) - bs.begin(), max_b = bs[i];
16         swap(bs[i], bs.back()), bs.pop_back(); // удалили максимальный элемент
17         bs = Sort(k - 1, bs); // отсортировали всё кроме максимума
18         for (int b : bs) result.push_back(<a, b>); // выписали результат без максимума
19         result.push_back(<a, max_b>); // отдельно добавили максимальный элемент
20     }
21     return result;
22 }
```

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:

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

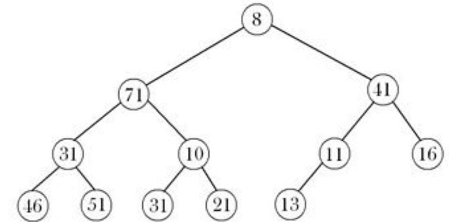
Рассмотрим случай полного бинарного дерева $n = 2^k - 1 \Rightarrow$
$\sum \log size_i = (\log 3)\frac{n+1}{4} + (\log 7)\frac{n+1}{8} + (\log 15)\frac{n+1}{16} + \dots = (n+1)(\frac{\log 3}{4} + \frac{\log 7}{8} + \frac{\log 15}{16} + \dots)$.
При $n \to +\infty$ величина $\frac{\sum \log size_i}{n+1}$ 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 $d(\texttt{NULL}) = 0$, $d(v) = \min(d(v.left), d(v.right)) + 1$

<u>Lm</u> 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.

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

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.

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

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.

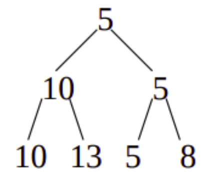
• **План.** Нам нужно несколько идей. Пройдём идеи, соединим их в quake-heap.

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;
```

$R = |$`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 $t_{quake} = n_{i+1} + n_{i+2} + n_{i+3} + \cdots = \Theta(n_{i+1})$ (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 $a_{quake} = t_{quake} + \Delta B \leqslant 9|\Delta B| + 9|\Delta B| + 20\Delta B < 0$.

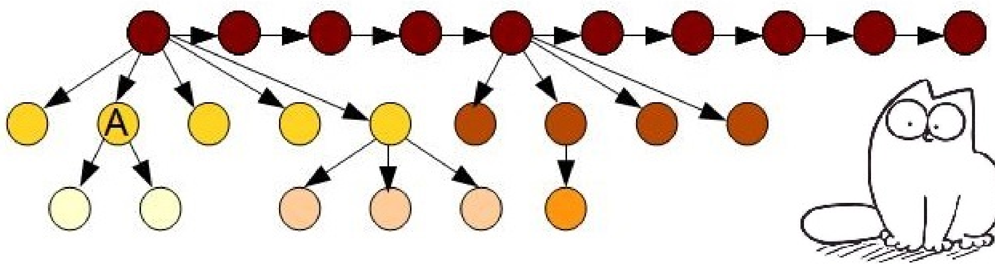
7.6. (*) Pairing Heap

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:

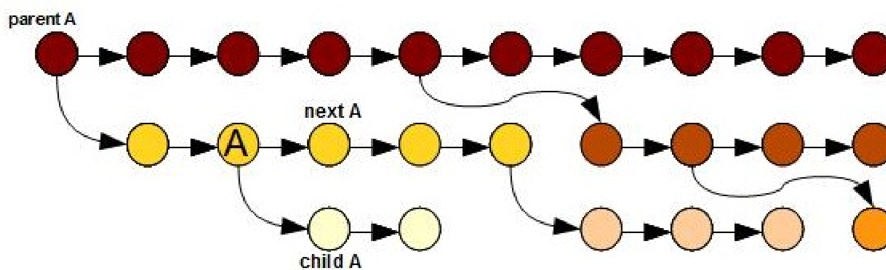
- `PairingHeap = minElement + список детей вида PairingHeap`

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.

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

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

```
1 DecreaseKey(a, newX):
2     a->x = newX;
3     ListDelete(a); // удалили a из списка её отца
4     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.

```
1 def ExtractMin:
2     root = Pairing(root->child)
3 def Pairing(a): # пусть наши списки питоно-подобны, "a- список куч
4     if |a| = 0 return Empty
5     if |a| = 1 return a[0]
6     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.

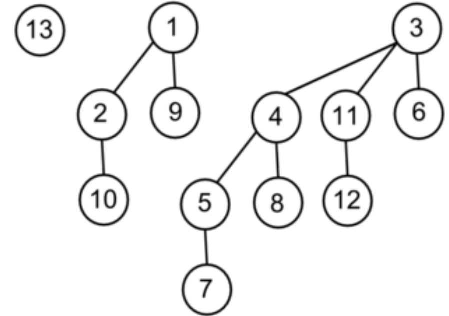
Оценка $\mathcal{O}(\log\log n)$ для оригинальной Pairing Heap на 2019-й год не доказана.

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

```
1 Node* join(Node* a, Node* b): // a->rank == b->rank
2     if (a->x > b->x) swap(a, b);
3     b->next = a->child, a->child = b; // добавили b в список детей a
4     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.

```
 1 list<Node*> Normalize(list<Node*> &a):
 2     list<Node*> roots[maxRank+1], result;
 3     for (Node* v : a) roots[v->rank].push_back(v);
 4     for (int i = 0; i <= maxRank; i++)
 5         while (roots[i].size() >= 2):
 6             Node* a = roots[i].back(); roots[i].pop_back();
 7             Node* b = roots[i].back(); roots[i].pop_back();
 8             roots[i+1].push_back(join(a, b));
 9         if (roots[i].size()): result.push_back(roots[i][0]);
10     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. $t_i = \texttt{Roots} + \texttt{maxRank}$, $\Delta\varphi \leqslant \log_2 n - \texttt{Roots} \Rightarrow a_i = t_i + \Delta\varphi = \mathcal{O}(\log n)$. 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`).

```
1 struct Node:
2     Node *child, *parent, *left, *right;
3     int x, degree; // ранг биномиального дерева равен степени
4     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.

```
1  list<Node*> heap; // куча - список корней биномиальных деревьев
2  void CascadingCut(Node *v):
3      Node *p = v->parent;
4      if (p == NULL) return; // мы уже корень
5      p->degree--; // поддерживаем степень, будем её потом использовать, как ранг
6      v->left->right = v->right, v->right->left = v->left; // удалили из списка
7      heap.push_back(v), v->marked = 0; // начнём новую жизнь в качестве корня!
8      if (p->parent && p->marked++) // если папа - корень, ничего не нужно делать
9          CascadingCut(p); // у p только что отрезали второго сына, пора сделать его корнем
10
11 void DecreaseKey(int i, int x): // i - номер элемента
12     pos[i]->x = x; // pos[i] = обратная ссылка
13     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 $\texttt{rank} = \texttt{degree}$. В общем случае:

Lm 7.8.2. $\texttt{v.rank} = \texttt{v.degree} + \texttt{v.mark}$

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 $\varphi = \texttt{Roots} + 2\texttt{Marked}$.
Амортизированное время операций кроме `DecreaseKey` не поменялось, так как они не меняют `Marked`. Пусть `DecreaseKey` отрезал $k+1$ вершину, тогда $\Delta\texttt{Marked} \leqslant -k$, $\Delta\texttt{Roots} \leqslant k+1$, $a_i = t_i + \Delta\varphi = t_i + \Delta\texttt{Roots} + 2\Delta\texttt{Marked} \leqslant (k+1) + (k+1) - 2k = \Theta(1)$. ■

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. $\forall n \geqslant 2\colon S_n = 1 + S_0 + S_1 + \dots + S_{n-2}$

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. $\forall n \geqslant 2\colon Fib_n = 1 + Fib_0 + Fib_1 + \dots + Fib_{n-2}$

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

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 задачу «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.

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

Можно также перебирать перестановки a не меняя порядок исходного массива.

```
1 vector<int> p = {0, 1, ... n-1};
2 do { // перемешанный массив a: a[p[0]], a[p[1]], a[p[2]],...
3 } 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...

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

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

```
1  void go(int i, int s): // i -- позиция в перестановке
2      if (i == n):
3          best = max(best, s); // решение сложной из двух версий задачи выше
4          return
5      for (int x = 0; x < n; x++)
6          if (!used[x]):
7              if (abs(i - x) > 2) continue; // пропускаем лишние
8              used[x] = 1 // далее нельзя использовать элемент x
9              p[i] = x, go(i+1, s + p[i]*b[i]); // пересчитали сумму
10             used[x] = 0 // а в других ветках рекурсии можно
11 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.

```
1 void go(int i, int sum):
2     if (sum > S) return // оптимизация!
3     if (i == n)
4         if (sum == S) // набрали подмножество суммы S
5         return
6     used[i]=0, go(i + 1, sum) // не берём
7     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).

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

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

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

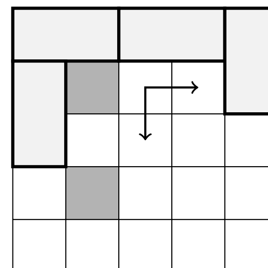
```
1 int go(int n, int x):
2     if (n == 0): return 1 // ровно 1 способ разбить 0 на слагаемые
3     return go(n, x - 1) + (n<x ? 0 : go(n-x, x))
```

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.



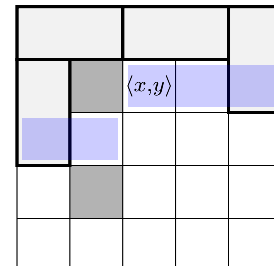
```
1  int go(int x, int y): {
2      if (x == w) x = 0, y++; // начали следующую строку
3      if (y == h) return 1; // все строки заполнены, 1 способ закончить заполнение
4      if (!empty[y][x]) return go(x + 1, y);
5      int result = 0;
6      if (y + 1 < h && empty[y + 1][x]):
7          empty[y + 1][x] = empty[y][x] = 0; // поставили вертикальную доминошку
8          result += go(x + 1, y);
9          empty[y + 1][x] = empty[y][x] = 1; // убрали за собой
10     if (x + 1 < w && empty[y][x+1]): // аналогично для горизонтальной
11         empty[y + 1][x] = empty[y][x] = 0;
12         result += go(x + 1, y);
13         empty[y + 1][x] = empty[y][x] = 1;
14     return result;
15 }
```

`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 $2^{\text{число доминошек}} \leqslant 2^{wh/2}$. Давайте теперь, как и задаче про клики добавим к нашему перебору запоминание. Что есть состояние перебора? Вся матрица.

Получаем следующий код:

```
1  vector<vector<bool>> empty;
2  map<vector<vector<bool>>, int> m; // запоминание
3  int go(int x, int y):
4      if (x == w) x = 0, y++; // начали следующую строку
5      if (y == h) return 1; // все строки заполнены, 1 способ закончить заполнение
6      if (!empty[y][x]) return go(x + 1, y);
7      if (m.count(empty)) return m[empty];
8      int result = &m[empty];
9      if (y + 1 < h && empty[y + 1][x]):
10         empty[y + 1][x] = empty[y][x] = 0; // поставили вертикальную доминошку
11         result += go(x + 1, y);
12         empty[y + 1][x] = empty[y][x] = 1; // убрали за собой
13     if (x + 1 < w && empty[y][x+1]): // аналогично для горизонтальной
14         empty[y + 1][x] = empty[y][x] = 0;
15         result += go(x + 1, y);
16         empty[y + 1][x] = empty[y][x] = 1;
17     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. ■

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

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

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

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

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

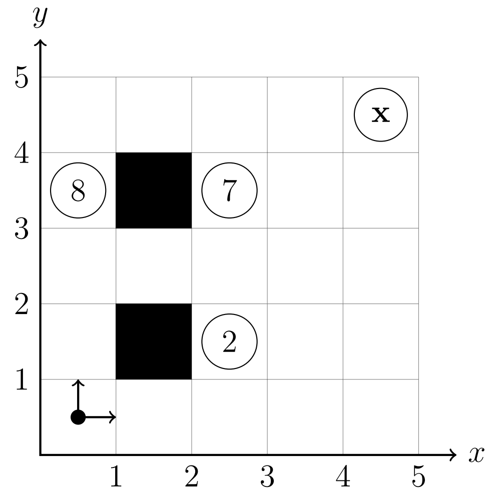
Это рекурсивный способ писать динамику назад, вычисляя значение только для тех состояний, которые действительно нужно посчитать.

```
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:

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

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

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:

```
1  cnt <-- 0, f <-- -∞; // нейтральные значения
2  cnt[0,0] = 1, f[0,0] = 0; // база
3  for (int x = 0; x < width; x++)
4      for (int y = 0; y < height; y++):
5          if (клетка не проходима) continue;
6          cnt[x+1,y] += cnt[x,y];
7          cnt[x,y+1] += cnt[x,y];
8          f[x,y] += value[x,y];
9          relax(f[x+1,y], f[x,y]);
10         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:

```
1  void relax(int x, int y, int F, int P):
2      if (f[x,y] < F)
3          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--;
```

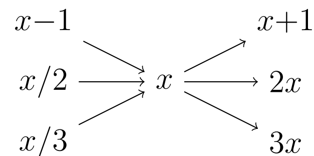
Второму способу нужно меньше памяти, но обычно он требует больше строк кода.

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

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

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

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* $a_{i_1}, a_{i_2}, \ldots, a_{i_k} \colon 1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$

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 $j\colon f[n',j] + g[n{-}n', m{-}j] = \max$, 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 ⇒ 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 #10: Формула включения-исключения

2024/25

10.1. (*) 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 $\sum\limits_{A \in [0, 2^k)} (-1)^{|A|} \lfloor \frac{n}{lcm(p_i \,|\, i \in A)} \rfloor$.

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 $\langle x_i, y_i \rangle$.

10.2. (*) 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