

# SPb HSE, 1 курс, осень 2021/22

## Конспект лекций по алгоритмам

Собрано 19 ноября 2021 г. в 12:09

---

### Содержание

<b>0. Разбор теста</b>	<b>1</b>
0.1. Разбор основных задач	1
<b>1. Асимптотика</b>	<b>2</b>
1.1. $\mathcal{O}$ -обозначения	2
1.2. Рекуррентности и Карацуба	3
1.3. Теоремы о рекуррентных соотношениях	5
1.4. Доказательства по индукции	6
1.5. Числа Фибоначчи	6
1.6. (*) $\mathcal{O}$ -обозначения через пределы	6
1.7. (*) Замена сумм на интегралы	7
1.8. Примеры по теме асимптотики	8
1.9. Сравнение асимптотик	9
<b>2. Структуры данных</b>	<b>10</b>
2.1. C++	11
2.2. Неасимптотические оптимизации	12
2.3. Частичные суммы	13
2.4. Массив	13
2.5. Двусвязный список	13
2.6. Односвязный список	14
2.7. Список на массиве	14
2.8. Вектор (расширяющийся массив)	15
2.9. Стек, очередь, дек	15
2.10. Очередь, стек и дек с минимумом	16
<b>3. Структуры данных</b>	<b>16</b>
3.1. Амортизационный анализ	17
3.2. Разбор арифметических выражений	18
3.3. Бинарный поиск	19
3.3.1. Обыкновенный	19
3.3.2. По предикату	19
3.3.3. Вещественный, корни многочлена	20
3.4. Два указателя и операции над множествами	20
3.5. Хеш-таблица	21
3.5.1. Хеш-таблица на списках	21
3.5.2. Хеш-таблица с открытой адресацией	21
3.5.3. Сравнение	22
3.5.4. C++	23

<b>4. Структуры данных</b>	<b>24</b>
4.1. Избавляемся от амортизации	24
4.1.1. Вектор (решаем проблему, когда случится)	24
4.1.2. Вектор (решаем проблему заранее)	24
4.1.3. Сравнение способов	24
4.1.4. Хеш-таблица	25
4.1.5. Очередь с минимумом через два стека	25
4.2. Бинарная куча	27
4.2.1. GetMin, Add, ExtractMin	27
4.2.2. Обратные ссылки и DecreaseKey	28
4.2.3. Build, HeapSort	28
4.3. Аллокация памяти	29
4.3.1. Стек	29
4.3.2. Список	29
4.3.3. Куча	30
4.4. Пополняемые структуры	31
4.4.1. Ничего → Удаление	31
4.4.2. Поиск → Удаление	31
4.4.3. Add → Merge	31
4.4.4. Build → Add	32
4.4.5. Build → Add, Del	33
<b>5. Сортировки</b>	<b>34</b>
5.1. Два указателя и алгоритм Мо	34
5.2. Квадратичные сортировки	35
5.3. Оценка снизу на время сортировки	36
5.4. Решение задачи по пройденным темам	36
5.5. Быстрые сортировки	36
5.5.1. CountSort (подсчётом)	36
5.5.2. MergeSort (слиянием)	37
5.5.3. QuickSort (реально быстрая)	38
5.5.4. Сравнение сортировок	38
5.6. (*) Adaptive Heap sort	38
5.6.1. (*) Модифицированный HeapSort	39
5.6.2. (*) Adaptive Heap Sort	39
5.6.3. (*) Ссылки	39
5.7. (*) Kirkpatrick'84 sort	40
<b>6. Сортировки (продолжение)</b>	<b>41</b>
6.1. Quick Sort	41
6.1.1. Оценка времени работы	41
6.1.2. Introsort'97	42
6.2. Порядковые статистики	43
6.2.1. Одноветочный QuickSort	43
6.2.2. Детерминированный алгоритм	43
6.2.3. C++	44
6.3. Integer sorting	44

6.3.1. Count sort	44
6.3.2. Radix sort	44
6.3.3. Bucket sort	45
6.4. Van Embde Boas'75 trees	46
6.5. (*) Inplace merge за $\mathcal{O}(n)$	47
6.6. (*) 3D Mo	48
6.6.1. (*) Применяем для mex	48
<b>7. Кучи</b>	<b>48</b>
7.1. Нижняя оценка на построение бинарной кучи	49
7.2. Min-Max Heap (Atkison'86)	49
7.3. Leftist Heap (Clark'72)	51
7.4. Skew Heap (Tarjan'86)	51
7.5. Списко-куча	52
7.6. (*) Pairing Heap	53
7.6.1. (*) История. Ссылки.	54
<b>8. Кучи</b>	<b>56</b>
8.1. Биномиальная куча (Vuillemin'78)	56
8.1.1. Основные понятия	56
8.1.2. Операции с биномиальной кучей	57
8.1.3. Add и Merge за $\mathcal{O}(1)$	57
8.2. Куча Фибоначчи (Fredman, Tarjan'84)	57
8.2.1. Фибоначчиевы деревья	59
8.2.2. Завершение доказательства	59
<b>9. Динамическое программирование</b>	<b>60</b>
9.1. Базовые понятия	60
9.1.1. Условие задачи	60
9.1.2. Динамика назад	60
9.1.3. Динамика вперёд	61
9.1.4. Ленивая динамика	61
9.2. Ещё один пример	62
9.3. Восстановление ответа	62
9.4. Графовая интерпретация	63
9.5. Checklist	64
9.6. Рюкзак	64
9.6.1. Формулировка задачи	64
9.6.2. Решение динамикой	64
9.6.3. Оптимизируем память	64
9.6.4. Добавляем <code>bitset</code>	65
9.6.5. Восстановление ответа с линейной памятью	65
9.7. Квадратичные динамики	66
9.8. Оптимизация памяти для НОП	67
9.8.1. Храним биты	67
9.8.2. Алгоритм Хиршберга (по wiki)	67
9.8.3. Оценка времени работы Хиршберга	67

9.8.4. Алгоритм Хиршберга (улучшенный) . . . . .	68
9.8.5. Область применение идеи Хиршберга . . . . .	68

# Лекция #0: Разбор теста

1 сентября

## 0.1. Разбор основных задач

Задача: дано  $n$ , найти число решений  $a^2 + b^2 = n$

Решение, за  $\mathcal{O}(\sqrt{n})$ :

```
1 for (a = 1; a * a <= n; a++) {
2     int b = sqrt(n - a * a);
3     res += (a * a + b * b == n);
4 }
```

Решение, за  $\mathcal{O}(\sqrt{n})$  элементарных арифметических операций с целыми числами:

```
1 int m = sqrt(n / 2), b = sqrt(n);
2 for (int a = 1; a <= m; a++) {
3     // Идея: если a увеличивается, то b обязательно уменьшается!
4     while ((tmp = a * a + b * b) > n) --b; // 2(n/2)1/2 умножений
5     if (tmp == n) res += (a == b ? 1 : 2);
6 }
```

## Лекция #1: Асимптотика

3 сентября

1.1.  $\mathcal{O}$ -обозначения

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

**Def 1.1.1.**  $f = \Theta(g)$   $\exists N > 0, C_1 > 0, C_2 > 0: \forall n \geq N, C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$

**Def 1.1.2.**  $f = \mathcal{O}(g)$   $\exists N > 0, C > 0: \forall n \geq N, f(n) \leq C \cdot g(n)$

**Def 1.1.3.**  $f = \Omega(g)$   $\exists N > 0, C > 0: \forall n \geq N, f(n) \geq C \cdot g(n)$

**Def 1.1.4.**  $f = o(g)$   $\forall C > 0 \exists N > 0: \forall n \geq N, f(n) \leq C \cdot g(n)$

**Def 1.1.5.**  $f = \omega(g)$   $\forall C > 0 \exists N > 0: \forall n \geq N, f(n) \geq C \cdot g(n)$

Понимание  $\Theta$ : “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”

$\Theta$	$\mathcal{O}$	$\Omega$	$o$	$\omega$
$=$	$\leq$	$\geq$	$<$	$>$

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

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

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

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

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

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

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

Докажем для примера 1.1.6.

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

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

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

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

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

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

*Доказательство.*  $g = o(f) \exists N: \forall n \geq N \quad g(n) \leq \frac{1}{2} f(n) \Rightarrow \frac{1}{2} f(n) \leq f(n) \pm g(n) \leq \frac{3}{2} f(n)$  ■

**Лм 1.1.18.**  $n^k = o(n^{k+1})$

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

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

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

## 1.2. 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 1.2.1.* 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  $A \cdot B = (A_1 + x^k A_2)(B_1 + x^k B_2) = A_1 B_1 + x^k (A_1 B_2 + A_2 B_1) + x^{2k} A_2 B_2$ .

Если написать рекурсивную функцию умножения, то получим время работы:

$$T_1(n) = 4T_1\left(\frac{n}{2}\right) + \Theta(n)$$

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

$$T_2(n) = 3T_2\left(\frac{n}{2}\right) + \Theta(n)$$

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 = 2k, 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 wi - сдвиг массива на i вправо
9   return x + y * wn + (z - x - y) * wn/2;
```

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



### 1.3. Теоремы о рекуррентных соотношениях

**Теорема 1.3.1.** *Мастер Теорема* (теорема о простом рекуррентном соотношении)

Пусть  $T(n) = aT(\frac{n}{b}) + f(n)$ , где  $f(n) = n^c$ . При этом  $a > 0, b > 1, c \geq 0$ . 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:

$$T(n) = f(n) + aT(\frac{n}{b}) = f(n) + af(\frac{n}{b}) + a^2f(\frac{n}{b^2}) + \dots = n^c + a(\frac{n}{b})^c + a^2(\frac{n}{b^2})^c + \dots$$

Тогда  $T(n) = f(n)(1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + \dots + (\frac{a}{b^c})^k)$ . 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 1.3.2.** *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  $a > 0, b > 1, c \geq 0, d \geq 0$ .

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

Без доказательства. ■

**Теорема 1.3.3.** *Об экспоненциальном рекуррентном соотношении*

Пусть  $T(n) = \sum b_i T(n - a_i)$ . При этом  $a_i > 0, b_i > 0, \sum b_i > 1$ .

Тогда  $T(n) = \Theta(\alpha^n)$ , при этом  $\alpha > 1$  и является корнем уравнения  $1 = \sum b_i \alpha^{-a_i}$ , его можно найти бинарным поиском.

*Доказательство.* Предположим, что  $T(n) = \alpha^n$ , тогда  $\alpha^n = \sum b_i \alpha^{n - a_i} \Leftrightarrow 1 = \sum b_i \alpha^{-a_i} = f(\alpha)$ .

Теперь нам нужно решить уравнение  $f(\alpha) = 1$  для  $\alpha \in [1, +\infty)$ .

Если  $\alpha = 1$ , то  $f(\alpha) = \sum b_i > 1$ , если  $\alpha = +\infty$ , то  $f(\alpha) = 0 < 1$ . Кроме того  $f(\alpha) \searrow [1, +\infty)$ .

Получаем, что на  $[1, +\infty)$  есть единственный корень уравнения  $1 = f(\alpha)$  и его можно найти бинарным поиском.

Мы показали, откуда возникает уравнение  $1 = \sum b_i \alpha^{-a_i}$ . Доказали, что у него  $\exists!$  корень  $\alpha$ .

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:

$$\exists C: \forall n \in B = [1 - \max_i a_i, 1] \quad T(n) \leq C \alpha^n$$

Переход индукции:

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

(\*) Верно, так как  $\alpha$  – корень уравнения. ■

## 1.4. Доказательства по индукции

### Lm 1.4.1. Доказательство по индукции

Есть простой метод решения рекуррентных соотношений: угадать ответ, доказать его по индукции. Рассмотрим на примере  $T(n) = \max_{x=1..n-1} (T(x) + T(n-x) + x(n-x))$ .

Докажем, что  $T(n) = \mathcal{O}(n^2)$ , для этого достаточно доказать  $T(n) \leq n^2$ :

База:  $T(1) = 1 \leq 1^2$ .

Переход:  $T(n) \leq \max_{x=1..n-1} (x^2 + (n-x)^2 + x(n-x)) \leq \max_{x=1..n-1} (x^2 + (n-x)^2 + 2x(n-x)) = n^2$

### • 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 1.3.3, получаем  $1 = \alpha^{-1} + \alpha^{-3}$ , находим  $\alpha$  бинпоиском, получаем  $\alpha = 1.4655\dots$

## 1.5. Числа Фибоначчи

**Def 1.5.1.**  $f_1 = f_0 = 1, f_i = f_{i-1} + f_{i-2}$ .  $f_n$  -  $n$ -е число Фибоначчи.

### • Оценки снизу и сверху

$f_n = f_{n-1} + f_{n-2}$ , рассмотрим  $g_n = g_{n-1} + g_{n-1}, 2^n = g_n \geq f_n$ .

$f_n = f_{n-1} + f_{n-2}$ , рассмотрим  $g_n = g_{n-2} + g_{n-2}, 2^{n/2} = g_n \leq f_n$ .

Воспользуемся 1.3.3, получим  $1 = \alpha^{-1} + \alpha^{-2} \Leftrightarrow \alpha^2 - \alpha - 1 = 0$ , получаем  $\alpha = \frac{\sqrt{5}+1}{2} \approx 1.618$ .

$f_n = \Theta(\alpha^n)$ .

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

**Def 1.6.1.**  $f = o(g)$  Определение через предел:  $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$

**Def 1.6.2.**  $f = \mathcal{O}(g)$  Определение через предел:  $\overline{\lim}_{n \rightarrow +\infty} \frac{f(n)}{g(n)} < \infty$

Здесь необходимо пояснение:  $\overline{\lim}_{n \rightarrow +\infty} f(n) = \lim_{n \rightarrow +\infty} (\sup_{x \in [n..+\infty]} f(x))$ , где  $\sup$  - верхняя грань.

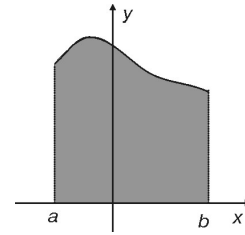
### Lm 1.6.3. Определения $\mathcal{O}$ эквивалентны

*Доказательство.* Вспомним, что речь о положительных функциях  $f$  и  $g$ .

Распишем предел по определению:  $\forall C > 0 \quad \exists N \quad \forall n \geq N \quad \frac{f(n)}{g(n)} \leq C \Leftrightarrow f(n) \leq Cg(n)$ . ■

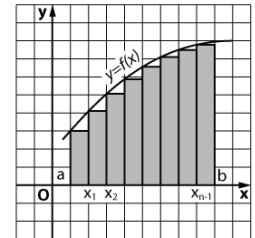
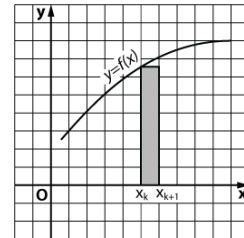
### 1.7. (\*) Замена сумм на интегралы

**Def 1.7.1.** Определённый интеграл  $\int_a^b f(x)dx$  положительной функции  $f(x)$  – площадь под графиком  $f$  на отрезке  $[a..b]$ .



**Lm 1.7.2.**  $\forall f(x) \nearrow [a..a+1] \Rightarrow f(a) \leq \int_a^{a+1} f(x)dx \leq f(a+1)$

**Lm 1.7.3.**  $\forall f(x) \nearrow [a..b+1] \Rightarrow \sum_{i=a}^b f(i) \leq \int_a^{b+1} f(x)dx$



*Доказательство.* Сложили неравенства из 1.7.2 ■

**Lm 1.7.4.**  $\forall f(x) \nearrow [a..b], f > 0 \Rightarrow \int_a^b f(x)dx \leq \sum_{i=a}^b f(i)$

*Доказательство.* Сложили неравенства из 1.7.2, выкинули  $[a-1, a]$  из интеграла. ■

#### Теорема 1.7.5. Замена суммы на интеграл #1

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

*Доказательство.* Из лемм 1.7.3 и 1.7.4 имеем  $I_1(n) \leq S(n) \leq I_2(n)$ .

$C_1 I_1(n) \leq I_2(n) \leq C_2 I_1(n) \Rightarrow I_1(n) \leq S(n) \leq I_2(n) \leq C_2 I_1(n)$  ■

#### Теорема 1.7.6. Замена суммы на интеграл #2

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

*Доказательство.* Первое неравенство – лемма 1.7.3. Второе – 1.7.4, применённая к  $\sum_{i=a}^{b-1}$ . ■

*Следствие 1.7.7.* Для убывающих функций два последних факта тоже верны. Во втором ошибкой будет не  $f(b)$ , а  $f(a)$ , которое теперь больше.

#### • Как считать интегралы?

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

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

## 1.8. Примеры по теме асимптотики

### • Вложенные циклы for

```

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

```

Чему равен counter? Во-первых, есть точный ответ:  $\binom{n}{5} \approx \frac{n^5}{5!}$ . Во-вторых, мы можем сходно 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?

```

1 f[0] = f[1] = 1;
2 for (int i = 2; i < n; i++)
3     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 $a^2 + b^2 = N$

```

1 int b = sqrt(N);
2 for (int a = 1; a * a <= N; a++)
3     while (a * a + b * b >= N; b--)
4         ;
5     if (a * a + b * b == N)
6         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

```

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

```

За сколько работает программа?

$$\sum_{a=1}^n \left\lceil \frac{n}{a} \right\rceil = \mathcal{O}(n) + \sum_{a=1}^n \frac{n}{a} = \mathcal{O}(n) + n \sum_{a=1}^n \frac{1}{a} \stackrel{1.7.5}{=} \mathcal{O}(n) + n \cdot \Theta\left(\int_1^n \frac{1}{x} dx\right) = \Theta(n \log n)$$

• Сумма гармонического ряда

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

$$1 + \lfloor \log_2 n \rfloor \geq \frac{1}{1} + \overbrace{\frac{1}{2} + \frac{1}{2}}^1 + \overbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}}^1 + \overbrace{\frac{1}{8} + \dots}^{1\dots} \geq \sum_{k=1}^n \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \dots \geq$$

$$\frac{1}{1} + \underbrace{\frac{1}{2}}_{1/2} + \underbrace{\frac{1}{4} + \frac{1}{4}}_{1/2} + \underbrace{\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}}_{1/2} + \dots \geq 1 + \frac{1}{2} \lfloor \log_2 n \rfloor \Rightarrow \sum_{k=1}^n \frac{1}{k} = \Theta(\log n)$$

1.9. Сравнение асимптотик

- Def 1.9.1. *Линейная сложность*  $\mathcal{O}(n)$
- Def 1.9.2. *Квадратичная сложность*  $\mathcal{O}(n^2)$
- Def 1.9.3. *Полиномиальная сложность*  $\exists k > 0: \mathcal{O}(n^k)$
- Def 1.9.4. *Полилогарифм*  $\exists k > 0: \mathcal{O}(\log^k n)$
- Def 1.9.5. *Экспоненциальная сложность*  $\exists c > 0: \mathcal{O}(2^{cn})$

**Теорема 1.9.6.**  $\forall x, y > 0, z > 1 \exists N \forall n > N: \log^x n < n^y < z^n$

*Доказательство.* Сперва докажем первую часть неравенства через вторую.

Пусть  $\log n = k$ , тогда  $\log^x n < n^y \Leftrightarrow k^x < 2^{ky} = (2^y)^k = z^k \Leftrightarrow n^y < z^n$  ■

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  $C \leq n'$  (возьмём достаточно большое  $n$ ), тогда  $n^y < z^n \Leftrightarrow n < 2^{\frac{1}{y}n \log z} \Leftrightarrow C \cdot n' < 2^{n'} \Leftrightarrow (n')^2 < 2^{n'}$

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  $n^2 \rightarrow 4 \cdot n^2$ , а  $2^n \rightarrow 2^{2n} = 2^n \cdot 2^n \geq 4 \cdot 2^n$  при  $n \geq 2$ .

Значит  $\forall n \geq 2$  если для  $n$  верно, то и для  $2n$  верно.

Переход: [10..20)  $\rightarrow$  [20..40)  $\rightarrow$  [40..80)  $\rightarrow$  ... ■

*Следствие 1.9.7.*  $\forall x, y > 0, z > 1: \log^x n = \mathcal{O}(n^y), n^y = \mathcal{O}(z^n)$

*Доказательство.* Возьмём константу 1. ■

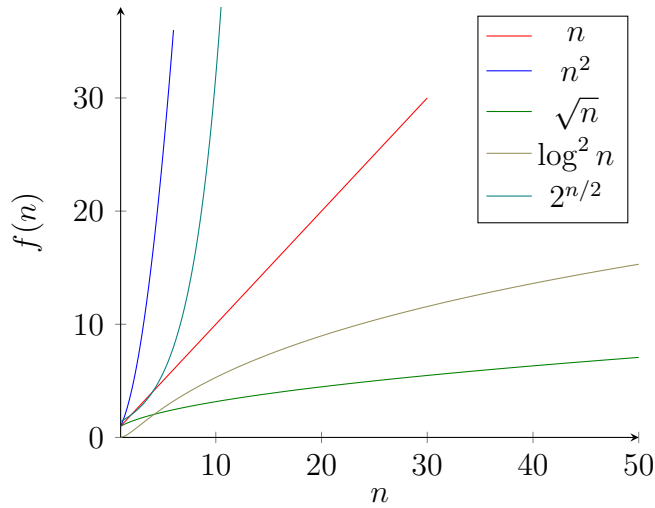
*Следствие 1.9.8.*  $\forall x, y > 0, z > 1: \log^x n = o(n^y), n^y = o(z^n)$

*Доказательство.* Достаточно перейти к чуть меньшим  $y, z$  и воспользоваться теоремой.

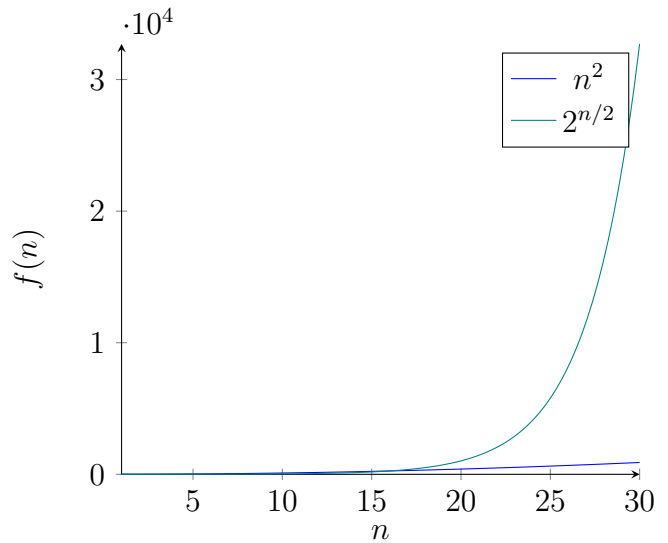
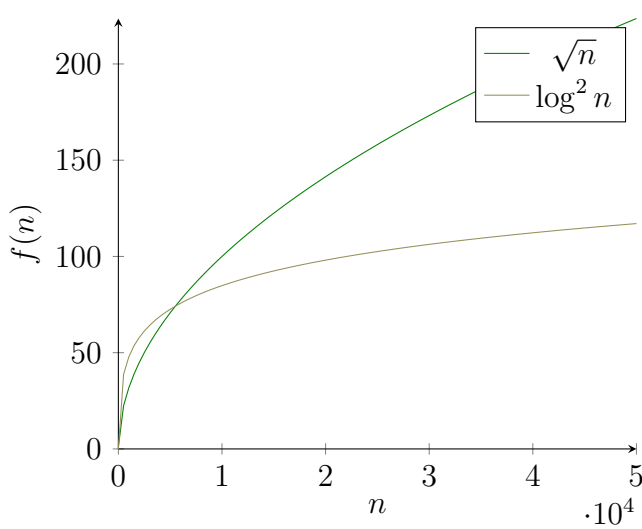
$$\exists N \forall n \geq N \log^x n < n^{y-\varepsilon} = \frac{1}{n^\varepsilon} n^y, \frac{1}{n^\varepsilon} \xrightarrow{n \rightarrow \infty} 0 \Rightarrow \log^x n = o(n^y).$$

$$\exists N \forall n \geq N n^y < (z - \varepsilon)^n = \frac{1}{(z/(z-\varepsilon))^n} z^n, \frac{1}{(z/(z-\varepsilon))^n} \xrightarrow{n \rightarrow \infty} 0 \Rightarrow n^y = o(z^n). \quad \blacksquare$$

• 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

10 сентября

### 2.1. C++

#### • Warnings

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

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

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

#### • Range check errors

Давайте рассмотрим стандартную багу: `int a[3]; a[3] = 7;`

В результате мы получаем *undefined behavior*. Код иногда падает по *runtime error*, иногда нет.

Чтобы такого не было, во-первых, используйте вектора, во-вторых, включите debug-режим.

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

Для пользователей linux есть более профессиональное решение: [valgrind](#).

#### • 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 переменную 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, указатель 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  $\geq 320$  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  $\mathcal{O}(n)$ 
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]; //  $[0..r] - [0..l)$ ,  $\mathcal{O}(1)$ 
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 };
```

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

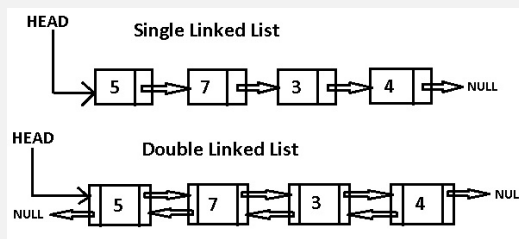
Указатель `tail` нужен, чтобы иметь возможность добавлять в конец, удалять из конца за  $\mathcal{O}(1)$ .

Ссылки `prev`, чтобы ходить по списку в обратном направлении, удалять из середины за  $\mathcal{O}(1)$ .

```

1 Node *find(List l, int x) { // найти в списке за линию
2     for (Node *p = l.head->next; p != l.tail; p = p->next)
3         if (p->x == x)
4             return p;
5     return 0;
6 }
7 Node *erase(Node *v) {
8     v->prev->next = v->next;
9     v->next->prev = v->prev;
10 }
11 Node *push_back(List &l, Node *v) {
12     Node *p = new Node();
13     p->x = x, p->prev = l.tail->prev, p->next = l.tail;
14     p->prev->next = p, p->next->prev = p;
15 }
16 void makeEmpty(List &l) { // создать новый пустой список
17     l.head = new Node(), l.tail = new Node();
18     l.head->next = l.tail, l.tail->prev = l.head;
19 }

```



## 2.6. Односвязный список

```

1 struct Node {
2     Node *next; // не храним ссылку назад, нельзя удалять из середины за O(1)
3     int x;
4 };
5 // 0 - пустой список
6 Node *head = 0; // не храним tail, нельзя добавлять в конец за O(1)
7 void push_front(Node* &head, int x) {
8     Node *p = new Node();
9     p->x = x, p->next = head, head = p;
10 }

```

## 2.7. Список на массиве

```

1 vector<Node> a; // массив всех Node-ов списка
2 struct {
3     int next, x;
4 };
5 int head = -1;
6 void push_front(int &head, int x) {
7     a.push_back(Node {head, x});
8     head = a.size() - 1;
9 }

```

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  $n \rightarrow 2n$  будет хотя бы  $\frac{n}{2}$  операций `push_back`, значит среднее время работы последней всех `push_back` между двумя удвоениями, включая последнее удвоение  $\mathcal{O}(1)$  ■

## 2.9. Стек, очередь, дек

Это названия интерфейсов (множеств функций, доступных пользователю)

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

Реализовывать все три структуры можно, как на списке так и на векторе.

Деку нужен двусвязный список, очереди и стеку хватит односвязного.

Вектор у нас умеет удваиваться только при `push_back`. Что делать при `push_front`?

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

### • Дек на циклическом векторе

```

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

## 2.10. Очередь, стек и дек с минимумом

В стеке можно поддерживать минимум.

Для этого по сути нужно поддерживать два стека – стек данных и стек минимумов.

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

`push(x)`: `a.push(x)`, `m.push(min(m.back(), x))`

Здесь `m` – “частичные минимумы”, стек минимумов.

- **Очередь с минимумом через два стека**

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

```
1 Stack a, b;
2 void push(int x) { b.push(x); }
3 int pop() {
4     if (a.empty()) // стек a закончился, пора перенести элементы b в a
5         while (b.size())
6             a.push(b.pop());
7     return a.pop();
8 }
9 int getMin() { return min(a.getMin(), b.getMin()); }
```

- **Очередь с минимумом через дек минимумов**

Будет разобрано на практике. См. разбор третьей практики.

- **Дек с минимумом через два стека**

Будет решено на практике. См. разбор третьей практики.

# Лекция #3: Структуры данных

17 сентября

## 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 элементарных кусков (операций),  $i$ -й из которых работает  $t_i$ .

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

$m = \frac{\sum_i t_i}{n}$  – average time (среднее время)

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

$\Delta\varphi_i = \varphi_{i+1} - \varphi_i$  = изменение функции  $\varphi$ , вызванное  $i$ -й операцией.

$a_i$  – время, амортизированное функцией  $\varphi$ .

Можно рассматривать  $\forall\varphi$ , интересно подобрать такую, чтобы  $t_i$  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  $m = \mathcal{O}(1)$

### • Строгие рассуждения.

**Lm 3.1.2.**  $\sum t_i = \sum a_i - (\varphi_{end} - \varphi_0)$

*Доказательство.* Сложили равенства  $a_i = t_i + (\varphi_{i+1} - \varphi_i)$  ■

**Теорема 3.1.3.**  $m = \mathcal{O}(\max a_i) + \frac{\varphi_0 - \varphi_{end}}{n}$

*Доказательство.* В лемме делим равенство на  $n$ ,  $\sum a_i/n \leq \max a_i$ , заменяем  $\frac{\sum_i t_i}{n}$  на  $m$  ■

*Следствие 3.1.4.* Если  $\varphi_0 = 0$ ,  $\forall i \varphi_i \geq 0$ , то  $m = \mathcal{O}(\max a_i)$

### • 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  $\varphi_0 = 0$ ,  $\varphi_{end} \geq 0$ .

### • 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  $\varphi_0 - \varphi_{end} \leq \sqrt{n} \stackrel{3.1.3}{\Rightarrow} m = \mathcal{O}(1)$ .

Теперь рассмотрим плохой потенциал  $\bar{\varphi} = y^2$ .

$$a_i = t_i + \Delta\bar{\varphi} = (y_{old} - y_{new} + 1) + (y_{new}^2 - y_{old}^2) = \mathcal{O}(1)$$

Но, при этом  $\varphi_0 = n$ ,  $\varphi_{end} = 0 \stackrel{3.1.3}{\Rightarrow} m = \mathcal{O}(\sqrt{n})$  = (

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  $size \rightarrow 2size$ , 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  $\varphi$  через её  $\Delta\varphi$ .  $\varphi$  – 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).

```

1 stack<int> value; // уже посчитанные значения
2 stack<char> op; // ещё не выполненные операции
3 void make() { // выполнить правую 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 скомпилился, но смысл такой
8 }
9 int eval(string s) { // s без пробелов
10    s = '(' + s + ')' // при выполнении последней ), выражение вычислится
11    for (char c : s)
12        if ('0' <= c && c <= '9') value.push(c - '0');
13        else if (c == '(') op.push(c);
14        else if (c == ')') {
15            while (op.top() != '(') make();
16            op.pop();
17        } else {
18            while (op.size() && priority(op.top()) >= priority(c)) make();
19            op.push(c);
20        }
21    return value.top();
22 }

```

**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. ■

• **Lowerbound.** Можно искать более сложную величину  $\min i: a_i \geq x$ .

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

Если все элементы  $[l, r)$  меньше  $x$ , `lower_bound` вернёт  $r$ . Время работы также  $\mathcal{O}(\log n)$ .

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

```
return a[lower_bound(l, r, x)] == x;
```

В языке C++ есть стандартные функции

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

Через них легко найти  $[\max i: a_i \leq x] = \text{upper\_bound} - 1$  и  $[\max i: a_i < x] = \text{lower\_bound} - 1$ .

#### 3.3.2. По предикату

Можно написать ещё более общий бинпоиск, при этом сделать код более простым. Предикат – функция, принимающая значения только 0 и 1. Пусть предикат  $f(x) = (x < i ? 0 : 1)$ .

Тогда мы бинпоиском можем найти такие  $l + 1 = r$ , что  $f(l) = 0, f(r) = 1$ .

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

Как это использовать для решения задачи `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

```

### 3.3.3. 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  $\varepsilon$ .

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

```

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

```

Теперь собственно поиск корня:

```

1 while (r - l > ε)
2     double m = (l + r) / 2;
3     (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 бинарным поиском 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 применения «двух указателей» для поиска пересечения.

**Вариант #1, for:**

```

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

```

**Вариант #2, while:**

```

1 int i = 0, j = 0;
2 while (i < |A| && j < |B|)
3     if (A[i] == B[j]) C[k++] = A[i++], j++;
4     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); } //  $\mathcal{O}(1)$  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 +  $\mathcal{O}(1)$ 
```

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  $n \leq N$  и равномерном распределении элементов, все операции работают за  $\mathcal{O}(1)$ . Как сделать распределение равномерным? Подобрать хорошую хеш-функцию!

*Утверждение 3.5.1.* Если  $N$  простое, то хеш-функция  $x \rightarrow x \% N$  достаточно хорошая.

Без доказательства.

Если добавлять в хеш-таблицу новые элементы, со временем  $n$  станет больше  $N$ .

В этот момент нужно перевыделить память  $N \rightarrow 2N$  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;`

**Лм 3.5.2.** Если в хеш-таблице с открытой адресацией размера  $N$  занято  $\alpha N$  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  $\alpha$ .

Вероятность того, что мы не остановимся и после  $k$  шагов равна  $\alpha^k$ , то есть, сделаем  $k$ -й шаг (не ровно  $k$  шагов, а именно  $k$ -й!). Время работы = матожидание числа шагов =  $1 + \sum_{k=1}^{\infty} (\text{вероятность того, что мы сделали } k\text{-й шаг}) = 1 + \alpha + \alpha^2 + \alpha^3 + \dots = \frac{1}{1-\alpha}$ . ■

### • Переполнение хеш-таблицы

При слишком **большом**  $\alpha$  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: открытая 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 пользоваться, 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 3.5.3.* Чтобы работало всегда, придётся выделять память со случайным запасом:

```
unordered_map<int, int> h(N + randomTime() % N),
```

чтобы ушлые люди не могли подобрать к вашей программ анти-хеш теста.

## Лекция #4: Структуры данных

24 сентября

### 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    }
12    b[n++] = x;
13 }
```

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 ( $i \leq pos$  ? `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 будем 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();
8 }
```

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 }
6 int pop() {
7     if (копирование завершено)
8         a = a1, b = b1, начать новое копирование;
9     STEP; // сделать несколько шагов копирования
10    return a.pop();
11 }
```

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 поддерживать инвариант, что до начала копирования  $|a| \geq |b|$ . В каждом `pop` будем делать 1 шаг копирования, в каждом `push` также 1 шаг. Проверка инварианта после серии `push`: за  $k$  пушей мы сделали  $\geq k$  копирований, поэтому  $|a_1| \geq |b_1|$ . Проверка корректности `pop`: после первых  $|b|$  операций `pop` 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()`. Извлечение (удаление) минимума.

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

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

### 4.2.1. 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 }
5 void siftDown(int i) {
6     while (1) {
7         int l = 2 * i;
8         if (l + 1 <= n && a[l + 1] < a[l]) l++; // выбрать меньшего из детей
9         if (!(l <= n && a[l] < a[i])) break; // если все дети не меньше нас, это конец
10        swap(a[l], a[i]), i = l; // перейти в ребёнка
11    }
12 }

```

**Lm 4.2.3.** Обе процедуры корректны

*Доказательство.* По индукции на примере `siftUp`. В каждый момент времени верно, что дерево  $i$  – 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 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     h[i] = h[n--], pos[h[i]] = i; // не забыли обновить pos
4     siftUp(i), siftDown(i); // новый элемент может быть и меньше, и больше
5 }

```

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: `a[i]=x`, `getMin(a)`, `a.push_back(x)`, `extractMin`.

*Решение:* «`push_back`» = `add`, «`a[i]=x`» = `del(i)`, поменять `a[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);
4 }

```

**Lm 4.2.6.** Функция `Build` построит корректную бинарную кучу.

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

**Lm 4.2.7.** Время работы функции `Build`  $\Theta(n)$



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

время работы  $\sum_{i=1}^k 2^{k-i} = 2^k \sum_{i=1}^k \frac{i}{2^i} \stackrel{(*)}{=} 2^k \cdot \Theta(1) = \Theta(n)$ . (\*) доказано на практике. ■

```

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

```

**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. Должно 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.

```

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

```

А как удалять? Произвольную нельзя, а последнюю – просто уменьшить `pos`.

**В C++ при вызове функции**, при создании локальных переменных используется ровно такая же модель аллокации памяти, 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  $x = \text{CONST}$ , все выделяемые ячейки одного размера.

**Идея:** разобьём всю память на куски по `x` байт, свободные куски образуют список...

Пусть наше адресуемое пространство 32-битное, то есть,  $\text{MAX\_MEM} \leq 2^{32}$ . Тогда давайте исходные `MAX_MEM` байт памяти разобьём на 4 байта `head` и на  $k = \lfloor \frac{\text{MAX\_MEM}-4}{\max(x,4)} \rfloor$  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.

```

1  const uint32_t size; // размер блока, size >= 4
2  uint32_t head = 0; // указатель на первый свободный блок
3  uint8_t mem[MAX_MEM-4]; // часть памяти, которой пользуется new
4  uint32_t* pointer(uint32_t i) { return (uint32_t*)(mem+i); } // magic =)
5  void init() {
6      for (uint32_t i = 0; i + size <= MAX_MEM-4; i += size)
7          *pointer(i) = i + size; // указываем на следующий блок
8  }
9  uint32_t new() { // вернёт адрес в нашем 32-битном пространстве mem
10     uint32_t res = head;
11     head = *pointer(head); // следующий свободный блок
12     return res;
13 }
14 void delete(uint32_t x) {
15     *pointer(x) = head; // записали в ячейки [x..x+4) старый head
16     head = x;
17 }

```

### 4.3.3. Куча

`MAX_MEM` байт памяти разобьются на отрезки свободной памяти и отрезки занятой памяти. 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  $\mathcal{O}(1) + \langle \text{время просеивания вниз в куче} \rangle = \mathcal{O}(\log n)$ .

- **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.

- **Полная версия (32-bit)**

Храним кучу свободных кусков. Пусть все свободные куски имеют размер хотя бы 12 байт.

- Память = служебная информация + пользовательская память.
- Первые  $4 + n \cdot 8$  байт = кол-во элементов в куче + собственно куча.  
Каждая ячейка кучи – пара (размер, указатель). В корне максимум.  
Далее идёт пользовательская память, которая и будет аллоцироваться.
- `new(size)` : смотрим корень, если `heap[1] >= size`, возвращаем `size` последних байт, уменьшаем `heap[1]` на `size`, вызываем `siftDown`.
- `delete(addr, size)`. Если соседи – не свободные куски, добавим новый элемент в кучу.  
Для этого расширим: кучу, откусим 8 байт смежного с кучей свободного куска. Если соседи – пустые куски, объединим нас и их в один большой кусок, сделаем `siftUp` в куче.

**Как выделять память под кучу?** Можно заранее фиксировать  $N$  и выделить  $8N$  байт памяти. Можно надеяться, что смежный с кучей кусок свободен, и при `n++` отщипывать от него очередные 8 байт. 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?**

Давайте поддерживать, что каждый свободный кусок устроен так: первые 4 байта = указатель на конец куска, вторые 4 байта = обратная ссылка (индекс куска в куче), последние 4 байта = указатель на начало куска. Смотрим на соседей, пытаемся их интерпретировать как свободные, за  $\mathcal{O}(1)$  проверяем, что место, на которое они указали в куче хранит именно их.

**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();
8 }

```

Время работы DelMin осталось тем же, стало амортизированным.

В худшем случае все DelMin в сумме работают  $\Theta(n \log n)$ .

Зачем это нужно? Например, `std::priority_queue`.

**4.4.2. Поиск → Удаление**

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

**4.4.3. Add → Merge**

Merge (слияние) – операция, получающая на вход две структуры данных, на выход даёт одну, равную их объединению. Старые структуры объявляются невалидными.

**Пример #1.** Merge двух сортированных массивов.

**Пример #2.** Merge двух куч. Сейчас мы научимся делать его быстро.

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

```

1 Heap Merge(Heap a, Heap b) {
2     if (a.size < b.size) swap(a, b);
3     for (int x : b) a.Add(x);
4     return a;
5 }

```

**Лм 4.4.1.** Если мы начинаем с  $\emptyset$  и делаем  $N$  произвольных операций из множества  $\{\text{Add},$

Merge}, функция Add вызовется не более  $N \log_2 N$  раз.

*Доказательство.* Посмотрим на код и заметим, что  $|a| + |b| \geq 2|b|$ , 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 $\rightarrow$ 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 бинарных поиска за  $\mathcal{O}(\log n)$

##### • Решение #1. Корневая.

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

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  $\mathcal{O}(\text{Build}(n) + \sqrt{n} \cdot \text{Build}(\sqrt{n})) = \mathcal{O}(\text{Build}(n))$  (для выпуклых функций Build)  $\Rightarrow$  среднее время работы Add =  $\mathcal{O}(\text{Build}(n)/\sqrt{n}) = \mathcal{O}(\sqrt{n} \log n)$ .

##### • 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  $a_1 a_2 \dots a_k$ . Для хранения  $N = 2^{a_1} + 2^{a_2} + \dots + 2^{a_k}$  элементов будем хранить  $k$  структур S из  $2^{a_1}, 2^{a_2}, \dots, 2^{a_k}$  элементов.  $k \leq \log_2 n$ . Новый Get работает за  $k \cdot \text{S.Get}$ , 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(a1)`, `Build(a2)`, ..., `Build(ak)`. Тогда  $\sum_{i=1}^k a_i \leq n \log_2 n$

*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  $\text{Build}(n \log_2 n)$

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

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

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

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

### 4.4.5. Build $\rightarrow$ Add, Del

Научим «пополняемый массив обрабатывать запросы».

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

Для этого будем хранить два «пополняемых массива» – добавленные элементы, удалённые 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

1 октября

### 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  $l_i \leq l_{i+1}, r_i \leq r_{i+1}$  – 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  $\sum_i (|l_{i+1} - l_i| + |r_{i+1} - r_i|) \rightarrow \min$ . 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  $|l_{i+1} - l_i| \leq k$ , при переходе между группами (движение только вперёд)  $\sum |l_{i+1} - l_i| \leq 2n$ . Итого  $mk + 2n$  шагов  $l_i$ . Посмотрим, как меняется  $r_i$ . Внутри группы указатель  $r$  сделает в сумме  $\leq n$  шагов вперёд, при переходе между группами сделает  $\leq n$  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  $k$   $f = mk \nearrow$ ,  $g = \frac{n}{k}n \searrow \Rightarrow$  оптимально взять  $k$ :  $mk = \frac{n}{k}n \Rightarrow k = (n^2/m)^{1/2} = n/m^{1/2} \Rightarrow$  время работы  $mk + \frac{n}{k}n = n\sqrt{m} + n\sqrt{m} \Rightarrow$  общее время работы  $\Theta(m + n\sqrt{m})$ .

## 5.2. Квадратичные сортировки

**Def 5.2.1.** Сортировка называется стабильной, если одинаковые элементы она оставляет в исходном порядке.

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

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

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

**Lm 5.2.4.** Массив отсортирован  $\Leftrightarrow I = 0$

- **Selection sort** (сортировка выбором)

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

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

- **Insertion sort** (сортировка вставками)

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

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

Корректность: по индукции по  $i$  ■

Заметим, что можно ускорить сортировку, место для вставки искать бинарным поиском.

Сортировка всё равно останется квадратичной.

- **Bubble sort** (сортировка пузырьком)

Бесполезна. Изучается, как дань истории. Простая.

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

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

- **Сравним пройденные сортировки.**

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

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  $I$ ).

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

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

**Lm 5.3.1.** 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 – меньше (0) или больше (1). Сортировка получает  $k$  бит информации, и результат её работы зависит только от этих  $k$  бит. То есть, если для двух перестановок она получит одни и те же  $k$  бит, одну из этих двух перестановок она отсортирует неправильно. Сортировка корректна  $\Rightarrow 2^k \geq n!$ .

$2^k < n! \Rightarrow$  сортировка не корректна. Осталось вспомнить, что  $\log(n!) = \Theta(n \log n)$ . ■

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

### 5.4. Решение задачи по пройденным темам

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

**Решения:**

1. Отсортировать первый массив, бинарным поиском 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  $\Theta(n)$  допамяти, 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++) //  $\Theta(n)$ 
3     count[x]++; // насчитали, сколько раз x встречается в a
4 for (int x = 0; x < m; x++) //  $\Theta(m)$ , перебрали x в порядке возрастания
5     while (count[x]--)
6         out(x);

```



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

### 5.5.2. MergeSort (слиянием)

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

```

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

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

**Lm 5.5.1.** 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); //  $\mathcal{O}(1)$ 
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 деления 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--]);
7     }
8 }

```

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

*Замечание 5.5.3.*  $a[l, j] \leq x, a(j, i) = x, a[i, r] \geq x$

*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  $l < i \geq j < r$ . Более того  $a[l] \leq x, a[r] \geq x \Rightarrow$  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
7 }

```

### 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	$\mathcal{O}(n \log n)$	$\Theta(\log n)$	-

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

Такая сортировка существует. Она получается на основе MergeSort и merge за  $\mathcal{O}(n)$  без дополнительной памяти. На практике мы научимся делать inplace stable merge за  $\mathcal{O}(n \log n)$ .

## 5.6. (\*) Adaptive Heap sort

Цель данного блока, предъявить 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:  $x = C.\text{extractMin}$ , 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-a) 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  $(x_i=i, y_i=a_i)$ . Э 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  $\sum(k_i - 1) \leq I$ , 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 отсортированном порядке стоят вместе и в таком же порядке. Если наш массив можно представить в виде конкатенации  $b$  блоков, то время сортировки  $\mathcal{O}(n + b \log b)$ .

### 5.6.3. (\*) Ссылки

[AdaptiveHeapSort]. Levkopoulos and Petersson'92, там же доказаны все теоремы.

```

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

```

## 5.7. (\*) Kirkpatrick'84 sort

Научимся сортировать  $n$  целых чисел из промежутка  $[0, C)$  за  $\mathcal{O}(n \log \log C)$ .

Пусть  $2^{2^{k-1}} < C \leq 2^{2^k}$ , 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 #6: Сортировки (продолжение)

8 октября

### 6.1. Quick Sort

#### • Глубина

Можно делать не два рекурсивных вызова, а только один, от меньшей части.

Тогда в худшем случае допямать = глубина =  $\mathcal{O}(\log n)$ .

Вместо второго вызова  $(l_2, r_2)$  сделаем  $l = l_2$ ,  $r = r_2$ , `goto start`.

#### • Выбор $x$

На практике и в дз мы показали, что при любом детерминированном 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  $x = a[\text{random } l..r]$ . 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 #1.

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

*Доказательство.* Докажем по индукции.

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

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

Воспользуемся индукционным предположением (индукция же!)

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

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

$$\sum_{i=0}^{n-1} (i \ln i) \leq \int_1^n i \ln i \, di$$

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

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

Какое же  $C$  выбрать? Мы хотим  $F \leq Cn \ln n$ , 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 не более одного раза.

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

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

**Lm 6.1.2.**  $Pr[\text{сравнения}(i, j)] = \frac{2}{j-i+1}$  при  $i < j$

*Доказательство.* Сравниться  $i$  и  $j$  могут только, если при некотором `Partition` выбор пал на один из них. Рассмотрим дерево рекурсии. Посмотрим на самую глубокую вершину,  $[l, r]$  всё ещё содержит  $i$ -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  $\Rightarrow$  `Partition` выберет один из  $j-i+1$  элементов отрезка  $[i, j]$ . С вероятностью  $\frac{2}{j-i+1}$  он совпадёт с  $i$  или  $j$ , тогда и только тогда  $i$  и  $j$  сравнятся. ■

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

### 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 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;
4   Partition(l, r, x = a[random [l, r]], i, j);
5   if (j < k && k < i) return x;
6   return k <= j ? Statistic(l, j, a) : Statistic(i, r, a);
```

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

**Теорема 6.2.2.** Время работы 6.2.1 равно  $\Theta(n)$

*Доказательство.* С вероятностью  $\frac{1}{3}$  мы попадем в элемент, который лежит во второй трети отсортированного 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:

$$T(n) = n + \frac{1}{3}T\left(\frac{2}{3}n\right) + \frac{2}{3}T(n) \Rightarrow T(n) = 3n + T\left(\frac{2}{3}n\right) \leq 9n = \Theta(n) \quad \blacksquare$$

*Замечание 6.2.3.* Мы могли бы повторить доказательство 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, это и есть  $x$ .

*Утверждение 6.2.4.* На массиве длины  $n$  медиану можно выбрать за  $6n$  сравнений.

Поскольку из  $\frac{n}{5}$  меньше  $\frac{n}{10}$  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:

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

### 6.2.3. C++

В C++: STL есть следующие функции

1. `nth_element(a, a + k, a + n)` –  $k$ -я статистика на основе однократного 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  $k: x \rightarrow \lfloor \frac{x}{k} \rfloor$  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. Count sort

Давайте используем уже известный нам Count Sort, 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 допамать!
```

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

### 6.3.2. Radix sort

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

- **Алгоритм:** отсортируем сперва по последнему символу, затем по предпоследнему и т.д.
- **Корректность:** мы сортируем стабильной сортировкой строки по символу номер  $i$ , строки уже отсортированы по символам  $(i, L]$ . Из стабильности имеем, что строки равные по  $i$ -му символу будут отсортированы как раз по  $(i, L] \Rightarrow$  теперь строки отсортированы по  $[i, L]$ .
- **Время работы:**  $L$  раз вызвали сортировку подсчётом  $\Rightarrow$  сортируем строки  $\Theta(L(n + k))$ .

Теперь заметим, что  $\forall k$  число из  $[0, m)$  – строка длины  $\log_k m$  над алфавитом  $[0, k) \Rightarrow$  умеем сортировать числа за  $\Theta((n + k) \lceil \log_k m \rceil)$ . При  $k = n$  получаем оптимум  $n \lceil \log_n m \rceil$ .

- **Выбор системы счисления:**

$\text{Time} = \Theta((n + k) \lceil \log_k m \rceil) = \Theta(\max(n, k) \frac{\log m}{\log k})$ .

При  $k \leq n$  это  $\Theta(n \frac{\log m}{\log k})$ , min достигается при  $k = n$ .

При  $k \geq n$  это  $\Theta(k \frac{\log m}{\log k})$ , min достигается при  $k = n$ .



### 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:

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

Каждое число  $x_j$  попадает в отрезок номер  $i_j = \lfloor \frac{x_j - \min}{\max - \min + 1} n \rfloor$ . 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 В.І.), чтобы минимизировать константу, или рекурсивным вызовом Bucket Sort (алгоритм В.В.)

#### 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.**  $\max - \min \leq n \Rightarrow$  и ВВ, и ВІ работают за  $\Theta(n)$

*Доказательство.* В каждом рекурсивном вызове  $\max - \min \leq 1$  ■

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

*Доказательство.* Ветвление происходит при  $n \geq 2 \Rightarrow$  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 ВІ =  $\Theta(n)$

*Доказательство.* Время работы ВІ:  $T(n) = \frac{1}{R} \sum_{\text{random}} (\sum_i k_i^2)$ , где  $k_i$  – размер  $i$ -го бакета.

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

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

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 ВІ 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.  $C \leq 2$
4. Мы храним хотя бы два числа,  $C > 2$ .

Первые три вы разберёте самостоятельно, здесь же детально описан **только 4-й случай**.

Пусть  $2^{2^{k-1}} < C \leq 2^{2^k}$ , 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  $k \geq 1$ ,  $\text{Heap}\langle k \rangle$ , 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     unordered_map<int, Heap<k-1>*> parts; // собственно куски
5 };

```

### • Как добавить новый элемент?

Номер куска по числу  $x$ :  $\text{index}(x) = (x \gg 2^{k-1})$ ;

```

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 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  $m \leq k$  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)` ... для всех  $m-1$  пар соседних кусков.

3. Сортируем элементы  $z$  за квадрат.

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

Нестабильность появляется в сортировке выбором и перемешивании элементов буфера  $z$ .

## 6.6. (\*) 3D Мо

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

В Offline даны массив длины  $n$  и  $q$  запросов двух типов:

- `get(li, ri)` – запрос на отрезке
- `a[ki] = xi` – 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  $l * \text{число блоков по } i$ )

$\sum \Delta r = 2n \frac{q}{xy}$  (движемся по возрастания \* на число блоков)

$T = \Theta(\max(\frac{n^2q}{xy}, q(x+y)))$ , возьмём  $x, y: \frac{n^2q}{xy} = q(x+y) \Leftrightarrow n^2 = xy(x+y) \Rightarrow x=y=\Theta(n^{2/3})$ ,  $T = qn^{2/3}$

### • 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.

### 6.6.1. (\*) Применяем для mex

**Def 6.6.1.**  $mex(A) = \min(\mathbb{N} \cup \{0\} \setminus A)$ .

Пример  $mex(\{1,2,3\}) = 0$ ,  $mex(\{0,1,1,4\}) = 2$ .

Задача – *mex* на отрезке меняющегося массива в offline.

Идея выше позволяет решить за  $\mathcal{O}(qn^{2/3} \log n)$ : кроме `unordered_map<int, int> count` нужно ещё поддерживать кучу (`set`)  $\{x: count[x] = 0\}$ .

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

### • Корневая для минимума

Хотим хранить диапазон целых чисел  $[0, C)$ .

Разобьём диапазон на  $\sqrt{C}$  кусков. В каждом хотим хранить количество чисел.

`add/del` числа  $x$  – обновление счётчиков `count[x]` и `sum_in_block[ $\lfloor \frac{x}{\sqrt{C}} \rfloor$ ]`.

`get` – найти перебором блок  $i: \text{sum\_in\_block}[i] > 0$  и внутри блока  $x: \text{count}[x] > 0$ .

## Лекция #7: Кучи

15 октября

## 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.

**Лм 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

$$x_i \leq H(n)$$

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

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

Рассмотрим случай полного бинарного дерева  $n = 2^k - 1 \Rightarrow$

$$\sum \log size_i = (\log 3) \frac{n+1}{4} + (\log 7) \frac{n+1}{8} + (\log 15) \frac{n+1}{16} + \dots = (n+1) \left( \frac{\log 3}{4} + \frac{\log 7}{8} + \frac{\log 15}{16} + \dots \right).$$

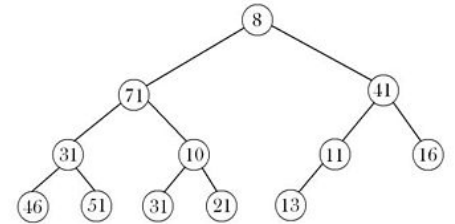
При  $n \rightarrow +\infty$  величина  $\frac{\sum \log size_i}{n+1}$  имеет предел, вычисление первых 20 слагаемых даёт 1.36442... и ошибку в 5-м знаке после запятой. ■

## 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  $i \rightarrow 2i, 2i + 1$ . 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  $v$  меньше, чем у отца, тогда делаем обычный `SiftUp` с шагом  $i \rightarrow \frac{i}{4}$ .
2. Иначе меняем местами  $v$  и  $\frac{v}{2}$ , и из  $\frac{v}{2}$  делаем обычный `SiftUp` с шагом  $i \rightarrow \frac{i}{4}$ .

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

**Lm 7.3.2.**  $d(v) \leq \log_2(\text{size} + 1)$

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

**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:  $\forall v \quad d(v.\text{left}) \geq d(v.\text{right})$

*Следствие 7.3.4.* В левацкой куче  $\log_2 n \geq d(v) = d(v.\text{right}) + 1$

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  $a \rightarrow x$ ,  $b \rightarrow x$  будет новым корнем.

Пусть это  $a \rightarrow x$ , тогда сделаем  $a \rightarrow r = \text{Merge}(a \rightarrow r, b)$  (рекурсия). Конец =)

Для удобства реализации EMPTY – пустое дерево, у которого  $l = r = \text{EMPTY}$ ,  $x = +\infty$ ,  $d = 0$ .

```

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;
9 }
```

Время работы: на каждом шаге рекурсии величина  $a \rightarrow d + b \rightarrow d$  уменьшается  $\Rightarrow$  глубина рекурсии  $\leq a \rightarrow d + b \rightarrow d \leq 2 \log_2 n$ .

### 7.4. Skew Heap (Tarjan'86)

Уберём условие  $d(v.\text{left}) \geq d(v.\text{right})$ . 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,  $\text{size}$  – размер поддерева. Ребро  $p \rightarrow v$  называется

Тяжёлым, если  $\text{size}(v) > \frac{1}{2} \text{size}(p)$

Лёгким, если  $\text{size}(v) \leq \frac{1}{2} \text{size}(p)$

**Def 7.4.2.** Ребро  $parent \rightarrow v$  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 амортизировать количество тяжёлых рёбер. Введём потенциал  $\varphi =$  “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**.

$$t_i = L_i + T_i \leq \log_2 n + T_i$$

Теперь распишем изменения потенциала  $\varphi$ . 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.

$$\Delta\varphi \leq L_i - T_i \leq \log_2 n - T_i \Rightarrow a_i = t_i + \Delta\varphi \leq 2\log n$$

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

## 7.5. Спско-куча

### • Мотивация.

Куча должна уметь Add, Min, Merge, DecreaseKey, ExtractMin.

Обычная бинарная умеет всё кроме Merge за  $\log$ , Merge за  $\log^2$ . Сегодня изучили Leftist и Skew, которые умеют всё за  $\log$ . Наша конечная цель – куча Фибоначчи, которая всё кроме ExtractMin умеет за  $\mathcal{O}(1)$ , ExtractMin за  $\mathcal{O}(\log n)$ . Спско-куча – важный шаг к нашей цели.

Будем хранить элементы кучи в двусвязном списке и поддерживать указатель на текущий минимум. Заметим, что операции Merge, Add, DecreaseKey, GetMin в списке работают за  $\mathcal{O}(1)$ .

Операция ExtractMin состоит из GetMin за  $\mathcal{O}(1)$ , удаления из списка за  $\mathcal{O}(1)$  и самой сложной части «найти новый минимум», которую мы пока умеем делать лишь за  $\mathcal{O}(n)$ .

P.S. Хочется линейный проход по всем элементам амортизировать уменьшением длины списка. Это мы сделаем в следующей домашке. На экзамене в конце первого модуля это уметь не нужно.



## 7.6. (\*) Pairing Heap

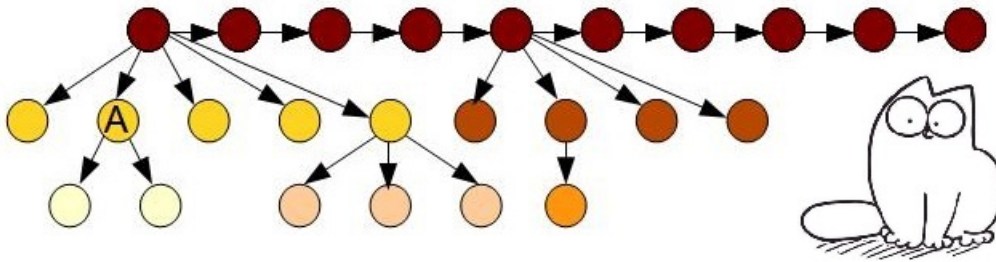
Сейчас мы из списко-кучи получим чудо-структуру, которая умеет амортизированно Add, Merge за  $\mathcal{O}(1)$ , ExtractMin за  $\mathcal{O}(\log n)$ .

DecreaseKey за  $\mathcal{O}(\log n)$ , сколько точно, никто не знает.

Идея, как ExtractMin в списко-куче амортизировать, – давайте, после прохода по списку длины  $k$  элементы списка разобьём 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.

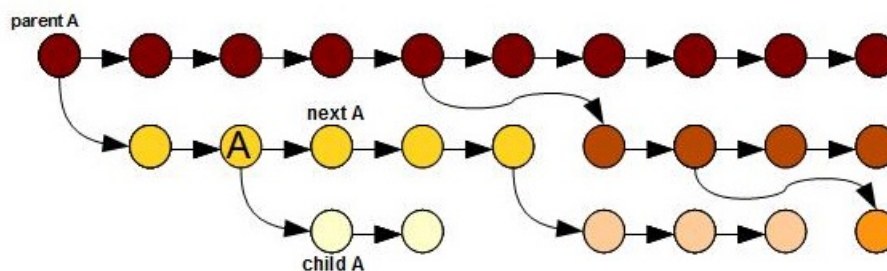


Детей вершины мы храним двусвязным списком. Будем хранить указатель на первого ребенка, для каждого ребенка указатели на соседей в списке и отца. Итого:

```

1 struct PairingHeap {
2     int x;
3     PairingHeap *child, *next, *prev, *parent;
4 };
5 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:

```

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

```

Основная операция Pair, она же Merge, – создать из двух куч одну. Выбирает кучу с меньшим ключом и 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;
```

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. Также пишется Delete  $\forall$  не минимума.

Осталась самая сложная часть – ExtractMin.

Чтобы найти новый минимум нужно пройти по всем детям. Пусть их  $k$ .

```

1 def ExtractMin:
2   root = Pairing(root->child)
3 def Pairing(a): # для простоты синтаксиса, пусть наши списки питоно-подобны
4   # a - список куч
5   if |a| = 0 return Empty
6   if |a| = 1 return a[0]
7   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, DecreaseKey. 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.<sup>[2]</sup> 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.<sup>[5]</sup>

[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 Фибоначчи.

Оценки времени работы DecreaseKey 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
<b>Binary</b> <sup>[11]</sup>	$\Theta(1)$	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(n)$
<b>Leftist</b>	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\Theta(\log n)$
<b>Binomial</b> <sup>[11][12]</sup>	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)^{[a]}$	$\Theta(\log n)$	$\mathcal{O}(\log n)^{[b]}$
<b>Fibonacci</b> <sup>[11][13]</sup>	$\Theta(1)$	$\mathcal{O}(\log n)^{[a]}$	$\Theta(1)$	$\Theta(1)^{[a]}$	$\Theta(1)$
<b>Pairing</b> <sup>[3]</sup>	$\Theta(1)$	$\mathcal{O}(\log n)^{[a]}$	$\Theta(1)$	$\mathcal{O}(\log n)^{[a][c]}$	$\Theta(1)$
<b>Brodal</b> <sup>[16][d]</sup>	$\Theta(1)$	$\mathcal{O}(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
<b>Rank-pairing</b> <sup>[18]</sup>	$\Theta(1)$	$\mathcal{O}(\log n)^{[a]}$	$\Theta(1)$	$\Theta(1)^{[a]}$	$\Theta(1)$

## Лекция #8: Кучи

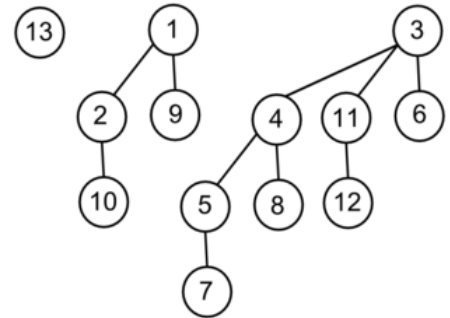
12 ноября

## 8.1. Биномиальная куча (Vuillemin'78)

## 8.1.1. Основные понятия

Определим понятие «биномиальное дерево» рекурсивно.

**Def 8.1.1.** 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 8.1.2.**  $|T_n| = 2^n$

*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 8.1.3.**  $\text{degRoot}(T_n) = n$

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

**Lm 8.1.4.** Сыновьями  $T_n$  являются деревья  $T_0, T_1, \dots, T_{n-1}$ .

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

**Lm 8.1.5.**  $\text{depth}(T_n) = n$

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

• 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  $v$ :  $v \rightarrow \text{child}$ ,  $v \rightarrow \text{child} \rightarrow \text{next}$ ,  $v \rightarrow \text{child} \rightarrow \text{next} \rightarrow \text{next}$ , ...

Теперь определим понятие «биномиальная куча».

**Def 8.1.6.** 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.

### 8.1.2. 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 8.1.5 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   return result;

```

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

### 8.1.3. 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` амортизируется потенциалом  $\varphi = \text{Roots}$  (длина списка, количество корней).

**Теорема 8.1.7.** Среднее время работы `ExtractMin` равно  $\mathcal{O}(\log n)$

*Доказательство.*  $t_i = \text{Roots} + \text{maxRank}$ ,  $\Delta\varphi \leq \log_2 n - \text{Roots} \Rightarrow a_i = t_i + \Delta\varphi = \mathcal{O}(\log n)$ .

Заметим также,  $0 \leq \varphi \leq n \Rightarrow$  среднее время `ExtractMin`  $\mathcal{O}(\log n)$ . ■

## 8.2. 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 имеют, 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 8.2.1.** Ранг вершины в Фибоначчиевой куче – её степень на тот момент, когда вершина последний раз была корнем.

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

**Lm 8.2.2.**  $v.\text{rank} = v.\text{degree} + v.\text{mark}$

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 8.2.3.** `DecreaseKey` работает в среднем за  $\mathcal{O}(1)$

*Доказательство.* `Marked` – число помеченных вершин. Пусть  $\varphi = \text{Roots} + 2\text{Marked}$ .

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

### 8.2.1. Фибоначчиевы деревья

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

**Def 8.2.4.** 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 8.2.5.**  $\forall n \geq 2: S_n = 1 + S_0 + S_1 + \dots + S_{n-2}$

*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 8.2.6.**  $\forall n \geq 2: Fib_n = 1 + Fib_0 + Fib_1 + \dots + Fib_{n-2}$

*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 8.2.7.**  $S_n = Fib_n$

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

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

### 8.2.2. 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  $x_i, i = 0..k-1, rank(x_i) \leq rank(x_{i+1})$ .

Будем параллельно по индукции доказывать два факта:

1. Размер поддерева ранга  $k$  не меньше  $S_k$ .
2.  $\forall$  корня  $v \quad rank(x_i) \geq i$ .

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.  $x'_i \geq x_i \geq i$  ■

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  $\forall i = 0..k \quad rank(x_i) \geq i$ . ■

# Лекция #9: Динамическое программирование

19 ноября

## 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);
8 }

```

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

```

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

```

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.  $x \rightarrow x + 1, x + 7, 2x, 3x$  – переходы динамики

Исходная задача – посчитать  $f[n]$ .

Чтобы её решить, мы сводим её к подзадачам такого же вида меньшего размера – посчитать



для всех  $1 \leq i < n$ , 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.

```

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

```

Для данной задачи код получился немного проще (убрали if-ы).

В общем случае нужно помнить про оба способа, выбрать более удобный.

Суть не поменялась: для каждого  $x$  будет верно  $f[x] = \min(f[x-1], f[x-7], f[\frac{x}{2}], f[\frac{x}{3}])$ .

- **Интуиция для динамики вперёд и назад.**

*Назад:* посчитали  $f[x]$  через уже посчитанные подзадачи.

*Вперёд:* если  $f[x]$  верно посчитано, мы можем обновить ответы для  $f[x+1], f[x+7], \dots$

### 9.1.4. Ленивая динамика

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

```

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

```

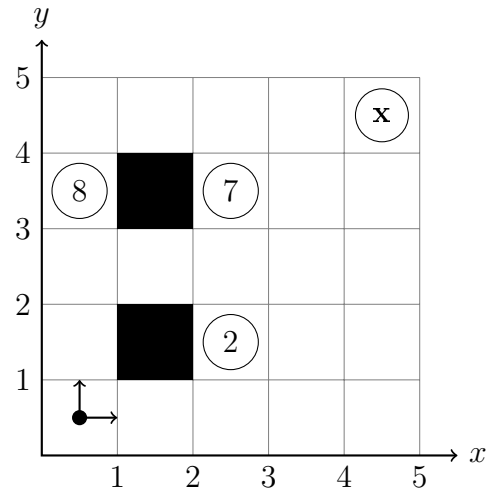
Для данной задачи этот код будет работать дольше, чем обычная «динамика назад циклом for», так как переберёт те же состояния с большей константой (рекурсия хуже цикла).

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

## 9.2. Ещё один пример

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

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



Решим задачу динамикой назад:

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

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

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

Решим задачу динамикой вперёд:

```

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, пройдем по обратным ссылкам от конца пути до его начала:

```

1 void outputPath() {
2     for (int x = width-1, y = height-1; !(x == 0 && y == 0); p[x,y] ? y-- : x--)
3         print(x, y);
4 }

```

- **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  $f[x, y]$ :

```

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

```

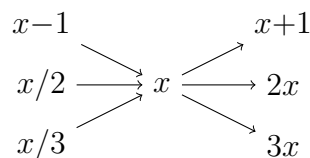
Второму способу нужно меньше памяти, но обычно он требует больше строк кода.

- **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 отличие 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 ( $a \rightarrow b$  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 ациклический граф.

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

*Утверждение 9.4.2.* Для любого ациклического 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 ациклическом?

Пусть есть ориентированный цикл  $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_k \rightarrow a_1$ . 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 ациклического 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  $f[\text{width}-1, \text{height}-1]$
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  $a_0, a_1, \dots, a_{n-1}$ .

Требуется выбрать подмножество предметов суммарного веса ровно  $S$ .

1. Задача NP-трудна, если решить её за  $\mathcal{O}(\text{poly}(n))$ , 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  $0, 1, 2, \dots$

Каждый из них будем или брать в подмножество-ответ, или не брать.

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

```

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

```

1 bool is[S+1] <-- 0;
2 is[0] = 1;
3 for (int i = 0; i < n; i++)
4     for (int j = S - a[i]; j >= 0; j--) // поменяли направление, важно
5         if (is[j])
6             is[j + a[i]] = 1;
7 // 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  $|, \&, \wedge, \ll$ , 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.

```

1 bitset<S+1> is;
2 is[0] = 1;
3 for (int i = 0; i < n; i++)
4     is |= is << w[i]; // выполняется за  $\mathcal{O}(\frac{S}{w})$ 
```

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 9.6.1, чтобы была возможность восстанавливать ответ.

```

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

И собственно восстановление ответа.

```

1 for (int w = S; w > 0; w -= a[last[w]])
2     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  $j$ :  $last[j], last[j-a[last[j]]], \dots$  все элементы строго меньше  $i$ .

## 9.7. Квадратичные динамики

**Def 9.7.1.** Подпоследовательностью последовательности  $a_1, a_2, \dots, a_n$  будем называть  $a_{i_1}, a_{i_2}, \dots, a_{i_k} : 1 \leq i_1 < i_2 < \dots < i_k \leq n$

**Задача НОП (LCS).** Поиск наибольшей общей подпоследовательности.

Даны две последовательности  $a$  и  $b$ , найти  $c$ , являющуюся подпоследовательностью и  $a$ , и  $b$  такую, что  $len(c) \rightarrow \max$ . 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  $\rightarrow \max$ .

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  $\max(f[1], f[2], \dots, f[n])$ , где  $n$  – длина последовательности.

Пересчёт:  $f[i] = 1 + \max_{j < i, a_j < a_i} f[j]$ , максимум пустого множества равен нулю.

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

## 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)

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  $i$ ? Для этого насчитаем две квадратные динамики:

$f[i, j]$  – НОП для первых  $i$  символов  $a$  и первых  $j$  символов  $b$  и

$g[i, j]$  – НОП для последних  $i$  символов  $a$  и последних  $j$  символов  $b$ .

Нас интересуют только последние строки –  $f[n']$  и  $g[n - n']$ , поэтому при вычислении можно хранить лишь две последние строки,  $\mathcal{O}(m)$  памяти. Выберем  $j$ :  $f[n', j] + g[n - n', m - j] = \max$ , сделаем два рекурсивных вызова, 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  $Time \leq nm + \lceil \frac{n}{2} \rceil m + \lceil \frac{n}{4} \rceil m + \dots \leq 4nm$ . ■

Подведём итог проделанной 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  $lcs[i, j] = \langle \text{длина НОП для } a[0..i], b[0..j], \text{ссылка назад} \rangle$ . 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.

```

1 def relax(i, j, pair, add):
2     pair.first += add
3     lcs[i, j] = min(lcs[i, j], pair)
4
5 def solve(n, a, m, b):
6     if n <= 2 or m <= 2:
7         return naive_lcs(n, a, m, b) # запустили наивное решение
8
9     # нужно хранить только 2 последние строчки lcs
10    lcs[] <-- -INF, lcs[0, 0] = 0; # инициализация
11    for i = 0..n-1:
12        for j = 0..m-1:
13            relax(i, j, lcs[i, j-1], 0)
14            relax(i, j, lcs[i-1, j], 0)
15            if (i > 0 and j > 0 and a[i-1] == b[j-1]):
16                relax(i, j, lcs[i-1, j-1], 1)
17            if (i == n/2):
18                lcs[i, j].second = j # самое важное, сохранили, где мы были в n/2-й строке
19
20    # нашли клетку, через которую точно проходит последовательность-ответ
21    i, j = n/2, lcs[n, m].second;
22    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 9.8.3  $\Rightarrow$  оценки те же.

#### Теорема 9.8.4.

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

### 9.8.5. Область применения 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