Первый курс, весенний семестр 2016/17
Конспект лекций по алгоритмам

Собрано 13 июня 2017 г. в 14:06

Содержание

1. Введение в теорию сложности **1**
- 1.1. Основные классы . . . 1
- 1.2. NP (non-deterministic polynomial, класс decision задач) . . . 2
- 1.3. Сведения, новые NP-полные задачи . . . 3

2. Рандомизированные алгоритмы **5**
- 2.1. Определения: RP, coRP, ZPP . . . 5
- 2.2. Примеры . . . 5
- 2.3. ZPP = RP ∩ coRP . . . 7
- 2.4. Двусторонняя ошибка, класс BPP . . . 7
- 2.5. Как ещё можно использовать случайные числа? . . . 8
- 2.6. Парадокс дней рождений. Факторизация: метод Полларда . . . 8
- 2.7. 3-SAT и random walk . . . 10
- 2.8. Лемма Шварца-Зиппеля . . . 10
- 2.9. Random shuffle . . . 11

3. Кратчайшие пути **12**
- 3.1. Short description . . . 12
- 3.2. bfs . . . 13
- 3.3. Модификации bfs . . . 13
  - 3.3.1. 1-k-bfs . . . 13
  - 3.3.2. 0-1-bfs . . . 14
- 3.4. Дейкстра . . . 14
- 3.5. A* (A-звездочка) . . . 15
- 3.6. Флойд . . . 16
  - 3.6.1. Восстановление пути . . . 16
  - 3.6.2. Поиск отрицательного цикла . . . 16

4. Кратчайшие пути **17**
- 4.1. Алгоритм Форд-Беллмана . . . 17
- 4.2. Выделение отрицательного цикла . . . 17
- 4.3. Модификации Форд-Беллмана . . . 18
  - 4.3.1. Форд-Беллман с break . . . 18
  - 4.3.2. Форд-Беллман с очередью . . . 18
  - 4.3.3. Форд-Беллман с random shuffle . . . 19
- 4.4. Потенциалы Джонсона . . . 20
- 4.5. Цикл минимального среднего веса . . . 20
- 4.6. Алгоритм Карпа . . . 21
- 4.7. (*) Алгоритм Гольдберга . . . 22

4.7.1. Решение за $\mathcal{O}(VE)$ . . . 22
4.7.2. Решение за $\mathcal{O}(EV^{1/2})$ . . . 22
4.7.3. Общий случай . . . 23

5. DSU и MST **23**
5.1. DSU: Система Непересекающихся Множеств . . . 24
5.1.1. Решения списками . . . 24
5.1.2. Решения деревьями . . . 24
5.1.3. Оценки $\mathcal{O}(\log^* n)$ и лучше . . . 26
5.2. MST: Минимальное Остовное Дерево . . . 27
5.2.1. Алгоритм Краскала . . . 27
5.2.2. Алгоритм Прима . . . 27
5.2.3. Алгоритм Борувки . . . 27
5.2.4. Лемма о разрезе . . . 28
5.3. (*) Оценка $\mathcal{O}(\alpha^{-1}(n))$ . . . 28
5.3.1. Введение обратной функции Аккермана . . . 28
5.3.2. Доказательство . . . 29

6. Жадность и приближённые алгоритмы **32**
6.1. Приближённые алгоритмы . . . 32
6.2. Коммивояжёр . . . 32
6.2.1. 2-OPT через MST . . . 32
6.2.2. 1.5-OPT через MST (Кристофилдс) . . . 32
6.3. Хаффман . . . 32
6.3.1. Хранение кодов . . . 33
6.4. Компаратор и сортировки . . . 33

7. Центроидная декомпозиция **33**
7.1. Введение и построение . . . 34
7.2. Реализация . . . 35
7.3. Решение задач . . . 35

8. BST и AVL **36**
8.1. BST . . . 36
8.2. Операции в BST, введение в персистентность . . . 37
8.3. AVL . . . 39
8.4. Split/Merge . . . 40
8.5. Дополнительные операции, групповые операции . . . 40
8.6. Неявный ключ . . . 41
8.7. Reverse на отрезке . . . 42
8.8. Итоги . . . 42
8.9. Персистентность: итоги . . . 42

9. B-tree и Treap **42**
9.1. B-дерево . . . 43
9.1.1. Поиск по B-дереву . . . 43
9.1.2. Добавление в B-дерево . . . 43
9.1.3. Удаление из B-дерева . . . 43

9.1.4. Модификации . . . 44
9.1.5. Split/Merge . . . 44
9.2. Производные B-дерева . . . 44
9.2.1. 2-3-дерево . . . 44
9.2.2. 2-3-4-дерево и RB-дерево . . . 44
9.2.3. AA-дерево (Arne Anderson) . . . 45
9.3. Treap . . . 45
9.4. Операции над Treap . . . 46

10. Splay и корневая оптимизация **47**
10.1. Rope . . . 48
10.2. Skip-list . . . 48
10.3. Splay tree . . . 49
10.4. SQRT decomposition . . . 51
10.4.1. Корневая по массиву . . . 51
10.4.2. Корневая через split/merge . . . 51
10.4.3. Корневая через split/rebuild . . . 51
10.4.4. Применение . . . 52
10.4.5. Оптимальный выбор k . . . 53
10.4.6. Корневая по запросам, отложенные операции . . . 53
10.5. Дополнение о персистентности . . . 53
10.5.1. Offline . . . 54
10.5.2. Персистентная очередь за $\mathcal{O}(1)$ . . . 54
10.6. Статическая оптимальность . . . 55
10.7. Другие деревья поиска . . . 55

11. Дерево отрезков **56**
11.1. Общие слова . . . 57
11.2. Дерево отрезков с операциями снизу . . . 57
11.3. Дерево отрезков с операциями сверху . . . 58
11.4. Динамическое дерево отрезков и сжатие координат . . . 60
11.5. 2D-деревья . . . 60
11.6. Сканирующая прямая . . . 61
11.7. k-я порядковая статистика на отрезке . . . 62
11.8. (*) Fractional Cascading . . . 63
11.9. (*) КД-дерево . . . 63

12. LCA & RMQ **63**
12.1. RMQ & Sparse table . . . 64
12.2. LCA & Двоичные подъёмы . . . 65
12.3. RMQ±1 за $\langle n, 1\rangle$ . . . 66
12.4. LCA $\to$ RMQ±1 и Эйлеров обход . . . 66
12.5. RMQ $\to$ LCA . . . 67
12.6. LCA в offline, алгоритм Тарьяна . . . 68
12.7. LA (level ancestor) . . . 68
12.8. Euler-Tour-Tree . . . 68
12.9. (*) LA, быстрые решения . . . 69

13. HLD & LC **69**
13.1. Heavy Light Decomposition 70
13.2. Link Cut Tree 71
13.3. MST за $\mathcal{O}(n)$ 73
13.4. RMQ Offline 74

Лекция #1: Введение в теорию сложности

15 февраля

1.1. Основные классы

• Алгоритмически не разрешимые задачи

Задач таких много (much enough).
Можно выделить такую общую тему, как "предсказание будущего сложного процесса".
Например, "в игре «жизнь» достижимо ли из состояния A состояние B?"
Наиболее каноническим примером является «проблема останова» (halting problem):
дана программа, остановится ли она когда-нибудь на данном входе?

Теорема 1.1.1. Halting problem алгоритмически не разрешима

Доказательство. От противного. Пусть есть алгоритм $M(A)$, всегда останавливающийся, и возвращающий `true` iff A останавливается. Дадим ему на вход следующую программу f:

```
1 void f() {
2   bool result = M(f)
3   if (result)
4     while (1);
5 }
```

Пришли к противоречию, т.к. если `result = true`, то f зависнет и наоборот. ■

• Decision/search problem

Если в задаче ответ – `true`/`false`, то это decision problem (задача распознавания). Иначе это search problem. Примеры:

1. Decision. Проверить, есть ли x в массиве a.
2. Search. Найти позицию x в массиве a.
3. Decision. Проверить, есть ли путь из a в b в графе G.
4. Search. Найти сам путь.
5. Decision. Проверить, есть ли в графе клика размера хотя бы k.
6. Search. Найти максимальный размер клики.

Decision problem f можно задавать, как язык (множество) $L = \{x \colon f(x) = \texttt{true}\}$.

• DTime, P, EXP (классы для decision задач)

Def 1.1.2. $\mathrm{DTime}(f(n))$ – *множество задач распознавания, для которых* $\exists C > 0$, *детерминированный алгоритм, работающий на всех входах не более чем* $C \cdot f(n)$

Def 1.1.3. $\mathrm{P} = \bigcup\limits_{k>0} \mathrm{DTime}(n^k)$. *Т.е. задачи, имеющие полиномиальное решение.*

Def 1.1.4. $\mathrm{EXP} = \bigcup\limits_{k>0} \mathrm{DTime}(2^{n^k})$. *Т.е. задачи, имеющие экспоненциальное решение.*

Теорема 1.1.5. $\mathrm{DTime}(f(n)) \subsetneq \mathrm{DTime}(f(n)\log^2 n)$

Доказательство. Задача, которую нельзя решить за $f(n)$: завершится ли данная программа за $f(n)\log n$ шагов. Подробнее можно прочесть на wiki. Там же дана более сильная формулировка теоремы, требующая большей аккуратности при доказательстве. ■

Следствие 1.1.6. $\mathrm{P} \neq \mathrm{EXP}$

Доказательство. $\mathrm{P} \subseteq \mathrm{DTime}(2^n) \subsetneq \mathrm{DTime}(2^{2n}) \subseteq \mathrm{EXP}$ ■

1.2. NP (non-deterministic polynomial, класс decision задач)

Def 1.2.1. $\mathrm{NP} = \{L\colon \exists M$, *работающий за полином от* $|x|$, $\forall x\,((\exists y\, M(x,y)=1) \Leftrightarrow (x \in L))\}$

Неформально: NP – класс таких decision задач, что для любого входа x ответ можно проверить с полиномиальной подсказкой ($\exists$ подсказка y).

Def 1.2.2. $\mathrm{coNP} = \{L\colon \overline{L} \in \mathrm{NP}\}$

Def 1.2.3. $\mathrm{coNP} = \{L\colon \exists M$, *работающий за полином от* $|x|$, $\forall x\,((\forall y\, M(x,y)=0) \Leftrightarrow (x \in L))\}$

Неформально: дополнение языка лежит в NP или "не существует подсказки" Примеры:

1. HAMPATH: есть ли в графе гамильтонов путь. Подсказка – сам путь. Алгоритм – проверка корректности подсказки-пути.
2. BOUNDED-HALTING: $x = \{M0\underbrace{11\ldots1}_{k}\}$, задача – проверить, существует ли вход, на котором M остановится за k шагов. Подсказка – такой вход для M. Алгоритм – моделирование M, работающее за $\mathcal{O}(poly(k))$. Заметим, что если бы число k было записано, используя $\log_2 k$ бит, моделирование работало бы за экспоненту от длины входа, и нельзя было бы сказать "задача лежит в NP".
3. k-CLIQUE – проверить наличие в графе клики размером k.
4. IS-SORTED – отсортирован ли массив? Лежит даже в P.
5. PRIME – является ли число простым. Лежит в coNP, подсказкой является делитель. На самом деле PRIME $\in$ P, но этого мы пока не умеем понять.

Замечание 1.2.4. $\mathrm{P} \subseteq \mathrm{NP}$ (можно взять пустую подсказку)

Замечание 1.2.5. Вопрос $\mathrm{P} = \mathrm{NP}$ или $\mathrm{P} \neq \mathrm{NP}$ (wiki) остаётся открытым.
Многие предполагают, что не равно.

• NP-hard, NP-complete

Def 1.2.6. $\exists$ *полиномиальное сведение задачи* A *к задаче* B: $(A \leqslant_P B) \Leftrightarrow \exists M$, *работающий за полином,* $(x \in A) \Leftrightarrow (f(x) \in B)$

Def 1.2.7. $\exists$ *сведение по Куку задачи* A *к задаче* B: $(A \leqslant_C B) \Leftrightarrow \exists M$, *решающий* A, *работающий за полином, которому разрешено обращаться к полиномиальному решению* B.

Ещё говорят задача A сводится к задаче B.

В обоих сведениях мы решаем задачу A, используя уже готовое решение задачи B.
Другими словами доказываем, что «A не сложнее B». Различие в том, что в первом случае решением B можно воспользоваться только 1 раз, во втором случае полином раз.

Def 1.2.8. $\mathrm{NP\text{-}hard} = \mathrm{NPh} = \{L\colon \forall A \in \mathrm{NP}\ A \leqslant_P L\}$

NP-трудные задачи. Класс задач, которые не проще любой задачи из класса NP.

Def 1.2.9. NP-complete = NPc = NPh ∩ NP

NP-полные задачи – самые сложные задачи в классе NP.
Задачи, которые не проще, чем любая другая в NP.

Замечание 1.2.10. Когда хотите выразить мысль, что задача трудная для решения (например, поиск гамильтонова пути), **неверно** говорить "это NP задача!" (любая из P тоже в NP) и странно говорить "задача NP-полная" (в этом случае вы имеете в виду сразу, что и трудная, и в NP). Логично сказать "задача NP-трудная".

• **NP-полные задачи существуют!**

Приведём простую и очень важную теорему. Доказательство можно на экзамене сформулировать в одно предложение, здесь же оно для понимания расписано максимально подробно.

Теорема 1.2.11. BH = BOUNDED-HALTING ∈ NPc

Доказательство. NPc = NP ∩ NPh. Мы уже показали, что BH лежит в NP.
Теперь покажем, что BH ∈ NPh. Алгоритм, проверяющий подсказки для BH обозначим B.
Пусть $A \in$ NP, тогда $\exists M$, полиномиальный, проверяющий подсказки для A.
Полиномиальный, значит $\exists P(n)$, ограничивающий время работы M.
Нам нужно научиться преобразовывать вход x для M к входу B. Рассмотрим

```
1 def R(y):
2   if M(x, y) = false: зависни
```

T.e. R останавливается на каком-то входе iff существует подсказка, на которой M даёт `true`. Если R останавливается, то делает это за $P(|x|)$ шагов. $R0\underbrace{11\ldots1}_{P(|x|)}$ – вход для B. ■

1.3. Сведения, новые NP-полные задачи

Чтобы доказать, что $B \in$ NPh, нужно взять любую $A \in$ NPh и свести A к B полиномиально. Пока такая задача A у нас одна. На самом деле их очень много.

Чтобы доказать, что $B \in$ NPc, нужно ещё не забыть проверить, что $B \in$ NP. Во всех теоремах ниже эта проверка очевидна, мы проведём её только в доказательстве первой.

• BH → CIRCUIT-SAT → SAT → 3-SAT → k-INDEPENDENT → k-CLIQUE

Def 1.3.1. CIRCUIT-SAT. *Дана схема, состоящая из входов, выхода, булевых гэйтов. Проверить, существует ли набор значений на входах, что на выходе* `true`.

Теорема 1.3.2. CIRCUIT-SAT ∈ NPc

Доказательство. Подсказка – набор значений на входах ⇒ CIRCUIT-SAT ∈ NP.
Нам дано время выполнения программы t, вход x. За время t программа обратится не более чем к t ячейкам памяти. Обозначим за $x_{i,j}$ состояние `true`/`false` j-й ячейки памяти в момент времени i. Заметим, что $x_{0,j}$ – вход. $x_{t,output}$ – выход (результат), а каждая $x_{i,j} \forall i \in [1, t)$ зависит от $\mathcal{O}(1)$ переменных предыдущего слоя. Чтобы все внутренние гейты были булевы, перепишем их в КНФ-форме, при этом количество гейтов увеличится в $\mathcal{O}(1)$ раз. ■

Теорема 1.3.3. SAT $\in$ NPC

Доказательство. В разборе практики смотрите сведение из CIRCUIT-SAT. ■

Теорема 1.3.4. 3-SAT $\in$ NPC

Доказательство. Пусть есть клоз $(x_1 \vee x_2 \vee \cdots \vee x_n), n \geqslant 4$.
Введём новую переменную w и заменим его на $(x_1 \vee x_2 \vee w) \wedge (x_3 \vee \cdots \vee x_n \vee \overline{w})$. ■

Теорема 1.3.5. k-INDEPENDENT $\in$ NPC

Доказательство. Построим для каждого из m клозов $(l_1 \vee l_2 \vee l_3)$, где l_i – литералы, треугольник $\langle l_1, l_2, l_3 \rangle$. Получили граф из $3m$ вершин и $3m$ рёбер. В этом графе в любое независимое множество входит максимум одна вершина из каждого треугольника. Теперь $\forall i$ соединим все вершины x_i со всеми вершинами $\overline{x_i}$. Теперь в независимое множество нельзя одновременно включить вершины с разными значениями x_i.
Утверждается, что $\exists$ m-INDEPENDENT iff у 3-SAT было решение. ■

Теорема 1.3.6. k-CLIQUE $\in$ NPC

Доказательство. Есть простое двустороннее сведение k-CLIQUE $\leftrightarrow$ k-INDEPENDENT.
c_{ij} – есть ли ребро между i и j вершинами. Создадим новый граф: $c'_{ij} = \overline{c_{ij}} \wedge (i \neq j)$. ■

• Сведение search задач к decision задачам: число $\rightarrow$ bool

Пусть мы умеем проверять, есть ли в графе клика размера k.
Чтобы найти размер максимальной клики, достаточно применить бинпоиск по ответу.
Это общая техника, применимая для максимизации/минимизации численной характеристики.

• Сведение search задач к decision задачам: набор для SAT

Мы умеем проверять, разрешим ли SAT. Найдём выполняющий набор по индукции:
сделаем $x_n = 0$, получим новую формулу, если она не выполнима, $x_n = 1$.
В любом случае получили задачу размера $n - 1$.

Любую NP-полную задачу можно свести к SAT. То же сведение обычно применимо и для восстановления из выполняющего набора для SAT ответа к исходной задачи. Например, чтобы найти решение для k-INDEPENDENT, выберем вершины, которым соответствуют $x_i =$ `true`.

• Решение NP-полных задач

Пусть вам дана NP-полная задача. С одной стороны плохо – для неё нет быстрого решения. С другой стороны её можно свести к SAT, для которого несколько десятилетий успешно оптимизируются специальные SAT-solvers. Например, вы уже можете решать k-CLIQUE, построив вход к задаче SAT и скормить его `python3` пакету pycosat.
А ещё можно принять участие в соревновании.

Лекция #2: Рандомизированные алгоритмы

22 февраля

2.1. Определения: RP, coRP, ZPP

Рандомизированными называют алгоритмы, использующие случайные биты. Первый тип алгоритмов: решающие decision задачи, работающие всегда за полином, ошибающиеся в одну сторону. Строго это можно записать так:

Def 2.1.1. $\text{RP} = \{L\colon \exists M \in \text{PTime} \begin{cases} x \notin L \Rightarrow M(x,y) = 0 \\ x \in L \Rightarrow Pr_y[M(x,y) = 1] \geqslant \frac{1}{2} \end{cases}\}$

x – вход, y – подсказка из случайных бит.
То есть, если $x \notin L$, M не ошибается, иначе работает корректно с вероятностью хотя бы $\frac{1}{2}$.
Заметим, если для какого-то y алгоритм M вернул 1, то это точно правильный ответ, $x \in L$.

Def 2.1.2. $\text{coRP} = \{L\colon \exists M \in \text{PTime} \begin{cases} x \in L \Rightarrow M(x,y) = 1 \\ x \notin L \Rightarrow Pr_y[M(x,y) = 0] \geqslant \frac{1}{2} \end{cases}\}$

Заметим, если для какого-то y алгоритм M вернул 0, то это точно правильный ответ, $x \notin L$.

• **Сравнение классов** NP**,** RP

Если ответ 0, оба алгоритма на любой подсказке выдадут 0. Если ответ 1, для NP-алгоритма должна существовать хотя бы одна подсказка, а RP-алгоритм должен корректно работать хотя бы на половине подсказок.

• **Понижение ошибки**

Конечно, алгоритм, ошибающийся с вероятностью $\frac{1}{2}$ никому не нужен.

Lm 2.1.3. Пусть M – RP-алгоритм, ошибающийся с вероятностью p.
Запустим его n раз, если хотя бы раз вернул 1, вернём 1. Вероятность ошибки – p^n.

Например, если повторить 100 раз, получится вероятность ошибки $2^{-100} \approx 0$.

Если есть алгоритм, *корректно* работающий с близкой к нулю вероятностью p, ошибающийся с вероятностью $1-p$, то повторив его $\frac{1}{p}+1$ раз, получим вероятность ошибки $(1-p)^{\frac{1}{p}+1} \leqslant e^{-1}$.

• **ZPP: алгоритмы без ошибки**

ZPP – класс задач, для которых есть никогда не ошибающийся алгоритм, с полиномиальным *в среднем* временем работы.

Def 2.1.4. $\text{ZPP} = \{L\colon \exists P(n), M \textit{ такие, что } E_y[Time(M(x,y))] \leqslant P(|x|)\}$

В русском и те, и те алгоритмы называют вероятностными/рандомизированными.
В английском более точно RP и coRP называть *probabilistic*, а ZPP *randomized*.

Мы определили основные классы только для полиномиального времени.
Для любого другого (экспоненциального, линейного и т.д.) можно определить аналогичные.

Важно помнить, что в RP, coRP, ZPP алгоритмы работают на *всех* тестах.
Например, вероятность ошибки в RP для любого теста не более $\frac{1}{2}$.

2.2. Примеры

• Поиск квадратичного невычета

Для любого простого p ровно $\frac{p-1}{2}$ ненулевых остатков обладают свойством $a^{\frac{p-1}{2}} \equiv -1 \bmod p$.
Алгоритм: пока не найдём, берём случайное a, проверяем. Время одной проверки – возведение в степень ($\log p$ для $p \leqslant 2^w$). $E_{random}[Time] = X = (\log p) + \frac{1}{2}X \Rightarrow X = 2\log p$.

• Проверка на простоту: тест Ферма

Малая теорема Ферма: $\forall p \in \mathbb{P}\ \forall a \in [1..p-1]\ \ a^{p-1} \equiv 1 \bmod p$.
Чтобы проверить простоту p, можно взять случайное a и проверить $a^{p-1} \equiv 1 \bmod p$

Lm 2.2.1. $\exists a$: $a^{p-1} \not\equiv 1 \bmod p \Rightarrow$ таких a хотя бы $\frac{p-1}{2}$.

Доказательство. Пусть b такое, что $b^{p-1} \equiv 1 \bmod p \Rightarrow (ab)^{p-1} = a^{p-1}b^{p-1} \not\equiv 1 \bmod p$. ■

Мы показали, что если тест вообще работает, то с вероятностью хотя бы $\frac{1}{2}$.
К сожалению, есть составные числа, для которых тест вообще не работает – числа Кармайкла.

Утверждение 2.2.2. У числа Кармайкла a есть простой делитель не более $\sqrt[3]{a}$.

Получаем следующий всегда корректный вероятностный алгоритм:
проверим возможные делители от 2 до $\sqrt[3]{a}$, далее тест Ферма.

• Проверка на простоту: тест Миллера-Рабина

```
1  bool isPrime( int a ) {
2    // a = 2^s t, t - нечётно
3    if (a < 2) return 0; // 1 и 0 не простые
4    a --> (s, t)
5    g = pow(a, t) // возвели в степень по модулю
6    for (int i = 0; i < s; i++) {
7      if (g == 1) return 1;
8      if (g mod p != -1 && g * g mod p == 1) return 0;
9      g = g * g mod p;
10   }
11   return g == 1; // тест Ферма
12 }
```

В строке 8 мы проверяем, что нет корней из 1 кроме 1 и -1.

Утверждение 2.2.3. $\forall a$ вероятность ошибки не более $\frac{1}{4}$.

• Изученные алгоритмы

k-я порядковая статистика учит нас "*вместо того, чтобы выбирать медиану, достаточно взять случайные элемент*". Эта техника нередко срабатывает.

Поиск числа, которое встречается в массиве больше половины раз ещё раз иллюстрирует мощь техники "взять случайный элемент". У этой задачи есть две версии.
ZPP-версия: "*мы уверены, что такой элемент есть, пробуем случайные, пока не попадём*".
RP-версия: "*хотим определить, есть ли такой элемент в массиве, делаем одну пробу*".

3-LIST-COLORING: вычеркнуть из каждого клоза случайный элемент, получить 2-SAT. Вероятность успеха этого алгоритма $\frac{2^n}{3^n}$, чтобы получить вероятность $\frac{1}{2}$, нужно было бы повторить

его $1.5^n \ln 2$ раз, поэтому к RP-алгоритмам он не относится.
Ещё про него полезно понимать "*работает на* 2^n *подсказках из* 3^n *возможных*".

• **Проверка** $AB = C$

Даны три матрицы $n \times n$, нужно за $\mathcal{O}(n^2)$ проверить равенство $AB = C$. Все вычисления в $\mathbb{F}_2$. Вероятностный алгоритм генерирует случайный вектор x и проверяет $A(Bx) = Cx$.

Lm 2.2.4. Это RP-алгоритм, вероятность ошибки не более $\frac{1}{2}$.

Доказательство. Пусть $AB \neq C$, посчитаем $Pr_x\big[A(Bx) = Cx\big]$. Обозначим $D = AB - C$. Значит $D \neq 0$, а мы считаем вероятноть и $Pr_x\big[Dx = 0\big]$. Пусть $D_{i,j} \neq 0$. Зафиксируем все элементы x кроме j-го, $(Dx)_i = D_{i,0}x_0 + \cdots + D_{i,j}x_j + \cdots + D_{n-1,0}x_{n-1}$. Теперь, меняя значение x_j, меняем значение $(Dx)_i \Rightarrow Pr_x\big[(Dx)_i = 0\big] = \frac{1}{2} \Rightarrow Pr_x\big[Dx = 0\big] \leqslant \frac{1}{2}$ ■

2.3. ZPP = RP ∩ coRP

Lm 2.3.1. Неравенство Маркова: $\forall a > 1\, Pr[x \leqslant aE(x)] \geqslant \frac{1}{a}$, иначе говоря $Pr[x > aE(x)] < \frac{1}{a}$

Доказательство. От противного. Пусть $Pr[x > aE(x)] \geqslant \frac{1}{a} \Rightarrow E(x) > \big(aE(x)\big)\frac{1}{a} = E(x)$?!? ■

Теорема 2.3.2. ZPP = RP ∩ coRP

Доказательство. Пусть $L \in$ ZPP $\Rightarrow \exists$ алгоритм M, работающий в среднем за $P(n)$ $\overset{2.3.1}{\Rightarrow}$ $Pr[Time(M) \leqslant 2P(n)] \geqslant \frac{1}{2} \Rightarrow$ запустим M с будильником на $2P(n)$ операций. Если алгоритм завершился до будильника, он даст верный ответ, иначе вернём $1 \Rightarrow$ RP или $0 \Rightarrow$ coRP.
Пусть $L \in$ P ∩ coRP $\Rightarrow \exists M_1$(RP), M_2(coRP), работающие за полином, ошибающиеся в разные стороны, которые можно запускать по очереди $(M_1 + M_2)$. Пусть $x \notin L \Rightarrow M_1(x) = 0$, $Pr[M_2(x) = 0] \geqslant \frac{1}{2}$. Сколько раз нужно в среднем запустить $M_1 + M_2$? $E \geqslant 1 + \frac{1}{2}E \geqslant 2$ (всегда запустим 1 раз, затем с вероятностью $\frac{1}{2}$ получим $M_2(x) = 1$ и повторим процесс с начала). ■

Lm 2.3.3. P ⊆ coRP ∩ RP = ZPP ⊆ RP ⊆ NP

При этом строгие ли вложения неизвестно.
Зато известна масса задач, для которых есть простое RP-решение, но неизвестно P-решение.

2.4. Двусторонняя ошибка, класс BPP

На практике такие алгоритмы нам вряд ли встретятся, но всё же они есть:

Def 2.4.1. $\text{BPP} = \{L\colon \exists M \in \text{PTime} \begin{cases} x \notin L \Rightarrow Pr_y[M(x,y) = 0] \geqslant \alpha \\ x \in L \Rightarrow Pr_y[M(x,y) = 1] \geqslant \alpha \end{cases}\}$, *где* $\alpha = \frac{2}{3}$.

Другими словами алгоритм всегда даёт корректный ответ с вероятностью хотя бы $\frac{2}{3}$.
Для BPP тоже можно понижать ошибку: запустим n раз, выберем ответ, который чаще встречается (majority). На самом деле α в определении можно брать сколь угодно близким к $\frac{1}{2}$.

Lm 2.4.2. О понижении ошибки. Пусть $\beta > 0, \alpha = \frac{1}{2} + \varepsilon,\ \varepsilon > 0 \Rightarrow \exists n = poly(\frac{1}{\varepsilon})$ такое, что повторив алгоритм n раз, и вернув majority, мы получим вероятность ошибки не более β.

Доказательство. Пусть $x \in L$, мы повторили алгоритм $n = 2k$ раз.
Если получили 1 хотя бы $k{+}1$ раз, вернём 1, иначе вернём 0.
Вероятность ошибки $\texttt{Err} = \sum\limits_{i=0}^{i=k} p_i$, где p_i – вероятность того, что алгоритм ровно i раз вернул 1.
Для $\varepsilon = 0$ введём аналогичное обозначение q_i. Заметим, что $q_i = \binom{n}{i}(\frac{1}{2})^n$.
Из соображений симметрии мы знаем, что $\sum\limits_{i=0}^{i=k} q_i = \frac{1}{2} \Rightarrow \texttt{Err} = \frac{1}{2}\frac{\sum p_i}{\sum q_i} \leqslant \frac{1}{2}\max_i \frac{p_i}{q_i}$.
Осталось выписать формулу для p_i, при этом удобно отсчитывать i от середины: $i = k - j$.

$$p_i = p_{k+j} = \binom{n}{i}(\tfrac{1}{2}+\varepsilon)^{k-j}(\tfrac{1}{2}-\varepsilon)^{k+j} = \binom{n}{i}(\tfrac{1}{4}-\varepsilon^2)^k(\tfrac{1/2-\varepsilon}{1/2+\varepsilon})^j$$

Теперь оценим ошибку:

$$\frac{p_i}{q_i} = \frac{p_{k-j}}{q_{k-j}} = \frac{\binom{n}{i}\left(\frac{1}{4}-\varepsilon^2\right)^k\left(\frac{1/2-\varepsilon}{1/2+\varepsilon}\right)^j}{\binom{n}{i}\left(\frac{1}{2}\right)^n} = \frac{\left(\frac{1}{4}-\varepsilon^2\right)^k\left(\frac{1/2-\varepsilon}{1/2+\varepsilon}\right)^j}{(1/2)^n} = \left(\frac{1/4-\varepsilon^2}{1/4}\right)^k\left(\frac{1/2-\varepsilon}{1/2+\varepsilon}\right)^j$$

Теперь заметим, что максимум достигается при $j = 0$, поэтому

$$\texttt{Err} \leqslant \frac{1}{2}\frac{p_k}{q_k} = \frac{1}{2}\left(1 - \frac{1}{(1/4)(1/\varepsilon^2)}\right)^k$$

Хотим, чтобы ошибка была не больше β, для этого возьмём $k = (1/4)(1/\varepsilon^2)\cdot\ln\frac{1}{\beta} = poly(\frac{1}{\varepsilon})$. ■

2.5. Как ещё можно использовать случайные числа?

• Идеальное кодирование

Алиса хочет передать Бобу некое целое число x от 0 до $m-1$. У них есть общий ключ r от 0 до $m-1$, сгенерированный равномерным случайным распределением. Тогда Алиса передаст по открытому каналу $y = (x+r) \bmod m$. Знание **одного** такое числа не даст злоумышленнику ровно никакой информации. Боб восстановит $x = (y-r) \bmod m$.

• Вычисления без разглашения

В некой компании сотрудники стесняются говорить друг другу, какая у кого зарплата. Зарплату i-го обозначим x_i. Но очень хотят посчитать среднюю $\Leftrightarrow$ посчитать сумму $x_1 + \cdots + x_n$. Для этого они предполагают, что сумма меньше $m = 10^{18}$, и пользуются следующим алгоритмом:

1. Первый сотрудник генерирует случайное число $r = 0..m{-}1$.
2. Первый сотрудник передаёт второму число $(r + x_1) \bmod m$.
3. Второй сотрудник передаёт третьему число $(r + x_1 + x_2) \bmod m$.
4. ...
5. Последний сотрудник передаёт первому $(r + x_1 + x_2 + \cdots + x_n) \bmod m$.
6. Первый, как единственный знающий r, вычитает его и говорит всем ответ.

• Приближения

На практике будет разобрано $\frac{1}{2}$-`opt` приближение для MAX-3-SAT.

2.6. Парадокс дней рождений. Факторизация: метод Полларда

«В классе 27 человек $\Rightarrow$ с большой вероятностью у каких-то двух день рождение в один день»
Пусть p_k – вероятность, что среди k случайных чисел от 1 до n все различны. Оценим p_k снизу.

Lm 2.6.1. $1 - p_k \leqslant \frac{k(k-1)}{2n}$

Доказательство. $f = \#\{(i,j)\colon x_i{=}x_j\}$, $1{-}p_k = \texttt{Pr}[f{>}0] \leqslant E_k[f] = \frac{k(k-1)}{2}\cdot Pr[x_i{=}x_j] = \frac{k(k-1)}{2}\frac{1}{n}$ ■

Следствие 2.6.2. При $k = o(\sqrt{n})$ получаем $1 - p_k = o(1) \Rightarrow$ с вероятностью ≈ 1 все различны.

Теперь оценим p_k при $k = \sqrt{n}$.

Lm 2.6.3. $k = \sqrt{n} \Rightarrow 0.4 \leqslant p_k \leqslant 0.8$

Доказательство. $p_k = \underbrace{\frac{n}{n} \cdot \frac{n-1}{n} \cdot \ldots \cdot \frac{n-\sqrt{n}/2}{n}}_{(n-\sqrt{n}/2)/n \leqslant x \leqslant 1} \cdot \underbrace{\frac{n-\sqrt{n}/2-1}{n} \cdot \ldots \cdot \frac{n-\sqrt{n}}{n}}_{(n-\sqrt{n})/n \leqslant x \leqslant (n-\sqrt{n}/2)/n} \Rightarrow \left(\frac{n-\sqrt{n}/2}{n}\right)^{\sqrt{n}/2} \left(\frac{n-\sqrt{n}}{n}\right)^{\sqrt{n}/2} \leqslant$

$p_k \leqslant (1)^{\sqrt{n}/2} \left(\frac{n-\sqrt{n}/2}{n}\right)^{\sqrt{n}/2} \Rightarrow e^{-1/4} e^{-1/2} \leqslant p_k \leqslant e^{-1/4} \Rightarrow 0.4723 \leqslant p_k \leqslant 0.7788$ ■

$\forall k$ можно оценить p_k гораздо точнее:

$p_k = \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \ldots \cdot \frac{n-k+1}{n} = \frac{n!}{n^k (n-k)!} \approx \frac{(n/e)^n}{n^k ((n-k)/e)^{n-k}} = \frac{1}{((n-k)/n)^{n-k} e^k} = \frac{1}{(1-k/n)^{n-k} e^k} \approx \frac{1}{(e^{-1})^{(n-k)k/n} e^k}$

Первое приближение сделано по формуле Стирлинга, второе по замечательному пределу.

• Простой алгоритм факторизации

Сформулируем задачу: дано целое число n, найти хотя бы один нетривиальный делитель n. Можно предполагать, что число n не простое, так как есть тест Миллера-Рабина.

```
1 int factor( int n ) {
2   for (int x = 2; x * x <= n; x++)
3     if (n % x == 0)
4       return x;
5   return -1;
6 }
```

• Простой рандомизированный алгоритм факторизации

```
1 int factor( int n ) {
2   int x = random [2..n-1];
3   int g = gcd(n, x);
4   return g == 1 ? -1 : g;
5 }
```

Получили вероятностный алгоритм. Если $n = px$, то $Pr[g > 1] \geqslant Pr[p|x] \geqslant \frac{n/p-1}{n-1} = \frac{1}{p} - \frac{1}{n} \geqslant \frac{1}{2p}$. Если повторить $2p$ раз, получим константную вероятность ошибки и время работы $\mathcal{O}(p \cdot gcd)$.

• Поллард

Пусть у n минимальный делитель p, тогда $p \leqslant \sqrt{n}$. Сгенерируем $\sqrt[4]{n} \geqslant \sqrt{p}$ случайных чисел. По парадоксу дней рождений какие-то два (x и y) дадут одинаковый остаток по модулю p. При этом с большой вероятностью у x и y разный остаток по модулю $n \Rightarrow gcd(x - y, n)$ – нетривиальный делитель n. Осталось придумать, как найти такую пару x, y.

```
1 x = random [2..n-1]
2 k = pow(n, 1.0 / 4)
3 for i=0..k-1: x = f(x)
```

Здесь функция f – псевдорандом на $[0..n)$.
Например (без док-ва) нужными нам свойствами обладает функция $f(x) = (x^2 + 1) \bmod n$.
Так мы получили x, теперь переберём y.

```
1 y = f(x)
2 for i=0..k-1:
```

```
3   g = gcd(x - y, n)
4   if (g != 1 && g != n) return g
5   y = f(y)
```

Полученный нами алгоритм будет иметь проблемы на маленьких n, которые можно проверить за $n^{1/2}$. Также стоит помнить, что вероятность его успеха $\approx \frac{1}{2}$, поэтому для вероятности ≈ 1 всю конструкцию нужно запустить несколько раз.

2.7. 3-SAT и random walk

• **Детерминированное решение за** $3^{n/2}$

Пусть решение X^* существует и в нём нулей больше чем единиц. Начнём с $X_0 = \{0, 0, \ldots, 0\}$, чтобы из X_0 попасть в X^* нужно сделать не более $\frac{n}{2}$ шагов. Если X_i не решение, то какой-то клоз не выполнен, значит в X^* одна из трёх переменных этого клоза имеет другое значение. Переберём, какая. Получили рекурсивный перебор глубины $\frac{n}{2}$, с ветвлением 3. Если в X^* единиц больше, начинать нужно с $X_0 = \{1, 1, \ldots, 1\}$. Нужно перебрать оба варианта.

• **Рандомизированное решение за** $3^{n/2}$

Упростим предыдущую идею: начнём со случайного X_0, с вероятностью $\frac{1}{2}$ расстояние до X^* не более $\frac{n}{2}$, сделаем $\frac{n}{2}$ шагов, выбирая каждый раз случайное из трёх направлений. Если перебор за $3^{n/2}$ перебирал все варианты, то мы перебираем 1 вариант и угадываем с вероятностью как минимум $\frac{1}{3^{n/2}}$. Чтобы алгоритм имел константную вероятность ошибки, повторить процесс нужно $3^{n/2}$ раз.

• **Рандомизированное решение за** $1.334^{n/2}$

Schoning's algorithm. Получается из предыдущего решения заменой $\frac{n}{2}$ шагов на $3n$ шагов. Вероятность успеха будет не менее $(3/4)^n$. Доказательство.

• **Поиск хороших кафешек**

Пусть Вы в незнакомом городе, хотите найти хорошее кафе. Рассмотрим следующие стратегии:

1. Осматривать окрестность в порядке удаления от начальной точки
2. Random walk без остановок
3. Random walk на фиксированную глубину с возвратом

Первый, если Вы начали в не очень богатом на кафе районе не приведёт ни к чему хорошему. У второго будут проблемы, если в процессе поиска вы случайно зайдёте в промышленный район. Третий же в среднем не лучше, зато минимизирует риски, избавляет нас от обеих проблем. Для решения 3-SAT мы использовали именно третий вариант.

2.8. Лемма Шварца-Зиппеля

Lm 2.8.1. Пусть дан многочлен P от нескольких переменных над полем $\mathbb{F}$.
$(P \not\equiv 0) \Rightarrow Pr_x[P(x) = 0] \leqslant \frac{degP}{|\mathbb{F}|}$

Доказательство. На экзамене его не будет. Индукция по числу переменных.
База: многочлен от 1 переменной имеет не более $degP$ корней. Переход: wiki. ■

Следствие 2.8.2. Задача проверки тождественного равенства многочлена нулю $\in$ RP.

Доказательство. Подставим случайный x в поле $\mathbb{F}_q$, где q – простое, $q > 2degP$. ■

• Совершенное паросочетание в произвольном графе

Дан неорграф, заданный матрицей смежности c. Нужно проверить, есть ли в нём совершенное паросочетание.

Def 2.8.3. *Матрица Тамма* T: $T_{ij} = -T_{ji}, T_{ij} = \begin{cases} 0 & c_{ij} = 0 \\ x_{ij} & c_{ij} = 1 \end{cases}$

Здесь x_{ij} – различные переменные. Пример: $c = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \to \begin{bmatrix} 0 & x_{12} & x_{13} \\ -x_{12} & 0 & 0 \\ -x_{13} & 0 & 0 \end{bmatrix}$

Теорема 2.8.4. $\det T \not\equiv 0 \Leftrightarrow \exists$ совершенное паросочетание

Теорему вам докажут в рамках курса дискретной математики, нам же сейчас интересно, как проверить тождество. Если бы матрица состояла из чисел, определитель можно было бы посчитать Гауссом за $\mathcal{O}(n^3)$. В нашем случае определитель – многочлен степени n. Подставим во все x_{ij} случайные числа, посчитаем определитель по модулю $p = 10^9 + 7$. Получили вероятностный алгоритм с ошибкой $\frac{n}{p}$ и временем $\mathcal{O}(n^3)$.

• Гамильтонов путь

В 2010-м его в неорграфе научились проверять наличие гамильтонова пути за $\mathcal{O}^*(1.657^n)$. Можно почитать в оригинальной статье Бьёркланда (Björklund).

2.9. Random shuffle

```
1 void Shuffle( int n, int *a ) {
2   for (int i = 0; i < n; i++) {
3     int j = rand() % (i + 1);
4     swap(a[i], a[j]);
5   }
6 }
```

Утверждение 2.9.1. После процедуры `Shuffle` все перестановки a равновероятные.

Доказательство. Индукция. База: после фазы $i = 0$ все перестановки длины 1 равновероятны. Переход: выбрали равновероятно элемент, который стоит на i-м месте, после этого часть массива $[0..i)$ по индукции содержит случайную из $i!$ перестановок. ■

• Применение

Если на вход даны некоторые объекты, полезно их первым делом перемешать.

Пример: построить бинарное дерево поиска, содержащее данные n различных ключей. Если добавлять их в пустое дерево в данном порядке, итоговая глубина может быть n, а время построения соответственно $\Theta(n^2)$. Если перед добавлением сделать `random shuffle`, то матожидание глубины дерева $\Theta(\log n)$, матожидание времени работы $\Theta(n \log n)$. Оба факта мы докажем при более подробном обсуждении деревьев поиска.

Лекция #3: Кратчайшие пути

1 марта

Def 3.0.1. *Взвешенный граф – каждому ребру соответствует вещественный вес, обычно обозначают w_e (weight) или c_e (cost). Вес пути – сумма весов рёбер.*

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

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

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

3.1. Short description

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

Задача	Время	Название	Год, Автор	Изучим?
ациклический граф	$V+E$	Динамика	–	+
$w_e = 1$	$V+E$	Поиск в ширину	1950, Moore	+
$w_e \geqslant 0$	V^2+E	–	1956, Dijkstra	+
$w_e \geqslant 0$	$V \log V + E$	Dijkstra + fib-heap	1984, Fredman & Tarjan	+
$w_e \in \mathbb{N}$, неорграф	$V+E$	–	2000, Thorup	-
$w_e \geqslant 0$	$A^* \leqslant$ Dijkstra	(A^*) A-star	1968, Nilsson & Raphael	+
любые веса	VE	–	1956, Bellman & Ford	+
$w_i \in \mathbb{Z} \cap [-N, \infty)$	$E\sqrt{V}\lceil\log(N+2)\rceil$	–	1994, Goldberg	+

Чтобы решить APSP запуском SSSP-решение от всех вершин. Время получится в V раз больше.
Кроме того есть несколько алгоритмов специально для APSP:

Задача	Время	Название	Год, Автор	Изучим?
любые веса	V^3	–	1962, Floyd & Warshall	+
любые веса	$VE + V^2 \log V$	–	1977, Johnson	+
любые веса	$V^3/2^{\Omega(\log^{1/2} V)}$	–	2014, Williams	-

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

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

3.2. bfs

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

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

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

• Версия с очередью

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

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

Заметим, что `q` – очередь.

3.3. Модификации bfs

3.3.1. 1-k-bfs

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

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

Решение несколькими очередями

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

3.3.2. 0-1-bfs

Возьмём обычный `bfs` с очередью, заменим очередь на дек. Разрешим вершине добавляться в дек несколько раз: добавлять будем каждый раз, когда расстояние уменьшается. Алгоритм, как ни пиши корректен, осталось добавить эвристику, ограничивающую время работы:

• **Алгоритм**

Если идём по ребру $e\colon a \to b$ и расстояние до b уменьшилось, то кладём b в дек. Если $w_e = 0$, то в начало дека, иначе в конец дека.
Оптимизация: если мы достаём вершину из дека второй раз, пропускаем её.

Теорема 3.3.1. Каждая вершина попадёт в дек не более двух раз.

Доказательство. Когда мы первый раз достаём вершину v из дека по индукции верно
(a) Расстояние d_v до вершины v посчитано правильно.
(b) Все вершины с меньшим расстоянием мы уже рассмотрели, и рёбра из них тоже.
(c) В деке сейчас лежат вершины $x\colon d_x = d_v \vee d_x = d_v + 1$.
Из третьего следует, что для всех будущих вершин $d_x \geqslant d_v$.
Пусть по ребру $e\colon v \to x$ произошла релаксация.
Если $w_e = 0$, то $d_x = d_v$ и d_x посчитано правильно, x сразу попадает в начало дека.
Если $w_e = 1$, то в d_x записано $d_v + 1$, может после этого уменьшиться ровно один раз до d_v. ■

Следствие 3.3.2. Время работы $\mathcal{O}(V + E)$.

3.4. Дейкстра

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

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

Время работы Дейкстры = $V \cdot$ `ExtractMin` $+ E \cdot$ `DecreaseKey`.
(a) Реализация без куч даст время работы $\Theta(E + V^2)$.
(b) С бинарной кучей даст время работы $\mathcal{O}(E \log V)$. Обычно пишут именно так.
(c) С кучей Фибоначчи даст время работы $\mathcal{O}(E + V \log V)$.
(d) С деревом Ван-Эмбде-Боэса даст время работы $\mathcal{O}(E \log\log C)$ для целых весов из $[0, C)$.
(e) Существуют более крутые кучи, дающие время $\mathcal{O}(E + V \log\log V)$ на целочисленных весах.

Замечание 3.4.1. Можно запускать Дейкстру и на графах с отрицательными рёбрами. Если разрешить вершину доставать из кучи несколько раз, алгоритм останется корректным, но на некоторых тестах будет работать экспоненциально долго. В среднем, кстати, те же $E \log V$.

3.5. A* (A-звездочка)

Представьте, что едете из Петербурга в Москву. Представьте себе дорожную сеть страны. Перед поездкой ищем кратчайший маршрут. Что делает Дейкстра? Идёт из Петербурга во все стороны, перебирает города в порядке удаления от Петербурга. То есть, попытается в частности ехать в Москву через Петрозаводск (427 км) и Выборг (136 км). Логичней было бы в процессе поиска пути рассматривать только те вершины (города, населённые пункты, развязки), проезжая через которые, теоретически можно было бы быстрее попасть в Москву...

Почему так? Дейкстра – алгоритм для SSSP. Цель Дейкстры – найти расстояния до всех вершин. При поиске расстояния до конкретной вершины t в Дейкстру можно добавить естественную оптимизацию – `break` после того, как достанем t из кучи.

Алгоритм A* можно воспринимать, как "модификацию Дейкстру, которая не пытается ехать Петербург $\rightsquigarrow$ Москва через Выборг". Обозначим начальную вершину s, конечную t, расстояние между вершинами a и b за d_{ab}. $\forall v$ оценим снизу d_{vt} *из физического смысла задачи*, как f_v.

Алгоритм A*: в Дейкстре ключ кучи d_v заменим на $d_v + f_v$.
То есть, вершины будем перебирать в порядке возрастания не d_v, а $d_v + f_v$.
Остановим алгоритм, когда вынем из кучи вершину t.

Если разрешить алгоритму одну вершину доставать из кучи несколько раз, и не останавливать его при вынимании вершины t, алгоритм в итоге найдёт корректные расстояния. При этом, возможно, он будет работать экспоненциально долго.

Теорема 3.5.1. Если функция f удовлетворяет неравенству треугольника: $\forall e\colon a \to b$ верно $f_a \leqslant f_b + w_{ab}$, то алгоритм A* достанет каждую вершину из кучи не более одного раза.

Доказательство. Рассмотрим вершину $a\colon d_a + f_a = \min$. Возьмём $\forall v$ и путь из неё в $a\colon\ v \to u \to \dots \to c \to b \to a$. Нужно доказать, что $d_a + f_a \geqslant (d_v + w_{vu} + \dots w_{cb} + w_{ba}) + f_a = F$.
Из неравенства $\triangle$ имеем $w_{ba} + f_a \geqslant f_b \wedge w_{cb} + f_b \geqslant f_a \wedge \dots \Rightarrow F \geqslant d_v + f_v \geqslant d_a + f_a$ ■

Следствие 3.5.2. Если для f верно неравенство треугольника, алгоритм можно остановить сразу после вынимания t из кучи.

Замечание 3.5.3. Заметим, что в доказательстве мы нигде не пользовались неотрицательностью весов. Для отрицательных обычно сложно найти функцию f с неравенством треугольника. Скоро мы введём технику потенциалов, узнав её, полезно ещё раз перечитать это место.

Замечание 3.5.4. "Оценка снизу" – лишь физический смысл функции f_v. Запускать алгоритм мы можем взяв любые вещественные числа.

$\forall f$, увеличив все f_v на константу, можно сделать $f_t = 0$. Будем рассматривать только такие f.

Теорема 3.5.5. Алгоритм A* на функции f, удоволетворяющей трём условиям: (a) неравенству треугольника, (b) $f_t = 0$, (c) $f_v \geqslant 0$ переберёт подмножество тех вершин, которые перебрала бы Дейкстра с `break`.

Доказательство. Дейкстра переберёт $v\colon d_v \leqslant d_t$.
A* переберёт $v\colon d_v + f_v \leqslant d_t + f_t = d_t \Rightarrow d_v \leqslant d_t - f_v \leqslant d_t$. ■

3.6. Флойд

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

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

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

$$\texttt{f[k+1,i,j] = min(f[k][i][j], f[k][i][k] + f[k][k][j])}.$$

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

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

3.6.1. Восстановление пути

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

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

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

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

3.6.2. Поиск отрицательного цикла

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

Lm 3.6.1. ($\exists$ отрицательный цикл, проходящий через i) $\Leftrightarrow$ (по окончании Флойда `d[i,i]<0`).

Lm 3.6.2. ($\nexists$ кратчайшего пути из a в b) $\Leftrightarrow$ ($\exists$ пути из a в i и из i в b, что `d[i,i]<0`).

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

```
int sum( int a, int b ) {
  if (a < 0 && b < 0 && a < -M - b)
    return -M;
  return a + b;
}
```

Лекция #4: Кратчайшие пути

15 марта

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

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

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

Теперь всё по порядку.

4.1. Алгоритм Форд-Беллмана

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

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

Соптимизируем память до линейной:

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

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

4.2. Выделение отрицательного цикла

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

Lm 4.2.1. $\forall$ итерации $\forall$ отрицательного цикла произойдёт релаксация хотя бы по одному ребру.

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

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

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

Lm 4.2.2. $\forall$ момент времени $\forall$ вершины v верно $d[p_v] + w[p_v, v] \leqslant d[v]$.

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

Следствие 4.2.3. После релаксации v произошла релаксация $p_v \Rightarrow d[p_v] + w[p_v, v] < d[v]$.

Lm 4.2.4. Откат по ссылкам p из вершины b зациклится.

Доказательство. Пусть не зациклился $\Rightarrow$ остановился в вершине s. При этом $p_s = -1 \Rightarrow d[s]$ не менялось $\Rightarrow d[s] = 0$. Обозначим вершины пути $s = v_1, v_2, \ldots, v_k = b$. Последовательно для всех рёбер пути используем неравенство из леммы 4.2.2, получаем $d[s] +$ `(вес пути)` $\leqslant d[b] \Leftrightarrow$ `(вес пути)` $\leqslant d[b]$. Но $d[b]$ уже обновился на n-й итерации, значит строго меньше веса любого пути из не более чем $(n-1)$ ребра. Противоречие. ■

Lm 4.2.5. Вес полученного цикла отрицательный.

Доказательство. Сложим по всем рёбрам цикла неравенство из леммы 4.2.2, получим $\sum d_i + \sum w_i \leqslant \sum d_i \Leftrightarrow \sum w_i \leqslant 0$. Чтобы получить строгую отрицательность рассмотрим y – последнюю вершину цикла, для которой менялось расстояние. Тогда для вершины цикла z: $p_z = y$ неравенство из леммы 4.2.2 строгое. ■

Из доказанных лемм следует:
Теорема 4.2.6. Алгоритм восстановления отрицательного цикла корректен.

4.3. Модификации Форд-Беллмана

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

4.3.1. Форд-Беллман с break

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

4.3.2. Форд-Беллман с очередью

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

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

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

• Добавляем очередь

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

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

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

4.3.3. Форд-Беллман с random shuffle

Сейчас мы оценим, сколько в худшем и лучшем случае работает Форд-Беллман с `break`.
Здесь и далее одна итерация – один раз перебрать за $\Theta(E)$ все рёбра графа.

Lm 4.3.1. На любом тесте в лучшем случае сделает 1 итерацию.

Доказательство. Рассмотрим дерево кратчайших путей. Пусть в массиве рёбер, который просматривает Форд-Беллман, все рёбра упорядочены от корня к листьям. Тогда при просмотре ребра $a \to b$ расстояние до вершины a по индукции уже посчитано верно. ■

Lm 4.3.2. На любом тесте Форд-Беллман сделает итераций не больше, чем глубина дерева кратчайших путей.

Lm 4.3.3. $\exists$ тест: Форд-Беллман в худшем случае сделает $V-1$ итерацию.

Доказательство. Возьмём граф, в котором дерево кратчайших путей – один путь дины $V-1$. Пусть в массиве рёбер, который просматривает Форд-Беллман, рёбра упорядочены от листьев к корню. Тогда после i итераций верно посчитано расстояние для ровно $i+1$ вершины. ■

Сделаем перед Форд-Беллманом `random_shuffle` массива рёбер. Получается псевдокод:

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

Lm 4.3.4. На любом тесте матожидание числа итераций не больше $\frac{1}{e-1} \approx \frac{1}{1.7}$.

Доказательство. Рассмотрим дерево кратчайших путей, пусть оно – путь. Пусть перед началом очередной итерации мы правильно знаем расстояние до первых i вершин пути. За следующую итерацию с вероятностью 1 мы узнаем расстояние до $(i+1)$-й вершины, с вероятностью $\frac{1}{2}$ до $(i+2)$-й вершины, с вероятностью $\frac{1}{k!}$ до $(i+k)$-й вершины: это произойдёт ровно в том случае, если k рёбер до этой вершины перебираются в том же порядке, что и лежат в пути. Порядков всего $k!$ с вероятностью $\frac{1}{k!}$ `random_shuffle` выберет нужный. Матожидание числа рёбер за одну фазу $E = \sum_k \Pr_k = 1 + \frac{1}{2} + \frac{1}{6} + \dots = e - 1$. Можно по индукции доказать, что если осталось найти расстояние до t вершин, то матожидание числа фаз $f_t \leqslant \frac{t}{E} + 1$. Если дерево – не путь, иногда матожидание числа рёбер за одну фазу будет больше, число фаз меньше. ■

4.4. Потенциалы Джонсона

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

Def 4.4.1. *Любой массив вещественных чисел p_v можно назвать потенциалами вершин. При этом потенциалы задают новые веса рёбер: $e\colon a \to b, w'_e = w_e + p_a - p_b$.*

Самое важное: любые потенциалы сохраняют кратчайшие пути.

Lm 4.4.2. $\forall s, t$ кратчайший путь на весах w_e перейдёт в кратчайший на весах w'_e

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

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

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

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

4.5. Цикл минимального среднего веса

Задача: найти цикл, у которого среднее арифметическое весов рёбер минимально.

Искомый вес цикла обозначим μ. Веса рёбер w_e.

Lm 4.5.1. Среди оптимальных ответов существует простой цикл.

Доказательство. Из оптимальных выберем цикл минимальный по числу рёбер. Пусть он не простой, представим его, как объединение двух меньших. У одного из меньших среднее арифметическое не больше. Противоречие $\Rightarrow$ посылка ложна $\Rightarrow$ простой. ■

Lm 4.5.2. Если все w_e уменьшить на x, μ тоже уменьшится на x.

• Алгоритм

Бинарный поиск по ответу, внутри нужно проверять "есть ли цикл среднего веса меньше x?"
$\Leftrightarrow$ "есть ли в графе с весами $(w_e - x)$ цикл среднего веса меньше 0"
$\Leftrightarrow$ "есть ли в графе с весами $(w_e - x)$ отрицательный цикл".
Умеем это проверять Форд-Беллманом за $\mathcal{O}(VE)$. Скоро научимся за $\mathcal{O}(E\sqrt{V}\log N)$.

Границы бинпоиска – минимальный и максимальный вес ребра.
Правую можно взять даже точнее – средний вес любого цикла (цикл умеем искать за $\mathcal{O}(E)$).
Если сейчас отрезок бинпоиска $[L, R)$, у нас уже точно есть цикл C веса меньше R.

Lm 4.5.3. $R - L \leqslant \frac{1}{V^2} \Rightarrow C$ – оптимален.

Доказательство. Рассмотрим любые два цикла, их средние веса $\frac{s_1}{k_1}$ и $\frac{s_2}{k_2}$. $(3 \leqslant k_1, k_2 \leqslant V)$.
Предположим, что $\frac{s_1}{k_1} \neq \frac{s_2}{k_2}$ и оценим снизу их разность: $|\frac{s_1}{k_1} - \frac{s_2}{k_2}| = \frac{|s_1k_2 - s_2k_1|}{k_1k_2} \geqslant \frac{1}{k_1k_2} \geqslant \frac{1}{V^2} \Rightarrow$
При $R - L \leqslant \frac{1}{V^2}$ на промежутке $[L, R)$ не может содержаться двух разных средних весов. ■

4.6. Алгоритм Карпа

Добавим фиктивную вершину s и рёбра веса 0 из s во все вершины.
Насчитаем за $\mathcal{O}(VE)$ динамику $d_{n,v}$ – вес минимального пути из s в v из ровно n рёбер.
Если для какой-то вершины пути нужной длины не существует, запишем бесконечность.

Теорема 4.6.1. Пусть $Q = \min_v \left(\max_{k=1..n-1} \frac{d_{n,v} - d_{k,v}}{n-k} \right)$. Считаем $+\infty - +\infty = +\infty$. Тогда $\mu = Q$.

Алгоритм заключается в замене бинпоика на формулу и поиске отрицательного цикла в графе с весами $(w_e - \mu - \frac{1}{V^2})$: в графе с весами $(w_e - \mu)$ есть нулевой, мы ещё чуть уменьшили веса.
Осталось доказать теорему ;-). Разобьём доказательство на несколько лемм.

Lm 4.6.2. Все w_e уменьшили на $x \Rightarrow Q$ уменьшится на x.

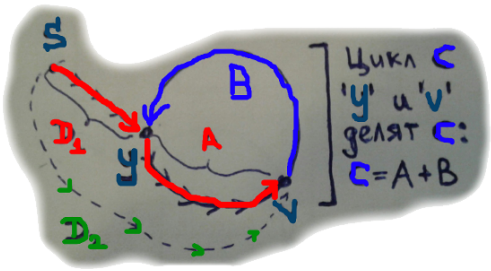
Доказательство. $d'_{n,v} = d_{n,v} + nx \Rightarrow \frac{d'_{n,v} - d'_{k,v}}{n-k} = \frac{d_{n,v} - d_{k,v} + (n-k)x}{n-k} = \frac{d_{n,v} - d_{k,v}}{n-k} - x.$ ■

Lm 4.6.3. Если доказать теорему для случая $\mu = 0$, то она верна в общем случае

Доказательство. Уменьшив все w_e на μ по 4.5.2 перейдём к случаю $\mu' = 0$ По теореме получаем $Q' = 0$. Увеличим обратно веса на μ, по лемме 4.6.2 получаем $Q = Q' + \mu = \mu$. ■

Lm 4.6.4. $(\mu = 0) \Rightarrow (Q = 0)$

Доказательство. Вместо сравнения $\max_{k=1..n-1} \frac{d_{n,v}-d_{k,v}}{n-k}$ с 0 достаточно сравнить с 0 максимум числителей $\max_{k=1..n-1}(d_{n,v} - d_{k,v}) = d_{n,v} - \min_{k=1..n-1} d_{k,v}$.
$\mu = 0 \Rightarrow$ нет отрицательных циклов $\Rightarrow \min_{k=1..n-1} d_{k,v}$ – вес кратчайшего пути.
Поскольку $d_{n,v}$ – вес какого-то пути, получаем $d_{n,v} - \min_{k=1..n-1} d_{k,v} \geqslant 0$.
Осталось предъявить вершину v, для которой $d_{n,v}$ – кратчайший путь от s до v. Для этого рассмотрим нулевой цикл C, любую вершину y на C, кратчайший путь из s в y. Пусть этот путь состоит из k рёбер и имеет вес D_1. Пройдём из y по циклу на $n-k$ рёбер вперёд (возможно прокружимся по циклу несколько раз). Остановились в вершине v, пройденный путь до v содержит n рёбер, имеет вес $D_1 + A$ (цикл C вершинами y и v разбился на две половины с весами A и B, при этом $A + B = 0$). Докажем что найденный путь кратчайший до v: от противного, пусть есть другой $D_2 < D_1 + A$, тогда продолжим его по циклу до y, получим $D_2 + B < D_1 + A + B = D_1$. Получили противоречие с минимальностью D_1. ■



4.7. (*) Алгоритм Гольдберга

Цель: придумать алгоритм поиска потенциалов, делающих все веса неотрицательными, работающий за $\mathcal{O}(EV^{1/2} \log N)$. В исходном графе веса целые в $[-N, \infty)$.
Для начала решим задачу для целых весов из $[-1, \infty)$.
Интересными будем называть вершины, в которые есть входящие рёбра веса -1.
Текущий граф будем обозначать G (веса меняются). Граф G, в котором остались только рёбра веса 0 и -1 будем обозначать G'. Теперь будем избавляться от интересных вершин, следя за тем, чтобы не появлялось новых отрицательных рёбер.

4.7.1. Решение за $\mathcal{O}(VE)$

Рассмотрим любую интересную вершину v и за $\mathcal{O}(E)$ сделаем её не интересной. Для начала попробуем просто уменьшить её потенциал на 1. При этом веса входящих рёбер увеличатся на 1, а исходящих уменьшатся на 1. Если бы не было исходящих рёбер веса $\leqslant 0$, это был бы успех... Тогда рассмотрим множество $A(v)$ – достижимые в G' из v. Если $A(v)$ содержит x: из x есть ребро в v веса -1, то мы нашли отрицательный цикл. Иначе уменьшим потенциал всех вершин из $A(v)$ на 1: рёбра внутри $A(v)$ не поменяли вес, все входящие в $A(v)$ увеличили вес на 1, исходящие уменьшили на 1. Новых отрицательных рёбер не появилось, вершина v перестала быть интересной.

4.7.2. Решение за $\mathcal{O}(EV^{1/2})$

Осталось научиться исправлять вершины пачками, а не по одной. Рассмотрим компоненты сильной связности в G', если хотя бы одна содержит ребро веса -1, то мы нашли отрицательный цикл. Иначе компоненту можно сжать в одну вершину. Теперь G' ацикличен, можно добавить фиктивную вершину s и из неё нулевые рёбра во все вершины, найти в G' за $\mathcal{O}(E)$ кратчайшие расстояния от s до всех вершин (динамика, поиск в глубину). Получили дерево кратчайших путей. Слоем B_d назовём все вершины, на расстоянии d от s ($d \leqslant 0$).

Lm 4.7.1. Пусть всего интересных вершин k. Или есть слой B_d, в котором $\sqrt{k}$ интересных вершин, или есть путь от s в лист дерева, на котором $\sqrt{k}$ интересных вершин.

Доказательство. Рассмотрим v: $d_v = \min$. Пусть $d_v \leqslant -\sqrt{k} \Rightarrow$ на пути до v хотя бы $\sqrt{k}$ раз было ребро -1, хотя бы столько интересных вершин. Иначе слоёв всего $\leqslant \sqrt{k}$, по принципу Дирихле хотя бы в одном $\geqslant \sqrt{k}$ вершин. ■

• **Решение для широкого слоя за $\mathcal{O}(E)$**

Рассмотрим любой слой B_d. Уменьшим на 1 потенциал всех вершин $A(B_d)$ (достижимые в G'). Докажем, что все вершины в B_d стали неинтересными. Рассмотрим ребро веса -1 из x в $v \in B_d$. Пусть $x \in A(B_d) \Rightarrow$ мы неверно нашли расстояние до v: мы считаем, что оно d, но есть путь $s \rightsquigarrow B_d \rightsquigarrow x \to v$ длины не более $d-1$. Значит $x \notin A(B_d), v \in A(B_d) \Rightarrow$ вес ребра увеличится.

• **Решение для длинного пути за $\mathcal{O}(E)$**

Занумеруем интересные вершины на пути **от конца** к s: $v_1, v_2, v_3, \ldots$.
Исправим вершину v_1, уменьшив на 1 потенциал $A(v_1)$. Теперь, исправляя вершину v_2, заметим, что $A(v_1) \subset A(v_2) \Rightarrow$ мы можем ходить только по непосещённым вершинам.
Казалось бы достаточно, запуская dfs из v_2, пропускать помеченные dfs-ом от v_1 вершины.
Но есть ещё ребра, исходящие из $A(v_1)$ веса 1. После изменения потенциала $A(v_1)$ их вес стал 0, поэтому dfs из v_2 должен по ним пройти. Чтобы быстро находить все такие рёбра, заведём структуру данных, в которую будем добавлять **все** пройденные рёбра.
Структура данных будет уметь делать три вещи: добавить ребро, уменьшить вес всех рёбер на 1, перебрать все рёбра веса 0 за их количество.

```
1 vector<int> edges[N];
2 int zero;
3 void add(int i, int w) { edges[zero + w].push_back(i); }
4 void subtract()        { zero += 1; }
5 vector<int> get()      { return edges[zero]; }
```

Хотим найти $A(v_i) \Rightarrow$ запускаем dfs от v_i и от всех концов рёбер, возвращённых `get()`.
После того, как нашли $A(v_i)$ вызываем `subtact()`.
Итого суммарное время поиска $A(v_1), A(v_2), A(v_3), \ldots$ равно $\mathcal{O}(E)$.

4.7.3. Общий случай

Также, как мы от графа с весами рёбер $\geqslant -1$ переходили к весам $\geqslant 0$,
можно от весов рёбер $\geqslant -(2k+1)$ перейти к весам $\geqslant -k$.
Выше мы делили рёбра графа на 3 группы: $\{-1, 0, [0, +\infty)\}$.
Для $-(2k+1)$ группы будут $\{[-(2k+1), -k), [-k, 0], [0, +\infty)\}$.

Лекция #5: DSU и MST

22 марта

5.1. DSU: Система Непересекающихся Множеств

Цель: придумать структуру данных, поддерживающую операции:

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

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

5.1.1. Решения списками

Напишем максимально простую реализацию, получится примерно следующее:

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

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

Теорема 5.1.1. Тогда суммарное число операций всех `for` во всех `join` не более $n \log n$.

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

Следствие 5.1.2. Суммарное время работы m `join`-ов $\mathcal{O}(m + n \log n)$.

5.1.2. Решения деревьями

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

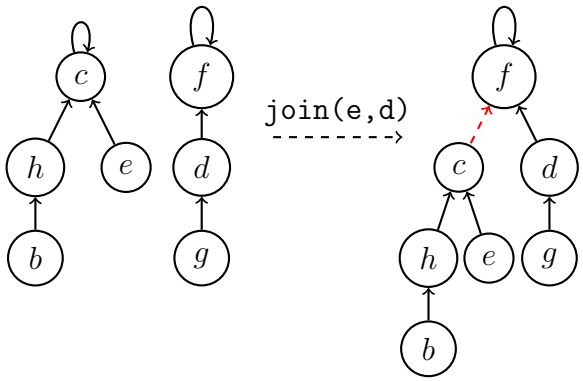
• **Ранговая сжатие путей:** все рёбра, по которым пройдёт `get` перенаправить в корень.

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

Def 5.1.3. *Ранг вершины – глубина её поддерева, если бы не было сжатия путей. Сжатие путей не меняет ранги. Ранг листа равен нулю.*

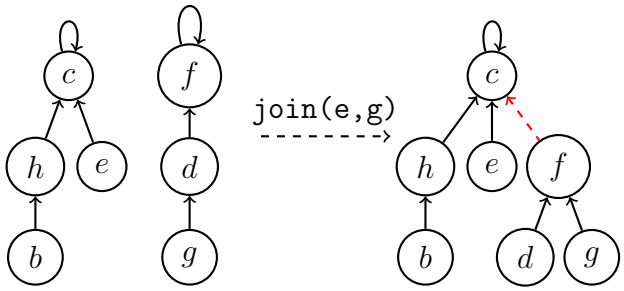
Вот базовая реализация без ранговой эвристики и без сжатия путей:

```
1  int p[n]; // для каждого элемента храним отца
2  void init( int n ) {
3    for (int i = 0; i < n; i++)
4      p[i] = i; // каждый элемент - сам себе корень
5  }
6  int get( int i ) {
7    // подняться до корня, вернуть корень
8    return p[i] == i ? i : get(p[i]);
9  }
10 void join( int a, int b ) {
11   a = get(a), b = get(b); // перешли к корням
12   p[a] = b; // подвесили один корень за другой
13 }
```



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

```
1  int p[n], rank[n];
2  void init( int n ) {
3    for (int i = 0; i < n; i++)
4      p[i] = i, rank[i] = 0;
5  }
6  int get( int i ) {
7    return p[i] == i ? i : (p[i] = get(p[i]));
8  }
9  void join( int a, int b ) {
10   a = get(a), b = get(b);
11   if (rank[a] > rank[b]) swap(a, b);
12   if (rank[a] == rank[b]) rank[b]++;
13   p[a] = b; // b - вершин с меньшим рангом
14 }
```



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

Теорема 5.1.4. Ранговая эвристика без сжатия путей даст $\mathcal{O}(\log n)$ на запрос.

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

Lm 5.1.5. Размер дерева ранга k хотя бы 2^k.

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

Напомним пару определений:

Def 5.1.6. *Пусть v – вершина, p – её отец, size – размер поддерева.*

Ребро $p \to v$ называется

$$\begin{array}{ll} \textit{Тяжёлым} & \textit{если } size(v) > \frac{1}{2} size(p) \\ \textit{Лёгким} & \textit{если } size(v) \leqslant \frac{1}{2} size(p) \end{array}$$

Теорема 5.1.7. Сжатие путей без ранговой эвристики даст следующую оценку:
m запросов `get` в сумме пройдут не более $m + (m+n)\log_2 n$ рёбер.

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

5.1.3. Оценки $\mathcal{O}(\log^* n)$ и лучше

Теорема 5.1.8. Сжатие путей вместе с ранговой эвристики дадут следующую оценку: m запросов `get` в сумме пройдут не более $m(1 + 2\log^* n) + \Theta(n)$ рёбер.

Доказательство. Обозначим $x = 1.7$, заметим $x^x \geqslant 2$. Назовём ребро $v \to p$ крутым, если $rank_p \geqslant x^{rank_v}$ (при подъёме по этому ребру ранг сильно возрастает). Обозначим время работы i-го `get`, как $t_i = 1 + a_i + b_i =$ одно верхнее ребро, a_i крутых и b_i не крутых.
$\forall v$ имеем $rank_{p_v} > rank_v \Rightarrow a_i \leqslant 2\log^* n$. Теперь исследуем жизненный цикл не крутых рёбер. $\forall v$, если v не корень, $rank_v$ уже не будет меняться. При сжатии путей у всех рёбер кроме самого верхнего возрастёт разность $(rank_{p_v} - rank_v) \Rightarrow$ каждое некрутое ребро уже через x^{rank_v} проходов по нему вверх навсегда станет крутым $\Rightarrow \sum_i b_i \leqslant \sum_v x^{rank_v} = \sum_r count(r)x^r$, где $count(r)$ – число вершин с рангом r. Поскольку вершины с рангом r имеют поддеревья размера хотя бы 2^r, и эти поддеревья не пересекаются, $count(r) \leqslant \frac{n}{2^r}$. Итого: $\sum_i b_i \leqslant \sum_r \frac{n}{2^r}x^r = n\sum_r \left(\frac{x}{2}\right)^r = \Theta(n)$ ■

Теорема 5.1.9. Сжатие путей вместе с ранговой эвристики дадут следующую оценку: m запросов `get` в сумме пройдут не более $\Theta(m + n\log^{**} n)$ рёбер.

Доказательство. Крутое ребро, которое k раз поучаствовало в сжатии путей в качестве не самого верхнего крутого, назовём *ребром крутизны* k. Крутизна любого ребра не более $\log^* n$. При сжатии не самого верхнего крутого ребра, его крутизна растёт $\Rightarrow \sum b_i \leqslant n\log^* n$. ■

Теорема 5.1.10. Сжатие путей вместе с ранговой эвристики дадут следующую оценку: m запросов `get` в сумме пройдут не более $\Theta(m\log^{**} n) + \Theta(n)$ рёбер.

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

$$rank_{p_v} \geqslant x^{x^{x^{\cdots^{rank_v}}}}$$

Здесь высота степенной башни – $rank_v$, поэтому из $rank_v > 2\log^* n$ следует $rank_{p_v} > n \Rightarrow$ "Высота башни с вершиной $rank = n$ не больше 1+"высота башни с вершиной $rank = 2\log^* n$". Поэтому число дважды крутых рёбер на пути не больше чем "сколько раз нужно применить $\log^*$ к n, чтобы получилась единица" = $\log^{**} n$. ■

5.2. MST: Минимальное Остовное Дерево

Def 5.2.1. *Остовное дерево связного графа – подмножество его рёбер, являющееся деревом.*

Def 5.2.2. *Вес остова – сумма весов рёбер.*

Задача: дан граф, найти остов минимальной стоимости.
Задача поиска максимального разреза равносильно, получается домножением весов на -1.

5.2.1. Алгоритм Краскала

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

5.2.2. Алгоритм Прима

Также, как алгоритм Дейкстры в каждый момент времени поддерживал множество вершин A и дерево кратчайших путей для вершин из A, алгоритм Прима будет в каждый момент времени поддерживать минимальное остовное дерево на вершинах A. Переход: добавить вершину v: ребро $A \to v$ имеет минимальный вес. Для того, чтобы за то же, что и в Дейкстре время выбирать вершину v, поддерживаем $d_v =$ минимальный вес ребра $A \to v$ и храним кучу всех d_v. Времена работы: $\mathcal{O}(E + V \log V)$, $\mathcal{O}(V^2)$.

5.2.3. Алгоритм Борувки

Вес ребра w_e поменяем на пару $\langle w_e, e\rangle$. Теперь все веса различны.

Фаза алгоритма: для каждой вершины выберем минимальное по весу ребро. Выбранные рёбра не образуют циклов (доказывается от противного, пользуемся тем, что веса различны). Добавим все выбранные рёбра в остов. Сожмём компоненты связности. Удалим кратные рёбра.

Пока граф не сожмётся в одну вершину выполняем очередную фазу.

Удалять кратные рёбра будем сортировкой рёбер.
Рёбра – пары чисел от 1 до V, поэтому цифровая сортировка справится за $\mathcal{O}(V + E)$.

Lm 5.2.3. Время работы алгоритма Борувки $\mathcal{O}((V + E) \log V)$.

Доказательство. Фаза работает за $V + E$. Число вершин уменьшится в два раза. ■

Lm 5.2.4. Время работы алгоритма Борувки $\mathcal{O}(E + V^2)$.

Доказательство. $\mathcal{O}(V + E)$ нужно на первое удаление кратных рёбер.
Теперь всегда $E \leqslant V^2 \Rightarrow$ алгоритм работает за $\mathcal{O}(V^2 + (\frac{V}{2})^2 + (\frac{V}{4})^2 + \dots) = \mathcal{O}(V^2)$. ■

Lm 5.2.5. Время работы алгоритма Борувки $\mathcal{O}(E + E\lceil \log V^2/E \rceil)$.

Доказательство. За $\lceil \frac{1}{2} \log V^2/E \rceil$ фаз V^2 станет не больше исходного E.
По предыдущей лемме оставшаяся часть алгоритма отработает за $\mathcal{O}(E)$. ■

5.2.4. Лемма о разрезе

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

Lm 5.2.6. Для любого разбиения множества вершин $V = A \sqcup B$, существует минимальный остов, содержащий e – минимальное по весу ребро, проходящее через разрез $\langle A, B\rangle$.

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

Если уже известно подмножество рёбер минимального остова, они задают компоненты связности, мы можем их сжать в вершины и в полученном графе применить лемму о разрезе. Итого:

Следствие 5.2.7. X – подмножество рёбер некого минимального остова. Зафиксируем разрез $V = A \sqcup B$ такой, что все рёбра X не пересекают разрез. Тогда $\exists$ минимальный остов, содержащий $X \cup \{e\}$, где e – минимальное по весу ребро, проходящее через разрез $\langle A, B\rangle$.

Применим лемму несколько раз, докажем описанные выше алгоритмы.

• Доказательство алгоритма Прима.

Разрез – текущее дерево и дополнение.

• Доказательство алгоритма Краскала.

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

• Доказательство алгоритма Борувки.

Лемма о разрезе говорит, что каждое из рёбер, выбранных Борувкой добавить можно. Тонкость в том, что лемма о разрезе не говорит, можно ли их добавить одновременно. В каждой компоненте связности добавляемые рёбра будут образовывать дерево, рёбра можно добавлять в таком порядке, что каждое следующее создаёт лист, тогда разрез – лист и дополнение к нему.

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

TODO

5.3.1. Введение обратной функции Аккермана

Def 5.3.1. *Функция Аккермана*

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

Lm 5.3.2. $A_k(n) > n$

Lm 5.3.3. $f(t) = A_t(1)$ монотонно возрастает

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

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

Def 5.3.4. *Обратная функция Аккермана* – $\alpha(t) = \min\{k \mid A_k(1) \geqslant t\}$

Посчитаем несколько первых значений функции α
$A_0(1) = 1 + 1 = 2$
$A_1(1) = 2 \cdot 1 + 1 = 3$
$A_2(1) = 2^2 * 2 - 1 = 7$
$A_3(1) = A_2(A_2(1)) = A_2(7) = 2^8 8 - 1 = 2047$
$A_4(1) = A_3(A_3(1)) = A_3(2047) > 2^{2^{2^{\cdots^{2047}}}}$ (башня высоты 2048).
Получившаяся оценка снизу на $A_4(1)$ – невероятно большое число, превышающее число атомов в наблюдаемой части Вселенной. Поэтому обычно предполагают, что $\alpha(t) \leqslant 4$.

5.3.2. Доказательство

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

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

Def 5.3.5. *Порядок элемента.* $\forall$ *обычного элемента* e $\quad level(e) = \max\{k \mid r_{p_e} \geqslant A_k(r_e)\}$.
Def 5.3.6. *Итерация порядка.* $\forall$ *обычного элемента* e $\quad iter(e) = \max\{i \mid r_{p_e} \geqslant A^{(i)}_{level(e)}(r_e)\}$.

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

Lm 5.3.7. $\forall$ обычного элемента e верно, что $0 \leqslant level(e) < \alpha(n)$.

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

Lm 5.3.8. $\forall$ обычного элемента e верно, что $1 \leqslant iter(e) \leqslant r_e$.

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

Будем доказывать время работы СНМ методом амортизационного анализа.

Def 5.3.9. *Потенциал элемента. Определим для любого элемента* e *величину* $\varphi(e)$.
$$\varphi(e) = \begin{cases} \alpha(n) \cdot r_e, \text{ если } e \text{ – представитель своего множества} \\ (\alpha(n) - level(e)) \cdot r_e - iter(e), \text{ иначе} \end{cases}$$

Def 5.3.10. *Потенциал. Возьмём* $\varphi = \sum\limits_e \varphi(e)$.

Обозначим через φ_t потенциал после первых t операций.

Lm 5.3.11. $\varphi_0 = 0$, $\forall i \varphi_i \geq 0$.

Доказательство. В начальный момент времени все элементы являются представителями своих одноэлементных множеств $\Rightarrow \forall e\colon \varphi(e) = \alpha(n) \cdot r_e = \alpha(n) \cdot 0 = 0 \Rightarrow \varphi_0 = \sum_e \varphi(e) = \sum_e 0 = 0$.
$r_e \geqslant 0 \Rightarrow \alpha(n) \cdot r_e \geqslant 0$ (доказали, что $\varphi_i \geqslant 0$ для любого представителя множества).
$level(e) < \alpha(n) \Rightarrow (\alpha(n) - level(e)) \cdot r_e \geqslant r_e$, осталось вспомнить, что $iter(e) \geqslant r_e$. ■

Теорема 5.3.12. Для операции типа *join* амортизированное время $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$.

Доказательство. Так как φ_i зависит от $\varphi(e)$, который зависит от r_e, $level(e)$ и $iter(e)$, которые зависят от r_{p_e}, то потенциал изменится только у тех элементов, у которых изменился ранг или ранг предка, а также у тех элементов, которые перестали быть представителями своего множества. Так как операция $join(e_1, e_2)$ может поменять ранг одного элемента, пусть e_1 (то есть он стал представителем объединения множеств), а элемент e_2 перестал быть представителем своего множества, то $\varphi(e)$ мог поменяться у e_1, e_2 и тех элементов, у которых $p_e = e_1$ (обозначим множество таких элементов за S).
Обозначим за $\varphi'(e)$, $level'(e)$ и $iter'(e)$ потенциал, порядок и характеристику порядка элемента e после операции *join* соответственно. Тогда $\Delta\varphi(e_2) = \varphi'(e_2) - \varphi(e_2) = (\alpha(n) - level(e_2)) \cdot r_{e_2} - iter(e_2) - (\alpha(n) \cdot r_{e_2}) = -level(e_2) \cdot r_{e_2} - iter(e_2)$, так как $level$ неотрицательный, а $iter$ положительный (леммы 6.3 и 6.4), то $\Delta\varphi(e_2) < 0$.
Если ранг e_1 остался прежним, то $\forall e \in S : \Delta\varphi(e) = 0$, так как r_e, $level(e)$ и $iter(e)$ остались прежними.
Если ранг e_1 поменялся, то если для некоторого элемента $e \in S$ $level(e)$ остался прежним, то $iter(e)$ не уменьшился, так как иначе бы это означало, что r_{p_e} уменьшился, что противоречит тому, что он увеличился. Если $iter(e)$ не изменился, то $\varphi(e)$ остался прежним, так как величины от которых он зависит не изменились. Если $iter(e)$ уменьшился, то $\Delta\varphi(e) = \varphi'(e) - \varphi(e) = (\alpha(n) - level'(e)) \cdot r_e - iter'(e) - ((\alpha(n) - level(e)) \cdot r_e - iter(e)) = (\alpha(n) - level(e)) \cdot r_e - iter'(e) - (\alpha(n) - level(e)) \cdot r_e + iter(e) = iter(e) - iter'(e) < 0$.
Если ранг e_1 поменялся, то если для некоторого элемента $e \in S$ $level(e)$ тоже поменялся, то $\Delta\varphi(e) = \varphi'(e) - \varphi(e) = (\alpha(n) - level'(e)) \cdot r_e - iter'(e) - ((\alpha(n) - level(e)) \cdot r_e - iter(e)) = (\alpha(n) - level'(e)) \cdot r_e - iter'(e) - (\alpha(n) - level(e)) \cdot r_e + iter(e) = r_e \cdot (level(e) - level'(e)) + iter(e) - iter'(e) \leq -r_e + iter(e) - iter'(e)$, так как $iter(e) \leq r_e$ (лемма 6.4), то $\Delta\varphi(e) = -r_e + iter(e) - iter'(e) \leq -iter'(e) < 0$, так как $iter'(e) > 1$ (лемма 6.4).
Получили, что изменение потенциала у всех элементов, кроме e_1 неположительное.
Если r_{e_1} не изменился, то $\Delta\varphi(e_1) = 0$, иначе
$\Delta\varphi(e_1) = \varphi'(e_1) - \varphi(e_1) = (r_{e_1}+1)\alpha(n) - r_{e_1}\alpha(n) = \alpha(n)$.
Получили $\Delta\varphi = \sum_e \Delta\varphi(e) \leq \Delta\varphi(e_1) \leq \alpha(n)$. Но так как $t_i = \mathcal{O}(1)$, то $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$. ■

Теорема 5.3.13. Для операции типа *get* амортизированное время $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$.

Доказательство. Аналогично рассуждениям из теоремы 6.8, потенциал меняется только у элементов на пути операции *get*, причем потенциал не увеличивается. i-ая операция *get* работает пропорционально $w_i + x_i + 1$:

1. w_i означает элементы e на пути операции *get*, у которых $\Delta\varphi(e) < 0$.
2. x_i означает элементы e на пути операции *get* (кроме представителя множества), у которых $\Delta\varphi(e) = 0$.
3. 1. Некоторое константное количество действий. Сюда включено посещение представителя множества.

Если порядок некоторого элемента e или характеристика порядка изменились, то они увеличились, так как величина r_{p_e} увеличилась, но тогда аналогично рассуждениям из теоремы 6.8 величина $\Delta\varphi(e) < 0$.
Пусть $x_i > \alpha(n)$. Тогда так как количество различных порядков $\alpha(n)$ (лемма 6.3), то $\exists e_1, e_2$

на пути операции *get*, такие, что $level(e_1) = level(e_2)$. Пусть e_2 находится ближе к представителю множества. Обозначим представителя множества за R, а $\varphi'(e)$, $level'(e)$ и $iter'(e)$ аналогично теореме 6.8. Тогда $r_R \geq r_{p_{e_2}} \geq A_{level(e_2)}(r_{e_2}) = A_{level(e_1)}(r_{e_2}) \geq A_{level(e_1)}(r_{p_{e_1}})$ значит если $level'(e_1) = level(e_1)$, то $iter'(e_1) > iter(e_1)$, так как в этом случае $r_{p_{e_1}} \geq A^{(iter(e_1))}_{level(e_1)}(r_{e_1}) = A^{(iter(e_1))}_{level'(e_1)}(r_{e_1}) \Rightarrow r_R \geq A_{level(e_1)}(r_{p_{e_1}}) = A_{level'(e_1)}(r_{p_{e_1}}) \geq A_{level'(e_1)}(A^{(iter(e_1))}_{level'(e_1)}(r_{e_1})) = A^{(iter(e_1)+1)}_{level(e_1)}(r_{e_1})$. Получаем противоречие, так как у элемента e_1 величина $\Delta\varphi(e_1) < 0$. Значит $x_i \leq \alpha(n)$.
Получаем, что величина a_i пропорциональна $w_i + x_i + 1 + \Delta\varphi$, но сумма каждого элемента e, учтенного в w_i, со своим $\Delta\varphi(e)$ неположительна, а $x_i \leq \alpha(n)$, значит $a_i = \mathcal{O}(\alpha(n))$. ■

Теорема 5.3.14. Суммарное время работы m произвольных операций с СНМ – $\mathcal{O}(m \cdot \alpha(n))$.

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

Лекция #6: Жадность и приближённые алгоритмы

28 марта

6.1. Приближённые алгоритмы

Как мы знаем, некоторые задачи NP-трудны, вряд ли их возможно решить за полином. Но на практике решать их всё равно нужно. В таких случаях часто ищут приближённое решение, причём ищут за полиномиальное или даже линейное время.

Def 6.1.1. *Алоритм M называется α-оптимальным (α-OPT) для задачи P, если $\forall$ входе для P, ответ выдаваемый M не более чем в α раз хуже оптимального.*

6.2. Коммивояжёр

Задача коммивояжёра заключается в поиске гамильтонова цикла минимального веса.
Решим задачу в случае полного графа с неравенством $\triangle$. Любой граф можно сделать полным, добавив вместо отсутствия ребра ребро веса $+\infty$. Наличие неравенства $\triangle$ важно.

Lm 6.2.1. $\forall C > 1$ $\nexists$ полиномиального C-OPT алгоритма для задачи коммивояжёра.

Доказательство. Если такой существует, то он в частности ищет гамильтонов путь. ■

6.2.1. 2-OPT через MST

Оптимальный маршрут коммивояжёра – какое-то остовное дерево + одно ребро $\Rightarrow$ OPT $\leqslant$ MST. Возьмём MST графа. Каждое ребро заменим на два ориентированных (туда и обратно), получили ориентированный эйлеров граф (число входящих равно числу исходящих). Возьмём эйлеров обход, его вес в два раза больше MST. Если в полученном пути некоторая вершина встречается второй раз, удалим её второе вхождение. По нер-ву $\triangle$ вес пути только уменьшится. Получили маршрут коммивояжёра веса $\leqslant 2 \cdot$ MST.

6.2.2. 1.5-OPT через MST (Кристофилдс)

Чтобы MST превратить в эйлеров граф, избавимся от вершин нечётной степени.
Возьмём совершенное паросочетание минимального веса на вершинах нечётной степени, добавим его к MST, полученный граф назовём G. По нер-ву $\triangle$ оптимальный маршрут коммивояжёра, построенный только на нечётных вершинах, $\leqslant$ OPT.
Его можно разделить на два паросочетания – чётные и нечётные рёбра.
Оба из них не меньше минимального $\Rightarrow$ минимальное $\leqslant \frac{1}{2}$OPT $\Rightarrow w(G) \leqslant$ MST$+\frac{1}{2}$OPT $\leqslant \frac{3}{2}$OPT.

Lm 6.2.2. $\exists$ полиномиальный алгоритм для поиска паросочетания минимального веса в произвольном графе. Без доказательства.

6.3. Хаффман

Для начала заметим печальный факт: идеального архиватора не существует.

Lm 6.3.1. $\nexists$ архиватора f и целого n: $\forall x$ $(|x| = n \Rightarrow f(x) < n)$

Доказательство. Входов 2^n, выходов $1+2+\cdots+2^{n-1}$ по принципу Дирихле f – не инъекция. ■

Алгоритм Хаффман – это наиболее известный алгоритм сжатия текста.
Хотим придумать безпрефиксные коды, минимизирующие величину

$$F = \sum_i len_i cnt_i$$

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

• Алгоритм построения беспрефиксных кодов

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

• Реализация

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

Теорема 6.3.2. Алгоритм корректен.

Доказательство. Если упорядочить символы: $len_i \searrow$, то $cnt_i \nearrow$, иначе можно поменять местами два кода и уменьшить F. У каждой вершины дерева ровно два ребёнка (если ребёнок только один, код можно было бы укоротить). Значит, существует два брата, две самых глубоких вершины, и это ровно те символы, у которых cnt минимален. Осталось сделать переход по индукции к задаче того же вида: заменим два самых глубоких символа-брата на их отца, и будем искать код отца. Длина кода отца домножается как раз на сумму частот братьев. ■

6.3.1. Хранение кодов

Есть несколько способов сохранить коды.

1. Сохранить не коды, а частоты. Декодер строит по ним коды так же, как кодер.
2. Рекурсивный код дерева: если лист, пишем бит 0 и символ, иначе пишем бит 1.
   Длина = $2 + \lceil \log \Sigma \rceil$ бит на каждый ненулевой символ.
3. Сделать dump памяти = `sizeof(Node) * (2Σ - 1)` байт.

Какой бы способ мы не выбрали, можно попробовать рекурсивно сжать коды Хаффманом. Чтобы рекурсия не была бесконечной добавим в код один бит – правда ли, что сжатый текст меньше. Если этот бит ноль, вместо кода запишем исходный текст.

6.4. Компаратор и сортировки

TODO

Лекция #7: Центроидная декомпозиция

12 апреля

Старый конспект на ту же тему

7.1. Введение и построение

Пусть дано дерево с весами на рёбрах или в вершинах. Решим две простых задачи:

1. Построить структуру данных, которая умеет за $\mathcal{O}(1)$ говорить минимум на пути от любой вершины до корня.
2. Построить структуру данных, которая умеет за $\mathcal{O}(1)$ говорить минимум на любом пути, проходящем через корень.

Для решения первой задачи поиском в глубину по дереву предподсчитаем все минимумы. Вторую задачу сведём к первой, разделив корнем путь на две половины.

Def 7.1.1. *Центроид – вершина, при удалении которой, размеры всех компонент* $\leqslant \frac{n}{2}$.

С помощью *центроидной декомпозиции* мы сможем поиск минимума на любом пути свести к уже решённой задаче "поиск минимума на пути, проходящем через корень дерева". Итак: в дереве мы уже умеем искать центроид. Все пути, проходящие через центроид мы уже умеем обработать. Что делать с остальными путями? После удаления центроида дерево распадается на компоненты связности. Каждый такой путь целиком лежит в одной из компонент. Поэтому рекурсивно для каждой из компонент построим такую же структуру данных: найдём центроид, от него сделаем предподсчёт поиском в глубину, удалим центроид, запустимся рекурсивно от образовавшихся компонент связности. Заметим, что в итоге каждая вершина ровно один раз будет найдена, как центроид.

У каждого центроида v есть центроид-предок p_v в дереве рекурсии и компонента связности $C(v)$, в которой он был найден, как центроид. Глубиной центроида d_v будем называть уровень глубины рекурсии, на котором вершина была найдена, как центр.

Def 7.1.2. *Центроидной декомпозицией будем называть два массива –* p_v *и* d_v.
Первый из массивов задаёт так называемое "дерево центроидной декомпозиции".

Замечание 7.1.3. Зная только глубины d_v можно восстановить компоненты $C(v)$. Для этого нужно от центра запустить dfs, который проходит только по вершинам большей глубины. Поэтому $C(v)$ хранить **не нужно**.

Lm 7.1.4. Для центроидной декомпозиции нужно $\Theta(n)$ памяти.

Lm 7.1.5. Глубина дерева центроидной декомпозиции не более $\log_2 n$.

Lm 7.1.6. Суммарный размер компонент $\mathcal{O}(n \log n)$.

Следствие 7.1.7. В задаче поиска min пути суммарный размер предподсчёта $\mathcal{O}(n \log n)$.

Вот мы нашли минимум, куда его сохранить?
Чтобы для $u \in C(v)$ сохранить минимум на пути $u \to v$, можно записать его в ячейку $f[d_v, u]$. Заметим, что для каждой пары $\langle v, u \in C_v \rangle$ ячейка уникальна.

Чтобы ответить на запрос "минимум на пути $a \to b$", нужно найти центр v:
$a, b \in C_v \wedge v \in path[a..b]$ и вернуть $\min(f[d_v, a], f[d_v, b])$. Чтобы найти такой центр v рассмотрим пути от a и b вверх в дереве центроидной декомпозиции и возьмём наименьшего общего предка a и b – это самый нижний такой центр, что $a, b \in C_v$.

7.2. Реализация

Старый конспект на ту же тему

7.3. Решение задач

TODO

Лекция #8: BST и AVL

19 апреля

8.1. BST

Это начало большого разговора о структурах данных для хранения множества значений (ключей), которые умеют добавлять/удалять/искать значения и кое-что ещё.

Def 8.1.1. *BST – binary search tree (бинарное дерево поиска). В каждой вершине ключ x.*
Левое поддерево содержит строго меньшие ключи.
Правое поддерево содержит строго большие ключи.

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

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

Относительно вершины v определим:
- `v->l` (`left`) – её левый сын
- `v->r` (`right`) – её правый сын
- `v->p` (`parent`) – её отец

По умолчанию будем считать, что у вершины есть только `v->l` и `v->r`. Отсутствие сына будем обозначать нулевым указателем. Если мы не пользуемся отцом, то для экономии памяти и времени, хранить и поддерживать его не будем.

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

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

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

```
13        v = v->p;
14      return v->p;
15    }
16  }
```

8.2. Операции в BST, введение в персистентность

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

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

Операция удаления (`del`). Сперва сделаем `find`. Если у вершины не более одного ребёнка, её очень просто удалить – скажем, что отец вершины не мы, а наш ребёнок. Если у вершины `v` два ребёнка, то перейдём в `next(v)`, сделаем `swap(v->x, next(v)->x)` и удалим `next(v)`. Заметим, что у `next(v)` точно нет левого ребёнка, т.к. чтобы её найти мы спустились до упора влево.

Все описанные операции: `add(x)`, `del(x)`, `find(x)`, `next(v)` работают за глубину дерева.

Кроме описанных в домашнем задании вам предстоит получить `Node* lower_bound(x)` работающий также за глубину дерева.

Def 8.2.1. *Если* $\exists C$: *в любой момент времени высота дерева не более* $C \log n$, *где* n *число вершин в дереве, будем называть дерево сбалансированным. Проще говоря, глубина –* $\mathcal{O}(\log n)$.

Перед тем, как изучить примеры сбалансированных деревьев, рассмотрим подробно возможные реализации `add`.

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

Более короткая версия, использующая “указатель на указатель”:

```
1  void add(Node* &v, int x) {
2    if (!v)
3      v = new Node(x);
4    else
5      add(x < v->x ? v->l : v->r, x);
6  }
```

А вот так называемая “персистентная” версия:

```
1  Node* add(Node* v, int x) {
2    if (!v)
3      return new Node(x); // дерево из одной вершины с ключом x
4    else if (x < v->x)
5      return new Node(v->x, add(v->l, x), v->r);
6    else
```

```
7       return new Node(v->x, v->l, add(v->r, x));
8   }
```

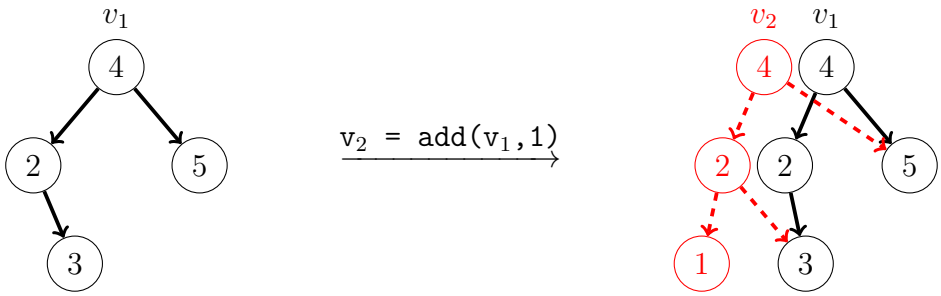
Эта версия прекрасна тем, что `struct Node` обладает свойством immutable.
В итоге можно писать так: `Node *root2 = add(root1, x)`,
после чего у вас есть два дерева – старое с корнем в `root1` и новое с корнем в `root2`.

Def 8.2.2. *Персистентной называется структура данных, в которой любая операция изменения создаёт новую версию структуры данных, при этом старая версия остаётся валидной для использования.*

По аналогии с `add` операцию `del` также можно сделать персистентной. Минусы этого подхода: если раньше операция `add`/`del` создавала 1 вершину, теперь создаётся `height(root)` новых вершин, т.е. даже в случае сбалансированного дерева нужно $\mathcal{O}(\log n)$ памяти. Плюсы персистентности: представьте, что на вашем жёстком диске все операции выполняются "персистентно", тогда в любой момент времени вы можете обратиться к любому состоянию из прошлого.

Замечание 8.2.3. Персистентная версия добавления не сможет пересчитать отцов ⇒
в персистентных деревьях отцами не пользуются.

Замечание 8.2.4. Каждая отдельная версия персистентного дерева – дерево.
Все версии в совокупности образуют ацикличный граф.



В конце этого раздела уделим время рекламе `Node**`. Полюбуйтесь на `find`, `del`, `add`.

```
1   Node** find(Node** root, int x) {
2     while (*root && (*root)->x != x)
3       root = (x < (*root)->x ? &(*root)->l : &(*root)->r);
4     return root;
5   }
6   bool isIn(Node *root, int x) {
7     return *find(&root, x) != 0;
8   }
9   void add(Node** root, int x) { // предполагает, что x точно нет в дереве
10    *find(root, x) = new Node(x);
11  }
12  void del(Node **root, int x) { // предполагает, что x точно есть в дереве
13    Node **v = find(root, x);
14    if (!(*v)->r) {
15      *v = (*v)->l; // поддепили левого сына v к её отцу
16      return;
17    }
18    Node **p = &(*v)->r;
19    while ((*p)->l)
20      p = &(*p)->l;
```

```
21    swap((*v)->x, (*p)->x);
22    *p = (*p)->r; // подцепили правого сына p к её отцу
23  }
```

8.3. AVL

Def 8.3.1. *Высота поддерева – максимальное из расстояний от корня до листа.*

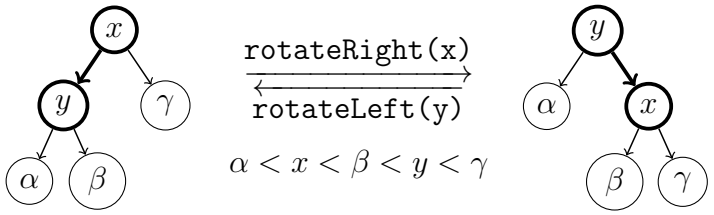
Def 8.3.2. *BST называют AVL-деревом, если оно удовлетворяет AVL-инварианту: в каждой вершине разность высот детей не более* 1.

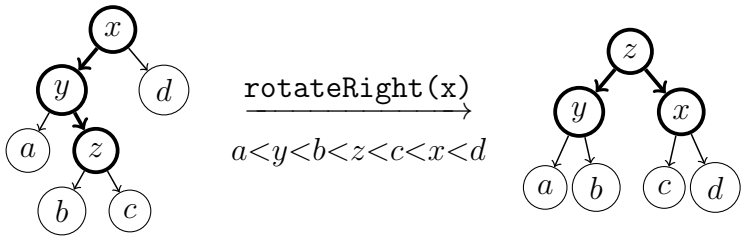
Добавление в AVL-дерево делается также, как в обычное BST. На обратном ходе рекурсии происходит перебалансировка: дети по индукции являются корректными AVL-деревьями, но разность их высот может быть два.

Пусть сейчас $h(v) = x + 3,\ h(v.l) = x + 2,\ h(v.r) = x \Rightarrow$ добавление происходило именно в $v.l$.
Есть два подслучая:

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

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





Высота вершины v и до добавления, и после *добавления и вращения* равна $x + 2 \Rightarrow$
сразу же после первого вращения операцию перебалансировки можно прервать.

Lm 8.3.3. При добавлении в AVL-дереве происходит $\mathcal{O}(1)$ присваиваний указателей.

К сожалению, высоты могут поменяться у $\mathcal{O}(\log n)$ вершин.
На практике мы покажем, что тем не менее амортизированное число изменений высот $\mathcal{O}(1)$.

Удаление из AVL-дерева происходит также, как удаление из обычного BST.
На обратном ходу рекурсии от удалённой вершины происходит перебалансировка.

Замечание 8.3.4. И в добавлении, и в удалении при подъёме вверх и перебалансировке можно пользоваться ссылками на родителя. Но удобнее всю перебалансировку делать именно на обратном ходу рекурсии.

8.4. Split/Merge

Def 8.4.1. *Пусть нам дано дерево t, $split(t, x)$ делит t на $l = \{a \mid a < x\}$ и $r = \{a \mid a \geqslant x\}$.*

Def 8.4.2. *Пусть нам даны деревья l, r: все ключи в l меньше всех ключей в r, $merge(l, r)$ – BST, полученное объединением ключей l и r.*

Замечание 8.4.3. *merge* – операция, обратная *split*.

В разборе практики показано, как можно в AVL дереве сделать split и merge за $\mathcal{O}(\log n)$.

Замечание 8.4.4. По умолчанию split и merge ломают деревья, получаемые на вход.
Но их, как и все другие процедуры работы с BST, можно сделать персистентными.

8.5. Дополнительные операции, групповые операции

Можно поддерживать в вершине размер поддерева.
Всегда, когда поддерево вершины v меняется, считаем `v->size = v->l->size+v->r->size+1`
Чтобы не обрабатывать отдельно случай "`v->l` $= \varnothing$", будем как пустое поддерево использовать специальную фиктивную вершину `Node *nullNode = {size:0, l:nullNode, r:nullNode}`.

BST используется для хранения упорядоченного множества. Структура дерева задаёт на элементах порядок, поэтому у каждого элемента дерева есть позиция (номер).
Теперь, когда у каждой вершины хранится размер поддерева, мы умеем делать 4 операции:

`getValue(i)` – получать значение ключа по номеру
`getPosition(x)` – по ключу находить его номер в дереве
`add(i, data)` – вставить вершину с записью *data* на i-ю позицию в дерево
`del(i)` – удалить вершину, находящуюся на i-й позиции.

В процедурах `getValue(i)`, `add(i,data)`, `del(i)` используется спуск по дереву,
в котором сравнение идёт не по ключу, а по размеру поддерева:

```
1 Node* getValue( Node* v, int i ) { // i - нумерация с нуля
2   if (v->l->size == i)
3     return v;
4   if (v->l->size > i)
5     return getValue(v->l, i);
6   return getValue(v->r, i - v->l->size - 1);
7 }
```

• Операции на отрезке и групповые операции

Пусть в каждой вершине дерева кроме ключа x_v хранится некое число y_v.
Научимся для примера обрабатывать запрос "`getMin(l,r)` $= \min\{y_v \mid l \leqslant x_v \leqslant r\}$"
Также, как мы поддерживали размер поддерева, поддерживаем минимальный y_v в поддереве.

Каждой вершине BST соответствует поддерево, которому соответствует отрезок значений.
Сделаем `getMin(l,r)` за $\mathcal{O}(\log n)$ спуском по дереву.
По ходу спуска, находясь в вершине v, будем поддерживать отрезок значений $[vl, vr]$.

```
1 int getMin( Node* v, int vl, int vr, int l, int r ) {}
2   if (v == nullNode || r < vl || vr < l) // [vl,vr] ∩ [l,r] = ∅
3     return INT_MAX;
4   if (l <= vl && vr <= r) // [vl,vr] ⊂ [l,r]
5     return v->min_y;
```

```
6   return min(
7     getMin(v->l, vl, v->x - 1, l, r);
8     getMin(v->r, v->x + 1, vr, l, r);
9   );
10 }
11 int result = getMin(root, INT_MIN, INT_MAX, l, r);
```

Таким образом можно считать любую ассоциативную функцию от y_v в $\{y_v \mid l \leqslant x_v \leqslant r\}$.

• **Модификация на отрезке**

Хотим со всеми $\{y_v \mid l \leqslant x_v \leqslant r\}$ сделать y_v `+=` Δ (групповая операция).
Если мы захотим честно обновить все нужные y_v, это займёт линейное время.
Чтобы получить время $\mathcal{O}(\log n)$ на запрос, воспользуемся отложенными операциями:
у каждой вершины v будем хранить `v->add` – число, которое нужно прибавить ко всем вершинам в поддереве v. Запрос "прибавления на отрезке" (`massAdd`) теперь обрабатывается по аналогии с `getMin`, есть три случая: (1) `[vl,vr]` $\cap$ `[l,r]` $= \varnothing$, (2) `vl,vr` $\subset$ `l,r` и (3) "отрезки пересекаются".

Операция называется отложенной, потому что в любом запросе (`add`, `del`, `getMin`, `massAdd`), проходя через вершину v, у которой `v->add` $\neq 0$, мы проталкиваем эту операцию вниз детям:

```
1 void add( Node* v, int x ) {
2   if (v != nullNode) // фиктивная вершина всегда должна оставаться в исходном состоянии
3     v->add += x, v->min += x;
4 }
5 void push( Node* v ) { // push = проталкивание вниз
6   add(v->l, v->add);
7   add(v->r, v->add);
8   v->add = 0; // при этом минимум не изменился
9 }
```

8.6. Неявный ключ

Теперь наша цель – на основе BST сделать структуру данных для хранения массива, которая умеет делать вставку в середину `insert(i,x)`, удаление из середины `del(i)`, считать функцию на отрезке массива и, конечно, как и положено массиву, обращаться к ячейке `a[i]=z`, `z=a[i]`.

Возьмём дерево по ключу "индекс в массиве": $x_v = i,\ y_v = a_i$.
Тогда, если в каждой вершине хранить размер поддерева и минимум y_v в поддереве, у нас уже есть операции `getValue(i)`, `add(i, data)`, `del(i)`, `getMin(l,r)`.
Нужно только после `add(i,data)` и `del(i)` пересчитывать ключи (индексы в массиве)...
Но зачем их пересчитывать и вообще хранить, если мы ими *нигде* не пользуемся?
Идея дерева по неявному ключу: можно просто не хранить ключ.

По неявному ключу также можно делать операции `split(v,i)` и `merge(l,r)`,
из них легко выразить циклический сдвиг массива – один `split` + один `merge`.

8.7. Reverse на отрезке

Если у нашего дерева есть split и merge, можно реализовать функции `getMin(l,r)`, `massAdd(l,r)` и другие чуть проще: высплитим отрезок $[l, r)$; в его корне будет содержаться минимум на $[l, r)$, к этому корню можно за $\mathcal{O}(1)$ применить отложенную операцию; сделав с корнем $[l, r)$ всё, что хотели, смерджим деревья обратно. Пример:

```
1 Node* massAdd( Node *v, int l, int r, int value ) {
2   Node *a, *b, *c;
3   split(v, b, c, r);
4   split(b, a, b, l);
5   b.add += value; // a = [0, l), b = [l, r), c = [r,end)
6   return merge(merge(a, b), c);
7 }
```

Если мы захотим перевернуть отрезок массива (`reverse(l,r)`), нам опять помогут отложенные операции. В случае `reverse` техника "высплитим отрезок $[l, r)$" всё ещё работает, а спуск по дереву уже не работает.

8.8. Итоги

Идея отложенных операций и идея неявного ключа применимы к любым BST.

Операции `getValue(i)`, `getPosition(x)`, `add(i, data)`, `del(i)`, `getMin(l,r)`, `massAdd(l,r)`, применимы к любому BST по явному ключу и неявному ключу.

Операция `reverse(l,r)` осмысленна только в дереве по неявному ключу, так как принципиально меняется порядок вершин дерева. В любом BST, где уже есть операции `split` и `merge`, можно получить `reverse(l,r)`.

8.9. Персистентность: итоги

Любое дерево поиска, в частности AVL, можно сделать персистентным, время работы останется пропорциональным глубине дерева, для AVL это $\mathcal{O}(\log n)$. Из этого следует, что дерево по неявному ключу тоже можно сделать персистентным с операциями за $\mathcal{O}(\log n)$.

Итого: у нас есть персистентный массив с обращением к i-й ячейке (`getValue(i)`) и кучей ништяков (вставка в середину, `reverse` на отрезке и т.д.), который всё это умеет за $\mathcal{O}(\log n)$.

Основной минус персистентности – каждая операция создаёт $\mathcal{O}(\log n)$ новых вершин дерева.

Лекция #9: B-tree и Treap

26 апреля

9.1. B-дерево

Зафиксируем число k. Вершиной дерева будем называть множество от $k-1$ до $2k-2$ ключей. Если в вершине хранится i ключей $x_1, x_2, \dots x_i$, то детей у неё $i+1$: $T_1, T_2, \dots, T_{i+1}$. При этом верно $T_1 < x_1 < T_2 < \dots < x_i < T_{i+1}$. Вспомним бинарное дерево поиска: $T_1 < x_1 < T_2$, один ключ, два ребёнка.

Def 9.1.1. *B-дерево: для всех вершин кроме, возможно, корня количество ключей* $[k-1, 2k-1)$, *соответственно количество детей* $[k, 2k)$, *все листья имеют одинаковую глубину.*

Lm 9.1.2. При $k \geqslant 2$ глубина B дерева, хранящего n ключей, – $\Theta(\log_k n)$

При больших k встаёт вопрос: как хранить множество ключей и детей? В основном используют два варианта: в виде отсортированного массива или в виде дерева поиска.

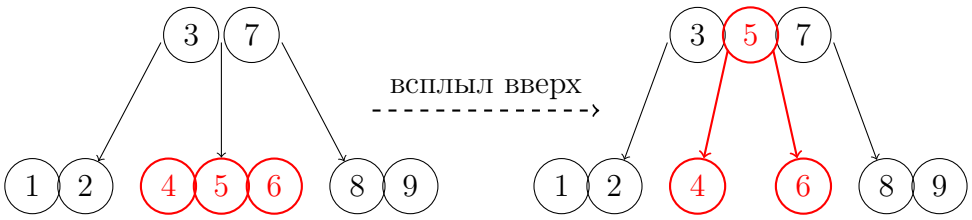
9.1.1. Поиск по B-дереву

Спуск вниз по дереву. Выбор направления, куда спускаться, происходит за $\mathcal{O}(\log k)$, – бинпоиск или поиск в BST. Итого $\log_k n \log k = \log n$ процессорных операций.

Преимущество B-дерева над другими BST – количество операций чтения с диска. Пусть мы храним в дереве большую базу данных на $10^9..10^{10}$ записей. Тогда в оперативную память такая база не помещается, и при чтении из базы происходит чтение с жесткого диска. При чтении с жесткого диска за одну операцию чтения мы читаем не 1 байт, а сразу 4096 последовательных байт. B-дерево при правильном выборе k, не ухудшая процессорное время, минимизирует число обращений к диску.

9.1.2. Добавление в B-дерево

Спустимся до листьев, добавим в листе в пачку из $[k-1..2k-2]$ ключей ещё один ключ. Если листьев в пачке стало $2k-1$, скажем "вершина переполнилась" и разделим её на две вершины по $k-1$ листу, соединённые общим ключом-отцом (итого $(k-1)+(k-1)+1 = 2k-1$ ключ). Этот ключ-отец вставим в соответствующее место "отца пачки листьев". Можно представлять эту операцию, как будто средний из ключей всплыл вверх на один уровень.



На уровень выше опять могла возникнуть проблема "слишком много ключей в вершине", тогда рекурсивно продолжим "толкать ключи вверх". Пример такой ситуации для $k = 2$ смотрите на картинке. При $k = 2$ в каждой вершине должно быть от 1 до 2 ключей.

При проталкивании ключа вверх важно, как мы храним $\Theta(k)$ ключей. Если в сортированном массиве, вставка происходит за $\mathcal{O}(k \log_k n)$. Если в дереве поиска, то за $\mathcal{O}(\log k \log_k n) = \mathcal{O}(\log n)$.

9.1.3. Удаление из B-дерева

Как обычно, сперва найдём вершину, затем, если она не лист, swap-нем её с соседним по значению ключом справа. Теперь задача свелась к удалению ключа из листа. Удалим. Сломаться могло то, что в вершине теперь меньше $k-1$ ключей. Тогда возьмём вершину-брата, и общего ключа-отца для этих двух вершин и "спустим ключ-отец вниз". Это операция обратная изображённой на картинке выше. В результате операции мы могли получить слишком толстую вершину, тогда обратно разобьём её на две (получится, что мы просто сколько-то ключей забрали от более толстого брата себе). У вершины-отца могла уменьшиться число ключей, тогда рекурсивно пойдём вверх от отца.

Итого. Мы сперва переходим к листу. Затем удаляем ключ из листа (из вершины по середине дерева нельзя просто взять и удалить ключ). И дисбаланс ключей мы исправляем подъёмом от листа к корню дереву.

Замечание 9.1.3. В удалении и добавлении за $\mathcal{O}(\log n)$ мы пользуемся тем, что дерево поиска умеет делать split и merge за логарифмическое время.

9.1.4. Модификации

B*-tree предлагает уплотнение до $[k, \frac{3}{2}k]$ ключей в вершине. Разобрано на практике.
B$^+$-tree предлагает хранить ключи только в листьях. Можно почитать в wiki.

9.1.5. Split/Merge

Делаются за высоту дерева. Будут в практике, как упражнения. Конечно, у деревьев должно быть одинаковое k.

9.2. Производные B-дерева

9.2.1. 2-3-дерево

При $k = 2$ у каждой вершины от 2 до 3 детей. Такая конструкция называется 2-3-деревом.

9.2.2. 2-3-4-дерево и RB-дерево

Можно ещё разрешить вершины, у которых ровно 4 сына. Такая конструкция будет называться 2-3-4-деревом. Поддерживать 2-3-4 свойство при добавлении и удалении также, как и у 2-3 дерева: слишком толстые вершины делим на две, тонкие соединяем с братьями.

2-3-4-дерево ничем не лучше 2-3-дерева, нам оно нужна лишь потому, что оно эквивалентно RB-дереву. А RB-дерево в свою очередь быстрая популярная реализация сбалансированного дерева поиска. Например, используется в `C++.STL` внутри `set<>`.

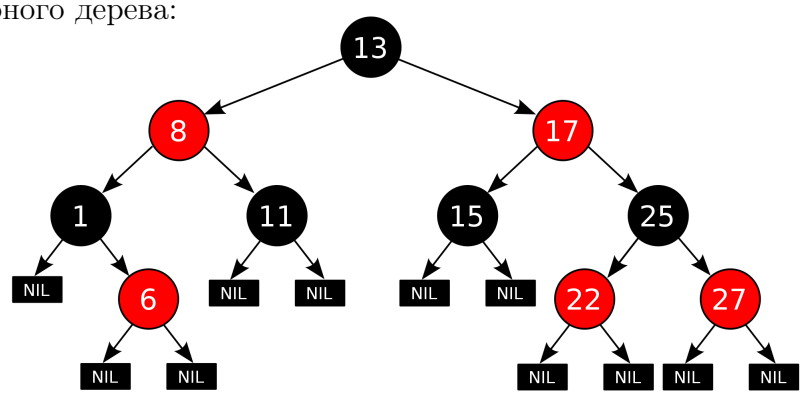
Def 9.2.1. *Рассмотрим BST, в котором у каждая вершина покрашена в красный или чёрный. Также в каждое мес то "отсутствия сына" мысленно добавим фиктивного сына. Дерево называется красно чёрным, если:*

(a) На пути от корня до любой фиктивной вершины одинаковое число чёрных вершин.
(b) Сыном красной вершины не может быть красная вершина.
(c) Корень – чёрная вершина.

Lm 9.2.2. Есть биекция между красночёрными и 2-3-4-деревьями.

Доказательство. Соединим красные вершины с их чёрными отцами, получим толстую вершину, в которой от 1 до 3 ключей и у которой от 2 до 4 детей. В обратную сторону: у вершины 2-3-4-дерева с двумя ключами создадим правого красного сына, у вершины 2-3-4-дерева с тремя ключами создадим два красных сына. ■

Пример красно-чёрного дерева:



9.2.3. AA-дерево (Arne Anderson)

Из 2-3-4-дерева разделением толстой вершины на "чёрную с красными детьми" можно получить RB-дерево. Но мы знаем, что можно жить без 4-вершин: если 2-3-дерево по тому же принципу преобразовать в RB-дерево, получится так называемое AA-дерево.

Def 9.2.3. *RB-дерево называется AA-деревом, если любая красная вершина является правым сыном своего отца.*

RB-дерево и AA-дерево при аккуратной реализации имеют отличную скорость работы. Также заметим, что оба дерева кроме обычных любому BST ссылок на детей хранят лишь один бит – цвет вершины. Для сравнения их между собой приведём цитату из работы Андерсона:

The performance of an AA tree is equivalent to the performance of a red-black tree. While an AA tree makes more rotations than a red-black tree, the simpler algorithms tend to be faster, and all of this balances out to result in similar performance. A red-black tree is more consistent in its performance than an AA tree, but an AA tree tends to be flatter, which results in slightly faster search times.

AA-дерево известно тем, что у него есть простая/красивая/быстрая реализация.

9.3. Treap

Def 9.3.1. *Декартово дерево (cartesian tree) от множества пар $\langle x_i, y_i \rangle$ – структура, являющаяся BST по x_i и кучей с минимумом в корне по y_i.*

Lm 9.3.2. Если все y_i различны, декартово дерево единственно.

Доказательство. Корень выбирается однозначно – минимальный y_i. Левое поддерево – декартово дерево от всех $x_j < x_i$, оно единственно по индукции. Аналогично правое поддерево. ■

Def 9.3.3. *Случайное дерево поиска (RBST) от множества $\{x_i\}$ получается выбором случайного элемента x в качестве корня, меньшие x образуют левое поддерево, которое строится по индукции, аналогично правое поддерево.*

Lm 9.3.4. Предыдущее определение равносильно добавлению случайной перестановки x_i в пустое обычное BST (спустились от корня вниз до упора, вставили лист).

RBST расшифровывается, как randomized balanced search tree. Покажем, почему balanced:

Lm 9.3.5. Матожидание средней глубины вершин в RBST – $\mathcal{O}(\log n)$.

Доказательство. Вспомним доказательство того, что quick sort работает за $\mathcal{O}(n \log n)$. Там мы строили дерево рекурсии и показывали, что $E(\sum size_i) \leqslant 2n \ln n$, где $size_i$ размер поддерева вершины i. Заметим, что $E(\frac{1}{n}\sum depth_i) = \frac{1}{n}E(\sum depth_i)$ Осталось показать, что $\sum depth_i = \sum size_i$. Вершина i в правой части суммы учтена ровно $depth_i$ раз, так как входит в своё поддерево, в поддерево своего отца и так далее в $depth_i$ поддеревьев. ■

Def 9.3.6. *Декартово дерево (treap) от множества ключей* $\{x_i\}$ *– декартово дерево (cartesian tree) пар* $\langle x_i, random \rangle$.

Lm 9.3.7. Treap, как BST, является RBST

Следствие 9.3.8. Матожидание средней глубины вершины в Treap – $\mathcal{O}(\log n)$

Lm 9.3.9. Матожидание средней глубины вершины в Treap – $\mathcal{O}(\log n)$

Lm 9.3.10. Матожидание высоты treap $\mathcal{O}(\log n)$

Доказательство. В нашем курсе идёт без доказательства. Доказывать можно разными способами. Вариант через неравенство Йенсена. Более детский вариант. ■

9.4. Операции над Treap

В основе Декартова дерева лежат операции Split и Merge, Add и Del выражается через них. И Split, и merge – спуск вниз по дереву. Время обеих операций – глубина вершины, до которой мы спускаемся, т.е. $\mathcal{O}(\log n)$.

Напомним `Split(x)` разделяет структуру данных на две того же типа – содержащую элементы "$< x$" и содержащую элементы "$\geqslant x$". `Merge(L, R)` – обратная к `Split` операция. В частности `Merge`, что все ключи в L меньше ключей в R.

Lm 9.4.1. Пусть над некоторым BST определены Split и Merge за $\mathcal{O}(\log n)$. Тогда можно доопределить: `Add(x) = Split + Merge + Merge`, `Del(x) = Split + Split + Merge`.

Доказательство. `Add`: разделим старое дерево на меньшие и большие x элементы. Итого, включая сам x, у нас три дерева. Склеим их. `Del`: разделим строе дерево на три – меньше x, сам x, больше x. Склеим крайние. ■

`Split` в декартовом дереве состоит из двух случаев – в какую из половин попал корень:

```
1 // чтобы вернуть пару, удобно получить в параметры две ссылки
2 void Split(Node* t, Node* &l, Node* &r, int x ) {
3   if (!t)
4     l = r = 0; // база индукции
5   else if (t->x < x)
6     Split(t->r, t->r, r, x), l = t;
7   else
8     Split(t->l, l, t->l, x), r = t;
9 }
```

Подробно объясним строку (5). Если корень `t->x` попал в левую половину, то элементы меньшие x – это корень, всё его левое поддерево и какая-то часть правого. Поэтому вызовемся рекурсивно от `t->r`, левую из образовавшихся частей подвесить к корню, правую запишем в `r`.

Лекция #10: Splay и корневая оптимизация

3 мая

10.1. Rope

Def 10.1.1. *Rope – интерфейс структуры данных, требующий, чтобы она умела производить с массивом следующие операции:*

(a) `insert(i)`, `erase(i)`
(b) `split(i)`, `merge(a, b)`, `rotate(k)`

Обычно подразумевают, что структура все выше перечисленные операции умеет делать быстро, например, за $\mathcal{O}(\log n)$. Всё выше перечисленное умеют сбалансированные деревья по неявному ключу со `split` и `merge`. У нас уже есть AVL-tree, RB-tree, Treap, сегодня ещё появится Splay-tree. Кроме того есть структуры, в основе которых лежат не деревья поиска, удовлетворяющие интерфейсу Rope. Из таких у нас сегодня появятся Skip-List и SQRT-decomposition.

10.2. Skip-list

Структура данных *односвязный список* хороша тем, что `split`, `merge`, `insert`, `erase` работают за $\mathcal{O}(1)$. За $\mathcal{O}(n)$ работает только операция поиска. Например `split(i) = find(i) + `$\mathcal{O}(1)$. Чтобы ускорить поиск можно добавить ссылки вперёд на 2^k шагов.

TODO: картинка с иллюстрацией структуры данных и нового `find`.

Правда после `insert` и `erase` такие ссылки неудобно пересчитывать.
Поэтому авторы Skip-List поступили хитрее. Skip-List – $\log_2 n$ списков. Нижний (нулевой) список – все данные нам элементы. Каждый элемент из i-го списка с вероятностью 0.5 содержится в $(i+1)$-м списке. Итого любой элемент в i-м списке содержится с вероятностью $2^{-i} \Rightarrow$
$E[|List_i|] = n2^{-i} \Rightarrow$ суммарный размер всех списков $2n$.

```
1  struct Node {
2    Node *down, *right;
3    int x, right_len;
4  };
5  const int LOG_N = 17;
6  vector<Node*> head[LOG_N+1]; // для удобства считаем, что log n не меняется
7  Node* find( int i ) {
8    Node *res;
9    int pos = -1; // в каждом списке голова - фиктивный (-1)-й элемент
10   for (Node *v = head[LOG_N]; v; res = v, v = v->down)
11     while (v->right && pos + v->right_len < i)
12       pos += v->right_len, v = v->right;
13   return res; // максимальный ключ меньший x в нижнем списке
14 }
```

Lm 10.2.1. Матожидание времени работы `find` – $\mathcal{O}(\log n)$.

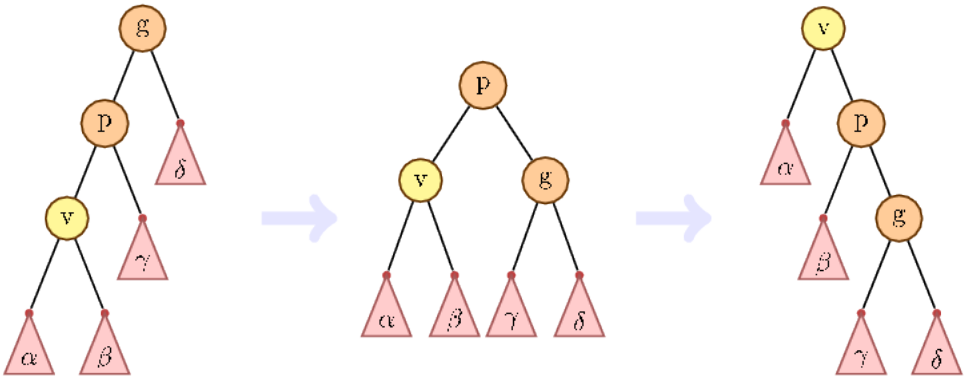
Доказательство. На каждом шаге `for` матожидание число шагов $\mathcal{O}(1)$. ■

Чтобы удалить i-элемент, сделаем `find(i)`, который в каждом списке пройдёт по элементу, предшествующему i. Для каждого списка: если i был в списке, удалим за $\mathcal{O}(1)$, сложим длины соседних рёбер. Если i не было, уменьшим длину ссылки вправо на 1.

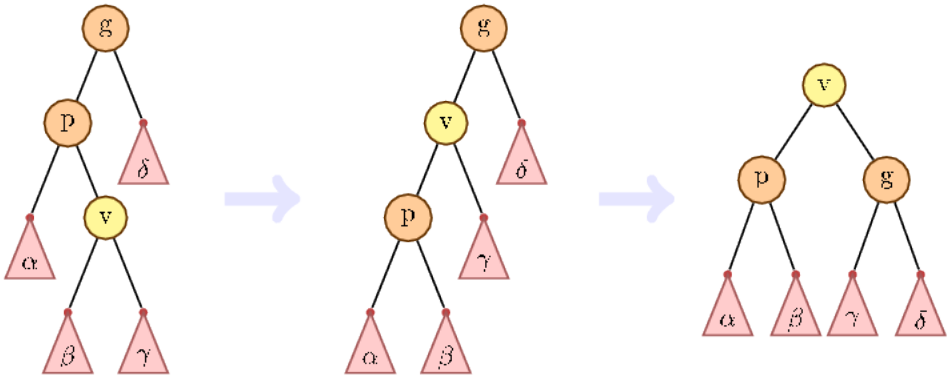
10.3. Splay tree

Splay-дерево — самобалансирующееся BST, не хранящее в вершине никакой дополнительной информации. В худшем случае глубина может быть линейна, но амортизированное время всех операций получится $\mathcal{O}(\log n)$. Возьмём обычное не сбалансированное дерево. При `add`/`del`. Модифицируем `add`: спустившись вниз до вершины v он самортизирует потраченное время вызовом `Splay(v)`, которая последовательными вращениями протолкнёт v до корня.

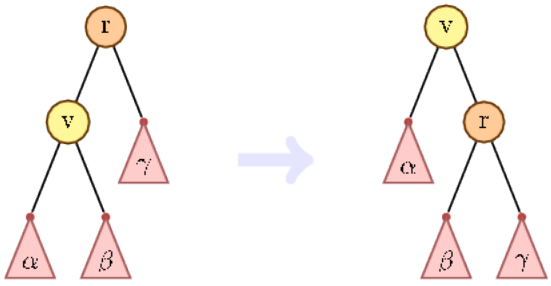
- **Zig-zig вращение**



- **Zig-zag вращение**



- **Если дедушки нет,** сделаем обычный single rotation (zig).



В частности из картинок видно, что все вращения выражаются через single rotation.

Любые другие операции со splay деревом делаются также, как и `add`: пусть v – самая глубокая вершина, до которой мы спустились $\Rightarrow$ вызовем `splay(v)`, который протолкнёт v в корень и самортизирует время работы. При этом всегда время `splay(v)` не меньше остальной полезной части $\Rightarrow$ осталось оценить время работы `splay`.

Lm 10.3.1. $x, y > 0, x + y = 1 \Rightarrow \log x + \log y \leqslant -2$

Доказательство. $\log x + \log y = \log x + \log(1 - x) = \log x(1 - x) \leqslant \log \frac{1}{2} \cdot 2 = -2$ ■

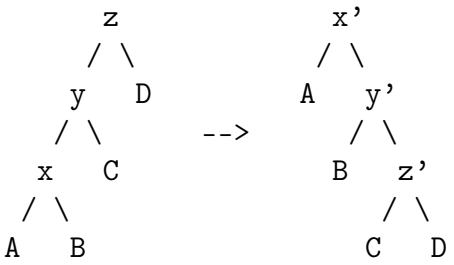
Lm 10.3.2. $x, y > 0, x + y = C \Rightarrow \log x + \log y \leqslant 2 \log C - 2$

Доказательство. $\log x + \log y = 2 \log C + \log \frac{x}{C} + \log \frac{y}{C} \leqslant 2 \log C - 2$ ■

Теперь введём потенциал. Ранг вершины $R_v = \log(size_v)$, где $size_v$ – размер поддерева. Потенциал $\varphi = \sum R_v$. Заметим сразу, что для пустого дерева $\varphi_0 = 0$ и $\forall$ момент времени $\varphi \geqslant 0$. Оценим амортизированное время операции `splay`, поднявшей v в u:

Теорема 10.3.3. $\forall v, u \; a_{v \to u} \leqslant 3(R_u - R_v) + 1 = 3 \log \frac{size_u}{size_v} + 1$

Доказательство. Полное доказательство доступно здесь. Мы разберём только случай zig-zig. Оставшиеся два аналогичны. +1 вылезет из случая zig (отсутствие деда). Итак, $a = t + \Delta\varphi = 2 + (R_{x'} + R_{y'} + R_{z'}) - (R_x + R_y + R_z) = 2 + R_{y'} + R_{z'} - R_x - R_y \leqslant 2 + R_{x'} + R_{z'} - 2R_x = F$.



Мы хотим показать $F \leqslant 3(R_{x'} - R_x) \Leftrightarrow R_{z'} \leqslant 2R_{x'} - R_x - 2 \Leftrightarrow R_{z'} + R_x \leqslant 2R_{x'} - 2$.
Теперь вспомним, что $R_{z'} = \log(C + D + 1), R_x = \log(A + B + 1) \overset{\text{лемма}}{\Rightarrow}$
$R_{z'} + R_x \leqslant 2 \log(A + B + C + D + 2) - 2 \leqslant 2R_{x'} - 2$. ■

Следствие 10.3.4. Среднее время одной операции в splay-дереве – $\mathcal{O}(\log n)$.

Доказательство. $\varphi_0 = 0, \varphi \geqslant 0 \Rightarrow \frac{1}{m} \sum t_i \leqslant a_i = \mathcal{O}(\log n)$. ■

Замечание 10.3.5. В теорему вместо $size_v$ можно подставить любой взвешенный размер: $size_v = w_v + size_l + size_r$, где w_v – вес вершины.
Чтобы потенциал всегда был неотрицательный, потребуем $w_v \geqslant 1 \Rightarrow \log w_v \geqslant 0$.

Теперь о преимуществах splay-дерева. Элемент, к которому мы обращаемся чаще, оказывается ближе к корню, поэтому обращения к нему быстрее. Формально это можно записать так:

Теорема 10.3.6. Пусть к splay-дереву поступают только запросы `find(v)`, k_v – количество обращений к вершине v, $m = \sum k_v$. Тогда суммарное время всех `find` равно $\sum_v k_v(3 \log \frac{n+m}{k_v} + 1)$.

Доказательство. Внутри суммы k_v – количество запросов к v, $(3 \log \frac{n+m}{k_v} + 1)$ – время на один запрос. Эта оценка на время получается из 10.3.3 подстановкой веса вершины $w_v = \max(k_v, 1)$.
Тогда $size_{root} \leqslant n+m$ и время $splay(v) \leqslant 3 \log \frac{size_{root}}{size_v} + 1 \leqslant 3 \log \frac{size_{root}}{k_v} + 1$. ■

На практике мы также доказали теорему о времени работы бора (задача 11, есть разбор).

В той ж практике в 9-й задаче предлагается более быстрая `top-down` реализация splay-дерева.

10.4. SQRT decomposition

10.4.1. Корневая по массиву

Идея: разобьём массив на $\sqrt{n}$ частей (кусков) размера $k = \sqrt{n}$.

• Сумма на отрезке и изменение в точке

Решение: $\mathcal{O}(1)$ на изменение, $\mathcal{O}(\sqrt{n})$ на запрос суммы. $\forall i$ для i куска поддерживаем сумму `s[i]`.

```
1 void change(i, x):
2   s[i/k] += (x-a[i]), a[i]=x
3 int get(l, r): // [l, r)
4   int res = 0
5   while (l < r && l % k != 0) res += a[l++]; // левый хвост
6   while (l < r && r % k != 0) res += a[--r]; // правый хвост
7   return accumulate(s + l / k, s + r / k, res); // цельные куски
```

Запрос на отрезке разбивается на два хвоста длины $\sqrt{n}$ и не более $\sqrt{n}$ цельных кусков.

Решение за $\mathcal{O}(\sqrt{n})$ на изменение и $\mathcal{O}(1)$ на запрос суммы: будем поддерживать частичные суммы для каждого куска и для массива s. При изменении пересчитаем за $\mathcal{O}(\sqrt{n})$ частичные суммы внутри куска номер `i/k` и частичные суммы массива s. Суммы на хвостах и на отрезке массива s считаются за $\mathcal{O}(1)$.

• Минимум на отрезке и изменение в точке

Решение за $\mathcal{O}(\sqrt{n})$ на оба запроса – поддерживать минимум в каждом куске. В отличии от суммы, минимум мы сможем пересчитать только за $\mathcal{O}(\sqrt{n})$.

10.4.2. Корневая через split/merge

Дополним предыдущие две задачи операциями `insert(i,x)` и `erase(i)`.
Будем хранить `vector` или `list` кусков. Здесь i-й кусок – это отрезок $[l_i, r_i)$ нашего массива, мы храним его как `vector`. Сам массив мы не храним, только его разбиение на куски.
Когда приходит операция `insert`/`erase`, ищем, про какой она кусок за $\mathcal{O}(\sqrt{n})$. Теперь сделаем эту операцию в найденном куске за $\mathcal{O}(\sqrt{n})$. При этом кусок мог уменьшиться/увеличиться. Кусок размера меньше $\sqrt{n}$ смерджим с соседним. Кусок размера $2\sqrt{n}$ посплитим на два.
Поскольку мы удерживаем размер куска в $[\sqrt{n}, 2\sqrt{n})$, количество кусков всегда $\Theta(\sqrt{n})$.
Время старых операций не изменилось, время новых – $\mathcal{O}(\sqrt{n})$.

10.4.3. Корневая через split/rebuild

Храним то же, что и в прошлой задаче.
Операция `split(i)` – сделать так, чтобы i-й элемент был началом куска (если это не так, кусок, в котором лежит i, нужно разделить на два). В задачах про сумму и минимум `split` $= \mathcal{O}(\sqrt{n})$.
Ещё удобнее, если `split` возвращает номер куска, началом которого является i-й элемент.
Любой запрос на отрезке $[l, r)$ теперь будем начинать со `split(r), split(l)`.
И вообще хвостов нигде нет, всегда можно посплитить.
Тогда код любой функции теперь прекрасен своей лаконичностью:

```
vector<Part> p;
void insert(int i, int x) {
  i = split(i);
  a[n++] = x; // добавили x в конец исходного массива
  p.insert(i, Part(n - 1, n));
}
void erase(int i) {
  split(i + 1);
  p.erase(split(i));
}
int get_sum(int l, int r) { // [l,r)
  int sum = 0;
  for (r = split(r), l = split(l); l < r; l++)
    sum += p[l].sum;
  return sum;
}
```

Есть небольшая проблема – число кусков после каждого запроса вырастет на $\mathcal{O}(1)$.
Давайте, если число кусок $\geqslant 3\sqrt{n}$, просто вызовем `rebuild` – процедуру, которая выпишет все куски в один большой массив и от него с нуля построит структуру данных.
Время работы такой процедуры $\mathcal{O}(n)$, вызываем мы её не чаще чем раз в $\sqrt{n}$ запросов, поэтому среднее время работы `rebuild` – $\mathcal{O}(\sqrt{n})$ на запрос.

10.4.4. Применение

Задачи про минимум, сумму мы уже умели решать через BST, все операции за $\mathcal{O}(\log n)$.
Из нового мы пока научились только запросы сумма/изменение "перекашивать":
$\langle \mathcal{O}(\log n), \mathcal{O}(\log n) \rangle \longrightarrow \langle \mathcal{O}(\sqrt{n}), \mathcal{O}(1) \rangle$ и $\langle \mathcal{O}(1), \mathcal{O}(\sqrt{n}) \rangle$.
На самом деле спектр задач, решаемых корневой оптимизацией гораздо шире. Для примера приведём максимально ужасную задачу. Выполнять нужно следующие запросы:

1. `insert(i,x)`; `erase(i)`
2. `min(l,r)`
3. `reverse(l,r)`; `add_to_all(l,r,x)`
4. `sum(l,r,x,y)`; `kth_stat(l,r,k)`

Здесь `kth_stat(l,r,k)` – k-я статистика на отрезке. Бинпоиском по ответу такой запрос сводится к задаче вида `sum(l,r,x,y)` – число элементов на $[l, r)$ со значением от x до y. Чтобы отвечать на запрос `sum(l,r,x,y)` для каждого куска будем хранить его сортированную версию, тогда ответ на запрос – обработка двух хвостов за $\mathcal{O}(\sqrt{n})$ и $2\sqrt{n}$ бинпоисков. Итого $\sqrt{n} \log n$.

Чтобы отвечать на запросы `reverse(l,r)` и `add_to_all(l,r,x)` для каждого куска будем хранить две отложенных операции – `is_reversed` и `value_to_add`. Как пример, код `reverse(l,r)`:

```
void reverse(l, r) {
  r = split(r), l = split(l);
  reverse(p + l, p + r);
  for (; l < r; l++)
    p[l].is_reversed ^= 1;
}
```

Единственное место, где будет использоваться `is_reversed` – `split` куска.

Если мы хотим решать задачу через `split/merge`, чтобы выполнять операцию `reverse`, всё равно придётся добавить `split(i)`. Теперь можно делать `reverse(l,r)` ровно, как описано выше, после чего при наличии слишком маленьких кусков, сделаем им "`merge` с соседним".

10.4.5. Оптимальный выбор k

Не во всех задачах выгодно разбивать массив ровно на $\sqrt{n}$ частей по $\sqrt{n}$ элементов.
Обозначим число кусков k. В каждом куске $m = \frac{n}{k}$ элементов.
На примере последней задачи оптимизируем k.

• **split/rebuild**

Время `inner_split` куска: $\mathcal{O}(m \log m)$[1], так как нам нужно сортировать.
Время `split(i)`: $\mathcal{O}(k) +$ `inner_split` $= \mathcal{O}(k + m \log m)$.
Время `reverse` и `add_to_all`: $\mathcal{O}(k) +$ `split(i)` $= \mathcal{O}(k + m \log m)$.
Время `insert` и `erase`: $\mathcal{O}(k) + \mathcal{O}(m)$.
Время `sum`: хвосты и бинпоиски в каждом куске $= \mathcal{O}(k \log m) + \mathcal{O}(m)$.
Суммарное время всех запросов равно $\mathcal{O}((k + m) \log m)$.
В худшем случае нам будут давать все запросы по очереди $\Rightarrow$ эта асимптотика достигается.

Вспомним про `rebuild`! В этой задаче он работает за $\mathcal{O}(k(m \log m)) = \mathcal{O}(n \log n)$.
И вызывается он каждые k запросов (мы оцениваем только асимптотику, константу не пишем).

Итого: `T(split)` $+$ `T(insert)` $+$ `T(sum)` $+ \cdots +$ $\frac{1}{k}$`T(rebuild)` $= \Theta((k+m) \log m + \frac{1}{k} n \log n) \to \min$

При минимизации таких величин сразу замечаем, что "все log-и асимптотически равны".
$\frac{1}{k} n = m \Rightarrow$ минимизировать нужно $(\frac{n}{k} + k) \log n$. При минимизации $\frac{n}{k} + k$ мы не будем дифференцировать по k. Нас интересует только асимптотика, а $\Theta(f + g) = \Theta(\max(f, g))$.
Одна величина убывает, другая возрастает $\Rightarrow$ достаточно решить уравнение $\frac{n}{k} = k$.
Итого: $k = \sqrt{n}$, среднее время работы одного запроса $\mathcal{O}(\sqrt{n} \log n)$.

• **split/merge**

В этом случае всё то же, но нет `rebuild`.
Предположим, что мы умеем делать `inner_split` и `inner_merge` за $\mathcal{O}(m)$.
Тогда нам нужно минимизировать `T(split)` $+$ `T(insert)` $+$ `T(sum)` $+ \cdots = \Theta(m + k \log m)$
Заменили $\log m$ на $\log n$, сумму на максимум $\Rightarrow$ решаем $\frac{n}{k} = k \log n$. Итого $k = \sqrt{n / \log n}$.

10.4.6. Корневая по запросам, отложенные операции

Задача: даны числа, нужно отвечать на запросы `lower_bound`. Самое простое и быстрое решение – отсортировать числа, на сортированном массиве вызывать стандартный `lower_bound`.
Решение: отложенные операции, разобрано на 29-й странице осеннего конспекта.
Решение работает в online. Тем не менее, мы как будто обрабатываем запросы пачками по $\sqrt{n}$.

Другой пример на ту же тему – решение задачи dynamic connectivity в offline.
Задача: дан неорграф. Есть три типа запросов – добавить ребро в граф, удалить ребро, проверить связность двух вершин. Нужно в offline обработать m запросов.
Решение: обрабатывать запросы пачками по $\sqrt{m}$. Подробно описано в разборе практики (№7).

[1] при большом желании можно за $\mathcal{O}(m)$

10.5. Дополнение о персистентности

Мы умеем представлять любую структуру данных в виде множества массивов и деревьев поиска. Дерево поиска само по себе может быть персистентным. Сделать массива персистентным массива мы умеем двумя способами:

1. Дерево поиска по неявному ключу. Оно кроме всего прочего будет Rope.
2. Дерево отрезков с запросами сверху.

Когда нам от массива не нужно ничего кроме присваивания“`a[i] := x`” и чтения “`x = a[i]`”, второй способ предподчтительней – он быстрее, его реализация проще.

10.5.1. Offline

Если все запросы к персистентной структуре известны заранее, мы можем построить дерево версий и обойти его поиском в глубину.

10.5.2. Персистентная очередь за $\mathcal{O}(1)$

В прошлом семестре мы прошли “очередь с минимумом без амортизации”. Там использовались две идеи: (а) очередь = два стека, (б) переворачивать стек можно лениво. Из последнего домашнего задания мы знаем, что персистентный стек с операциями push/pop/size/copy за $\mathcal{O}(1)$ существует и представляет собой дерево. Осталось собрать все знания в алгоритм:

```
struct Queue {
  Stack L, R; // L для pop, R для push
  Stack L1, R1, tmp; // вспомогательные стеки, итого 5 стеков
  int state, copied;

  Queue Copy() const {
    return Queue(L.copy(),R.copy(),L1.copy(),R1.copy(),tmp.copy(),state,copied);
  }
  int Front() const {
    return L.front(); // очередь не пуста => L не пуст!
  }
  pair<Queue, int> Pop() const {
    Queue res = Copy();
    int data = res.L.pop();
    forn(i, 3) res.Step();
    return make_pair(res, data);
  }
  Queue Push(int data) const {
    Queue res = Copy();
    res.R.push(data);
    forn(i, 3) res.Step();
    return res;
  }

  void Step() { // основной шаг переворачивания; этот метод не const!
    if (state == DO_NOTHING) {
      // if у на достаточно большой pop-стек then рано волноваться
      if (L.size > R.size) return;
      // В этот момент L.size == R.size
      R1 = R, R = new Stack(), tmp = new Stack();
      state = REVERSE_R;
```

```
32      }
33      if (state == REVERSE_R) {
34        tmp.push(R1.pop())
35        if (R1.size == 0)
36          L1 = L.copy(), state = REVERSE_L;
37      } else if (state == REVERSE_L) {
38        R1.push(L1.pop());
39        if (L1.size == 0)
40          copied = 0, state = REVERSE_L_AGAIN;
41      } else { // REVERSE_L_AGAIN
42        if (L.size > copied)
43          copied++, tmp.push(R1.pop());
44        if (L.size == copied)
45          L = tmp, state = DO_NOTHING;
46      }
47    }
48  };
```

У структуры `Stack` операции `pop` и `push` меняют стек.
Персистентность стека используется только в момент вызова метода `copy`.

10.6. Статическая оптимальность

Пусть дано множество пар $\langle x_i, p_i \rangle$. Здесь x_i – различные ключи, а p_i – вероятность того, что очередной запрос будет к ключу x_i. По ключам x_i можно построить различные BST. Новые ключи добавляться не будут. По ходу запросов менять структуру дерева нельзя. Время запроса к ключу x_i равно глубине соответствующего ключа, обозначим её d_i.

Def 10.6.1. *BST называется статически оптимальным,, если минимизирует матожидание времени запроса:* $\sum_i p_i d_i \to min$.

Задачу построения статически оптимального BST решил Дональд Кнут в 1971-м году. Кстати, похожую задачу решает алгоритм Хаффмана, разница в том, что Хаффману разрешено менять порядок ключей, а здесь порядок фиксирован: $x_1 < x_2 < \cdots < x_n$.

Статически оптимальное BST можно построить динамикой по подотрезкам за $\mathcal{O}(n^3)$:

$$f[l,r] = min_{root}\big[(\sum_{i\in[l..r]} p_i) - p_{root} + f[l, root{-}1] + f[root{+}1, r]\big]$$

Сумма p-шек обозначает, что все ключи кроме ключа x_{root} имеют глубину хотя бы 1, и эту единицу мы можем учесть уже сейчас. После чего разобьёмся на две независимые задачи – построение левого и правого поддеревьев.

Если обозначить минимальный из оптимальных $root$ за $i[l,r]$, то будет верно,
что $i[l,r{-}1] \leqslant i[l,r] \leqslant i[l{+}1,r]$, что и доказал в своей работе Кнут.
Это приводит нас к оптимизации динамики до $\mathcal{O}(n^2)$:
будем перебирать $i[l,r]$ не от l до r, а между двумя уже посчитанными i-шками.

10.7. Другие деревья поиска

В мире есть ещё много деревьев поиска, которые не охватывает курс.
Отдельное внимание хочется обратить на
Finger tree – чисто функциональный горе, умеющий обращаться к концам за $\mathcal{O}(1)$.
Finger Search tree – дерево поиска, которое почти всё делает за амортизированное $\mathcal{O}(1)$.

Tango tree – $\mathcal{O}(\log \log n)$ динамически оптимальное дерево.

Лекция #11: Дерево отрезков

10 мая

Дерево отрезков (range tree) – это и структура данных, и мощная идея. Как структура данных, оно не даёт почти ничего асимптотически нового по сравнению с BST. Основные плюсы дерева отрезков – малые константы времени и памяти, а также простота реализации.
Поэтому наш рассказ начнётся с самой эффективной реализации дерева отрезков – "снизу".

11.1. Общие слова

Дерево отрезков строится на массиве. Каждой вершине соответствует некоторый отрезок массива. Обычно дерево отрезков хранят, как массив вершин, устроенный, как бинарная куча:

`root = 1, parent[v] = v/2, leftSon[v] = 2 * v, rightSon[v] = 2 * v + 1.`

Если вершине v соответствует отрезок $[vl, vr]$ [2], её детям будут соответствуют отрезки $[vl, vm]$ и $(vm, vr]$, где `vm = (vl+vr)/2` [3]. Листья дерева – вершины с $vl = vr$, в них хранятся элементы исходного массива. Сам массив нигде кроме листьев обычно не хранится.

В вершинах дерева отрезков хранится некая фукнция от отрезка массива. Простейшие функции – `min`, `sum` чисел на отрезке, но хранить можно совершенно произвольные вещи. Например, `set` различных чисел на отрезке. Единственное ограничение на функцию: зная только значения функции в двух детях, мы должны иметь возможность посчитать функцию в вершине.

Дерево отрезков используют, чтобы вычислять значение функции на отрезке.
Обычно (но не всегда!) дерево отрезков допускает модификацию в точке, на отрезке.

11.2. Дерево отрезков с операциями снизу

```
1  void build(int n, int a[]): // O(n)
2    t.resize(2 * n) // нам понадобится не более 2n ячеек
3    for (i = 0; i < n; i++)
4      t[i + n] = a[i] // листья дерева находятся в ячейках [n..2n)
5    for (i = n - 1; i > 0; i--)
6      t[i] = min(t[2 * i], t[2 * i + 1]) // давайте хранить минимум
7
8  int get(int l, int r) {
9    int res = INT_MAX; // нейтральный элемент относительно операции
10   for (l += n, r += n; l <= r; l /= 2, r /= 2) {
11     // 1. Сперва спустимся к листьям: l += n, r += n
12     // 2. Вершины кроме, возможно, крайних, разбиваются на пары (общий отец)
13     // 3. Отрежим вершины без пары, перейдём к отрезку отцов: l /= 2, r /= 2
14     if (l % 2 == 1) res = min(res, t[l++])
15     if (r % 2 == 0) res = min(res, t[r--])
16   }
17   return res;
18 }
```

Lm 11.2.1. Время работы `get` на отрезке $[l, r]$ – $\mathcal{O}(1 + \log(r - l + 1))$

То есть, мало того, что `get` не рекурсивен, он ещё и на коротких отрезках работает за $\mathcal{O}(1)$.

[2] Можно писать на полуинтервалах. Я сам пишу и рекомендую именно полуинтервалы.
[3] Если vl и vr могут быть отрицательными или больше $\frac{1}{2}$`INT_MAX`, вычислять vm следует, как `vl+(vr-vl)/2`.

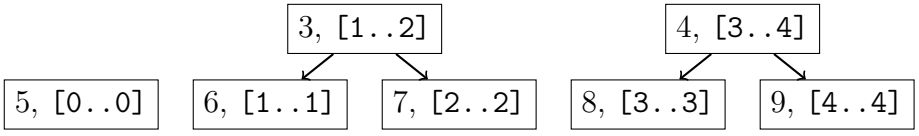
```
int change(int i, int x) {
  t[n + i] = x; // обновили значение в листе
  // пересчитали значения во всех ячейках на пути до корня (все отрезки, содержащие i)
  for (int v = (n + i) / 2; v >= 1; v /= 2)
    t[v] = min(t[2 * v], t[2 * v + 1]);
}
```

Время работы – $\mathcal{O}(\log n)$. Важно отметить отсутствие рекурсии.

Lm 11.2.2. `get` и `change` корректно решают задачу "минимум на меняющемся массиве"

Доказательство. Здесь пора обратить внимание на то, что "реализация дерева отрезков снизу" не является деревом, она является лесом (см. картинку). Этот лес состоит из слоёв. Нижний слой – исходный массив, ячейки $[n, 2n)$. Каждый следующий слой получается из предыдущего отрезанием концов без пары и переходом к отцам. Если i-му слою соответствуют вершины дерева $[l_i, r_i)$, то следующему слою соответствуют вершины $[\lceil \frac{l_i}{2} \rceil, \lfloor \frac{r_i}{2} \rfloor)$. По индукции получаем $r_i \leqslant 2l_i$ и то, что слои не пересекаются. ■

Дерево отрезков для $n = 5$:



Заметим, что `change` будет иногда ходить по бесполезным ячейкам.
Это ничему не мешает, главное, что все ячейки с полезной информацией он пересчитал.

11.3. Дерево отрезков с операциями сверху

Дерево отрезков с операциями сверху – гораздо более естественная структура.
Корень – отрезок $[0..n)$, далее дерево строится рекурсивно по определению (разд. 11.1).
Все функции работы с деревом, включая построение, – рекурсивные функции спуска. Пример:

```
int getMin( int v, int vl, int vr, int l, int r ) {
  if (vr < l || r < vl) return INT_MAX; // не пересекаются
  if (l <= vl && vr <= r) return t[v]; // вершина целиком внутри запроса
  int vm = (vl + vr) / 2;
  int fl = getMin(2 * v, vl, vm, l, r);
  int fr = getMin(2 * v + 1, vm + 1, vr, l, r);
  return min(fl, fr);
}
int result = getMin(1, 0, n - 1, l, r); // 1 = root
```

Здесь показана версия, в которой отрезок вершины $[vl, vr]$ не хранится, как свойство вершины, а вычисляется по ходу спуска сверху вниз. Версия с хранением ни чем не лучше – время пересчёта vl, vr сопоставимо с временем обращения к памяти, для чтения уже посчитанной величине.

Lm 11.3.1. В любой момент $v < 4n$. При некоторых n достигается $v = 4n - \log n$.

Доказательство. При $n = 2^k$ у нас получится полное бинарное дерево и в любой момент $v < 2n$. Проблемы будут при $2^{k-1} < n < 2^k$. Можно округлить n вверх до $2^k \leqslant 2n \Rightarrow v < 4n = 2 \cdot 2n$. ■

Есть способ чуть сэкономить память при реализации дерева отрезков сверху:
делить отрезок длины w не пополам, а на максимальное $2^k < w$ и $w - 2^k$.

Lm 11.3.2. Метод деления $w \to \max 2^k < w$ и $w - 2^k$ гарантирует $v < 3n$.

Доказательство. Дерево отрезков можно разбить на уровни: полное бинарное дерево из $n = 2^k$ вершин состоит из уровней $0, 1, \ldots, k$. При $2^k \leqslant n < 2^{k+1}$ важно, какие вершины будут созданы на $(k+1)$-м уровне. При $2^k \leqslant n \leqslant 2^k + 2^{k-1}$ на нижнем уровне будут только вершины левого поддерева $\Rightarrow$ их номера лежат в $[0, 2^k)$. Иначе $2^k + 2^{k-1} < n < 2^{k+1} \Rightarrow v < 2^{k+2} \leqslant 3n$. ■

Теперь оценим время работы запроса `getMin`.

Lm 11.3.3. `getMin` посетит не более $4 \log n$ вершин дерева.

Доказательство. Уровней в дереве $\log n$. На каждом уровне мы посетим не более четырёх вершин, потому что только в двух вершинах предыдущего уровня мы ушли в рекурсию – в вершинах, которые содержали края отрезка $[l, r]$. ■

Lm 11.3.4. `getMin` разбивает любой отрезок $[l, r]$ на не более чем $2 \log n$ вершин дерева отрезков.

Доказательство. Уровней в дереве $\log n$, на каждом уровне мы выберем не более 2 вершин. ■

Это было верно и для реализации "снизу".
Решая задачи, часто удобно думать про дерево отрезков так: "мы ему отрезок $[l, r]$, а оно разобьёт этот отрезок на $\mathcal{O}(\log_n)$ вершин, для которых уже что-то посчитано".

Минусы по сравнению с реализацией "снизу":
(a) Памяти нужно $3n$ вместо $2n$.
(b) `get` почти всегда за $\mathcal{O}(\log n)$, даже для отрезков длины $\mathcal{O}(1)$.
(c) Из-за рекурсии больше константа.
Зато есть много плюсов, главный из них: также, как и в BST, можно делать "модификацию на отрезке" – "все элементы на отрезке $[l, r]$ увеличить на x", или всем элементам присвоить x.

Модификация на отрезке делается *отложенными операциями*. Если в вершине v хранится отложенная операция, проходя через v сверху вниз, важно не забыть эту операцию протолкнуть вниз. Проталкивание вниз назовём `push`. Пример функции присваивания на отрезке:

```
1  void push(int v) {
2    if (value[v] == -1) return; // нет отложенной операции
3    value[2 * v] = value[2 * v + 1] = value[v];
4    value[v] = -1;
5  }
6  void setValue(int v, int vl, int vr, int l, int r, int x) {
7    if (vr < l || r < vl) return INT_MAX; // не пересекаются
8    if (l <= vl && vr <= r) { // вершина целиком внутри запроса
9      value[v] = t[v] = x; // не забываем пересчитывать минимум в вершине
10     return;
11   }
12   push(v);
13   int vm = (vl + vr) / 2;
14   setValue(2 * v, vl, vm, l, r, x);
15   setValue(2 * v + 1, vm + 1, vr, l, r, x);
16   // в нашей вершине отложенной операции нет, мы её толкнули вниз
17   // при написании кода важно заранее чётко решить t[v] - минимум с учётом value[v] или без
18   t[v] = min(t[2 * v], t[2 * v + 1]);
19 }
```

Ещё некоторые плюсы реализации сверху:
(a) Дерево отрезков сверху – реально дерево!
(b) Дерево отрезков сверху можно сделать персистентным.
(c) Дерево отрезков сверху можно сделать динамическим (следующий раздел).

11.4. Динамическое дерево отрезков и сжатие координат

Пусть наш массив длины $M = 10^{18}$ и изначально заполнен нулями.
Есть два способа реализовать на таком массиве дерево отрезков.

Первый способ элегантный, не содержит лишнего кода, но резко увеличивает константу времени работы: давайте все массивы заменим на `unordered_map`-ы. При этом `t[]` заменим на `unordered_map<int, T>`, где `T` – специальный тип, у которого конструктор создаёт нейтральный относительно нашей операции элемент.

Замечание 11.4.1. Первый способ работает и для версии снизу, и для версии сверху.

Второй способ предлагает отказаться от хранения в массиве:

```
1 struct Node {
2   Node *l, *r;
3   int min;
4 };
```

Теперь вершина дерева такая же, как и в BST, и вершины можно создавать лениво.
Изначально всё дерево отрезков состоит из `Node* root = NULL`.
Все запросы спускаются сверху вниз и, если попадают в `NULL`, создают на его месте `Node`.

Lm 11.4.2. Время работы любого запроса $\mathcal{O}(\log M)$.

Lm 11.4.3. После k запросов создано $\mathcal{O}(\min(k \log M, M))$ `Node`-ов.

Динамические деревья отрезков незаменимы при решении таких задач, как "k-я статистика на отрезке за $\mathcal{O}(\log n)$". Также их удобно применять в качестве внешнего дерева в 2D-деревьях.

• Сжатие координат

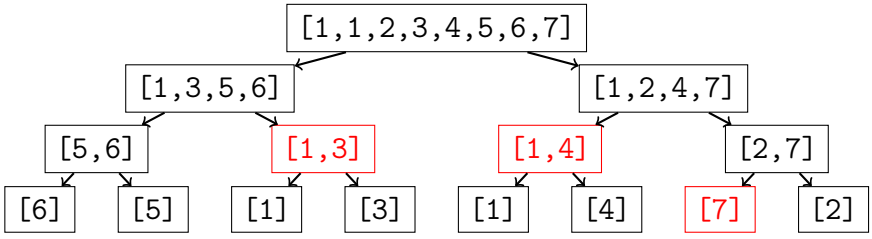
Динамическое дерево отрезков засчёт большей глубины и ссылочной структуры медленней обычного. Кроме того использует больше памяти. Поэтому, когда есть возможность, для решения исходной задачи применяют не его, а *сжатие координат*.

Задача: Начинаем с массива длины M, заполненного нулями, нужно обработать n запросов вида "изменение в точке i_j", "сумма на отрезке $[l_j, r_j)$".

Сжатие координат.
Если все запросы известны заранее (offline), на $[0, M)$ не более $3n$ интересных точек: i_j, l_j, r_j.
Давайте, сложим их в массив, отсортируем и заменим на позиции в этом массиве.
То есть, перенумеруем числами из $[0, 3n)$. Свели исходную к задаче на массиве длины $3n$.

11.5. 2D-деревья

Для начала попробуем сохранить в вершине дерева отрезков сортированную копию отрезка и посмотрим, что получится. На картинке дерево отрезков для массива `[6,5,1,3,1,4,7,2]`.



Как мы помним (11.3.4), дерево отрезков разбивает любой отрезок на $\mathcal{O}(\log n)$ вершин дерева отрезков. Красным выделены вершины, на которые распадётся отрезок $[2, 7)$.

Задача #1: дан массив длины n, отвечать на запросы `get(L,R,D,U)`: число элементов на отрезке $[L, R]$, значения которых от D до U, то есть, $\#\{i\colon L \leqslant i \leqslant R \wedge D \leqslant a_i \leqslant U\}$.

Задача #2: даны n точек (x_i, y_i) на плоскости, отвечать на запросы "число точек в прямоугольнике", то есть, $\#\{i\colon X_1 \leqslant x_i \leqslant X_2 \wedge Y_1 \leqslant y_i \leqslant Y_2\}$.

Мы описали так называемые 2D-запросы на массиве и на плоскости.

Теорема 11.5.1. Описанные выше задачи равносильны.

Доказательство. Если есть массив a_i, можно обозначить $(x_i, y_i) = (i, a_i)$. В другую сторону: отсортируем точки по x_i, теперь двумя бинпоисками условие вида $X_1 \leqslant x_i \leqslant X_2$ можно превратить в равносильное $L \leqslant i \leqslant R$. ■

Обе задачи решаются деревом отрезков сортированных массивов за $\mathcal{O}(\log^2 n)$ на запрос.
Решим задачу #1. Отрезок $[L, R]$ разделится на $\mathcal{O}(\log n)$ вершин, на каждой из них сделаем два бинпоиска, вернём "`upper_bound(U) - lower_bound(D)`".

Время построения дерева отрезков сортированных массивов – $\mathcal{O}(n \log n)$, так как каждая вершина получается, как `merge` своих детей, который считается за линейное время.

• **k-я статистика на отрезке.** Запрос `get(L,R,k)` – вернуть `sorted(a[L..R])[k]`.
Теперь мы можем решить такую задачу за $\mathcal{O}(\log^3 n)$, сделав бинпоиск по ответу. Внутри бинпоиска нам дают x и нужно узнать сколько чисел на $[L, R]$ не более x, что мы уже умеем за $\mathcal{O}(\log^2 n)$. Заметим, что бинпоиск по ответу можно реализовать, как бинпоиск по `sorted(a)`, поэтому бинпоиск сделает $\mathcal{O}(\log n)$ итераций.

Дерево отрезков сортированных массивов – структура данных на статичном (не меняющемся) массиве. Если добавить запросы изменения массива, "`a[i] = x`", то нам нужно в каждой вершине дерева отрезков динамический аналог сортированного массива, например, treap. Получили "дерево отрезков декартовых деревьев". Когда говорят "2D-дерево" обычно имеют в виду как раз дерево отрезков, в каждой вершине которого дерево.

В вершине дерева отрезков можно хранить вообще всё, что душе угодно. Например, 2D-дерево. Рассмотрим 3D-запрос на плоскости: даны n точек (x_i, y_i, z_i), нужно отвечать на запросы $\#\{i\colon X_1 \leqslant x_i \leqslant X_2 \wedge Y_1 \leqslant y_i \leqslant Y_2 \wedge Z_1 \leqslant z_i \leqslant Z_2\}$. Будем решать задачу также, как уже решили 2D-запросы: отсортируем точки по x_i, на полученном отрезке построим дерево отрезков. Пусть вершине v дерева отрезков соответствует отрезок $[vl, vr]$. Чтобы ответить на исходный запрос, в вершине v нашего дерева отрезков нужно иметь структуру данных, которая умеет отвечать на 2D-запросы для множества 2D-точек (y_i, z_i) из отрезка $[vl, vr]$. Итого $\mathcal{O}(\log^3 n)$ на запрос.

Аналогично можно на k-мерный запрос отвечать за $\mathcal{O}(\log^k n)$. При $k > 3$ это не эффективно.

11.6. Сканирующая прямая

Идея сканирующей прямой (scanline, sweep line, заметающая прямая) пришла из вычислительной геометрии и в самом общем виде звучит так: пусть на плоскости есть какие-то объекты, нарисуем вертикальную прямую и будем двигать её слева направо, обрабатывая по ходу движения события вида "объект начался", "объект закончился" и иногда "объект изменился".

Нам уже встречалась одномерная версия той же идеи: на прямой даны n точек и m отрезков, для каждого отрезка нужно узнать число точек внутри, для каждой точки узнать, сколько отрезками она покрыта. Решение: идём слева направо, обрабатываем события "отрезок начался", "отрезок закончился", "точка".

2D случай. Даны n точек и m прямоугольников со сторонами параллельными осям координат.

Задача #1: для каждой точки посчитать, сколько прямоугольниками она покрыта.
Решение: идём слева направо, встречаем и обрабатываем следующие события

(a) Прямоугольник начался: сделаем `count[y1..y2] += 1;`
(b) Прямоугольник закончился: сделаем `count[y1..y2] -= 1;`
(c) Встретили точку, тогда в `count[y]` хранится ровно число открытых ещё незакрытых прямоугольников, её покрывающих.

Дерево отрезков на массиве `count` справится с обеими операциями за $\mathcal{O}(\log n)$.
Итого время работы = сортировка + scanline = $\mathcal{O}((n+m)\log(n+m))$.

Задача #2: для каждого прямоугольника посчитать число точек внутри.
Сразу заметим, что прямоугольник можно разбить на два горизонтальных стакана:
количество точек в области $\{(x,y)\colon X_1 \leqslant x \leqslant X_2 \wedge Y_1 \leqslant y \leqslant Y_2\}$ равно разности
количеств в областях $\{(x,y)\colon x \leqslant X_2 \wedge Y_1 \leqslant y \leqslant Y_2\}$ и $\{(x,y)\colon x \leqslant X_1-1 \wedge Y_1 \leqslant y \leqslant Y_2\}$.
Итого осталось решить задачу для n точек и $2m$ горизонтальных стаканов.
Решение: идём слева направо, встречаем и обрабатываем следующие события:

(a) Встретили точку, сделаем `count[y] += 1;`
(b) Встретили конец стакана, посчитали $\sum_{y\in[y_1..y_2]}$ `count[y]`.

Дерево отрезков на массиве `count` справится с обеими операциями за $\mathcal{O}(\log n)$.

• Решение online версии.

Пусть теперь заранее известны только точки и в online приходят запросы-прямоугольники, для прямоугольника нужно посчитать число точек внутри. Возьмём решение задачи #2, дадим ему n точек и 0 стаканов. Теперь по ходу scanline-а мы хотим сохранить все промежуточные состояния дерева отрезков. Для этого достаточно сделать его персистентным. Асимптотика времени работы не изменилась (константна, конечно, хуже). Памяти теперь нужно $\mathcal{O}(n \log n)$.

Пусть `root[i]` – версия дерева до обработки события с координатой `x[i]`,
тогда запрос `get(x1, x2, y1, y2)` обработаем так:

```
1 return root[upper_bound(x, x + n, x2) - x].get(y1, y2) -
2        root[lower_bound(x, x + n, x1) - x].get(y1, y2);
```

Итого: используя предподсчёт за $\mathcal{O}(n \log n)$, мы умеем за $\mathcal{O}(\log n)$ отвечать на 2D-запрос на плоскости. Из 11.5.1 мы также умеем за $\mathcal{O}(\log n)$ обрабатывать 2D-запрос на массиве.

Важно запомнить, что любой "scanline с деревом отрезков" для решения offline задачи можно приспособить для решения online задачи, сделав дерево отрезков персистентным.

11.7. k-я порядковая статистика на отрезке

Мы уже умели бинпоиском по ответу искать k-ю статистику за $\mathcal{O}(\log^3 n)$.
Поскольку отвечаем на 2D-запросы мы теперь за $\mathcal{O}(\log n)$, это же решение работает за $\mathcal{O}(\log^2 n)$.
Перед тем, как улучшить $\mathcal{O}(\log^2 n)$ до $\mathcal{O}(\log n)$ решим вспомогательную задачу:

• **Бинпоиск → спуск по дереву.**

Задача: дан массив из нулей и единиц, нужно обрабатывать запросы "`a[i]=x`" и "k-я единица".

Решение за $\mathcal{O}(\log^2 n)$.
На данном нам массиве будем поддерживать дерево отрезков с операцией сумма. Чтобы найти k-ю единицу, сделаем бинпоиск по ответу, внутри нужно найти число единиц на префиксе $[0, x)$. Это запрос к дереву отрезков. Например, спуск сверху вниз.

Решение за $\mathcal{O}(\log n)$.
Спускаемся по дереву отрезков: если слева сумма хотя бы k, идём налево, иначе направо.

Мораль.
Внутри бинпоиска есть спуск по дереву $\Rightarrow$ скорее всего, от бинпоиска легко избавиться.

• **k-я статистика на отрезке за $\mathcal{O}(\log n)$.**

Сейчас у нас есть следующее решение за $\mathcal{O}(\log^2 n)$: возьмём точки (i, a_i), сделаем scanline с персистентным деревом отрезков. Теперь для ответа на запрос `get(l,r,k)`, делаем бинпоиск по ответу, внутри считаем `tree[r+1].get(x) - tree[l].get(x)`, где `tree[i].get(x)` обращается к i-й версии дерева отрезков и возвращает количество чисел не больше x на префиксе $[0, i)$.

Вместо бинпоиска по ответу будем параллельно спускаться по деревьям `tree[r+1]` и `tree[l]`. Пусть мы сейчас стоим в вершинах `a` и `b`. Обеим вершинам соответствует отрезок `[vl..vr]`, если (`a->l->sum - b->l->sum` $\geqslant$ `k`), есть хотя бы k чисел со значениями `[vl..vm]` и мы в обоих деревьях спустимся налево, иначе мы теперь хотим найти (`k - (a->l->sum - b->l->sum)`)-е число и в обоих деревьях спустимся направо.

11.8. (*) Fractional Cascading

TODO

11.9. (*) КД-дерево

TODO

Лекция #12: LCA & RMQ

17 мая

Если речь идёт о структуре данных, у которой есть функция построения (предподсчёт) и умение online отвечать на запросы, обозначение $\langle f(n), g(n)\rangle$ означает, что предподсчёт работает за время $\mathcal{O}(f(n))$, а ответ на запрос за $\mathcal{O}(g(n))$.

12.1. RMQ & Sparse table

Def 12.1.1. *RMQ = Range Minimum Query = запросы минимумов на отрезке.*

Задачу RMQ можно решать на не меняющемся массиве (static) и на массиве, поддерживающем изменения в точке (dynamic). Запросы минимума на отрезке будем обозначать "`get`", изменение в точке – "`change`". Построение структуры – "`build`".
Мы уже умеем решать задачу RMQ несколькими способами:

1. Дерево отрезков: `build` за $\mathcal{O}(n)$, `get` за $\mathcal{O}(\log n)$, `change` за $\mathcal{O}(\log n)$
2. Centroid Decomposition: `build` за $\mathcal{O}(n \log n)$ `get` за $\mathcal{O}(LCA)$, static.
3. SQRT decomposition: `build` за $\mathcal{O}(n)$ `get` за $\mathcal{O}(\sqrt{n})$, `change` за $\mathcal{O}(\sqrt{n})$.

Lm 12.1.2. $\nexists$ структура данных, поддерживающей `build` за $\mathcal{O}(n)$, `change` и `get` за $o(\log n)$.

Доказательство. Построим структуру от пар $\langle a_i, i\rangle$, чтобы вместе с минимумом получать и его позицию. После этого n раз достанем минимум, и на его место в массиве запишем $+\infty$. Получили сортировку за $o(n \log n)$. Противоречие. ■

А вот static версию задачи мы скоро решим за время $\langle n, 1\rangle$.

• Sparse Table

Пусть `f[k,i]` – минимум на отрезке $[i, i+2^k)$. Массив `f[]` можно предподсчитать за $n \log n$:
`f[0]` – исходный массив, `f[k,i] = min(f[k-1,i], f[k-1,i+2^{k-1}])`.
Теперь, `get` на `[l,r) = min(f[k,l], f[k,r-2^k]`, где $2^k \leqslant r - l < 2^{k+1}$.
Чтобы за $\mathcal{O}(1)$ найти такое k, используем предподсчёт "`log`", теперь `k = log[r-l]`.
Итого получили решение static RMQ за $\langle n \log n, 1\rangle$.

• Sparse Table++

Разобьём исходный массив a на куски длины $\log n$, минимум на i-м куске обозначим b_i.
Минимум на отрезке массива a, который пересекает границу двух кусков, разбивается на минимум на отрезке массива b и минимумы на "хвостах". Минимум на хвосте – это минимум на префиксе или суффиксе одного куска, все такие частичные минимумы можно предподсчитать за $\mathcal{O}(n)$. Минимум на отрезке массива b можно обработать, используя `Sparse Table` от b, который весит $\frac{n}{\log n} \log \frac{n}{\log n} \leqslant n$. Получили $\langle n, 1\rangle$ решение, работающее пока не для всех отрезков.

Осталось решить для отрезков, попадающих целиком в один из кусков.
Давайте на каждом куске построим структуру данных для решения RMQ.
Если построить Sparse Table, получим решение $\langle n \log\log n, 1\rangle$ $(\frac{n}{\log n} \cdot (\log n \log\log n))$.
Если построить дерево отрезков, получим решение $\langle n, \log\log n\rangle$.
Если построить внутри куска рекурсивно такую же структуру, получим $\langle n \log^* n, \log^* n\rangle$.

• **Disjoint Sparse Table**

Вопрос: для вычисления каких функций кроме min подходит Sparse Table?

Sparse Table покрывает любой отрезок двумя, возможно, пересекающимися.
Поэтому подходит он для любой идемпотентной[4] функции (например, для max, *gcd*).
Для суммы, произведения, композиции перестановок и т.д. не подойдёт.

Disjoint Sparse Table – аналог, разбивающий любой отрезок на два непересекающихся. Структура строится рекурсивно: посчитаем f на всех отрезках вида $[i, \frac{n}{2})$ и $[\frac{n}{2}, i)$ и вызовемся рекурсивно от частей массива $[0, \frac{n}{2})$ и $[\frac{n}{2}, n)$. Если отрезок пересекал точку $\frac{n}{2}$, его получится разбить на две части прямо на корневом уровне, иначе спустимся в рекурсию.

Получилась стандартная рекурсия от разделяй и властвуй, как в MergeSort. Такую рекурсию можно воспринимать, как дерево отрезков, – каждой вершине дерева рекурсии соответствует отрезок исходного массива. Итого время и память предподсчёта $\mathcal{O}(n \log n)$.

Чтобы быстро по отрезку `[l,r)` находить вершину получившегося дерева отрезков, где произойдёт разбиение `[l,r) = [l,m) + [m,r)`, нужно: предположить $n = 2^k$; вычислить "i = старший бит числа (`r ^ l`)" – уровень дерева отрезков; взять вершину номер (`l>>i`) на этом уровне.

12.2. LCA & Двоичные подъёмы

В дереве с корнем для двух вершин можно определить отношение "a – предок b".
Более того, на запрос `isAncestor(a,b)` легко отвечать за $\mathcal{O}(1)$.
Предподсчитаем времена входа выхода dfs-ом по дереву, тогда:

```
1 bool isAncestor(int a, int b):
2   return t_in[a] <= t_in[b] && t_out[b] <= t_out[a];
```

LCA(a, b) – общий предок вершин a и b максимальной глубины.
LCA = least common ancestor = наименьший общий предок.

Мы уже умеем искать LCA за $\Theta(dist(a, b))$: предподсчитаем глубины всех вершин, при подсчёте LCA сперва уравняем глубины a и b, затем будем параллельно подниматься на 1 вверх.

Соптимизируем эту идею – научимся прыгать сразу на 2^k вверх $\forall v, k$.
`up[k,v] = up[k-1,up[k-1,v]]` – прыжок на 2^k равен двум пряжкам на 2^{k-1}.
База: `up[0,v] = parent[v]`.
Переход: если уже построен слой динамики `up[k-1]`, мы за $\Theta(n)$ насчитаем слой `up[k]`.
Чтобы не было крайних случаев, сделаем `up[0,root] = root`.
LCA по-прежнему состоит из двух частей – уравнять глубины и прыгать вверх:

```
1 int up[MAX_K][N];
2 int LCA(int a, int b) {
3   if (depth[a] > depth[b]) a = jump(a, depth[a] - depth[b]);
4   if (depth[a] < depth[b]) b = jump(b, depth[b] - depth[a]);
5   for (int k = MAX_K - 1; k >= 0; k--)
6     if (up[k][a] != up[k][b])
7       a = up[k][a], b = up[k][b];
8   return a == b ? a : up[0][a];
9 }
```

[4] f идемпотентна $\Leftrightarrow f(a, a) = a$

На $2^{\texttt{MAX\_K}}$ вверх мы не будем пытаться прыгать $\Rightarrow$ пусть $\texttt{MAX\_K} = \lceil \log n \rceil$.
`jump(v,d)` прыгает вверх из v на d, для этого d нужно представить, как сумму степеней двойки.
Время и память предподсчёта – $\Theta(n \log n)$, время LCA – $\Theta(\log n)$.
Можно уменьшить константу времени работы, используя `isAncestor`:

```
int up[MAX_K][N];
int LCA(int a, int b) {
  for (int k = MAX_K - 1; k >= 0; k--)
    if (!isAncestor(up[k][a], b))
      a = up[k][a];
  return isAncestor(a, b) ? a : up[0][a];
}
```

12.3. RMQ±1 за $\langle n, 1 \rangle$

Def 12.3.1. *Говорят, что массив обладает ±1 свойством, если* $\forall i \ |a_i - a_{i+1}| = 1$

Собственно наша цель – решить RMQ на ±1 массиве.
Давайте возьмём уже имеющуюся у нас идею из Sparse Table++.
"*Разобьём исходный массив* `a` *на куски длины* $k = \frac{1}{2} \log n$, *минимум на* i*-м куске обозначим* $\texttt{b}_i$."
$\min(a_1, a_2 \ldots, a_k) = a_1 + \min(0, a_2 - a_1, a_3 - a_1, \ldots, a_k - a_1) \Rightarrow$ пусть первое число $\forall$ куска $= 0$.
Теперь благодаря ±1 свойству массива любой кусок длины k можно представить, как последовательность плюс-минус единиц длины k. Различных таких кусков $2^k = 2^{(\log n)/2} = \sqrt{n}$.
Воспользуемся так называемой *идеей четырёх русских*[5] –
предподсчитаем все ответы для всех подотрезков всех $\sqrt{n}$ возможных кусков.
В разборе практики описано, как это можно сделать за $\mathcal{O}(2^k) = \mathcal{O}(\sqrt{n})$.

12.4. LCA $\to$ RMQ±1 и Эйлеров обход

Напомним, эйлеров обход графа – цикл, проходящий по каждому ребру ровно один раз.
$\forall$ дерева, если каждое ребро заменить на два ориентированных, мы получим эйлеров орграф.
Чтобы построить эйлеров обход дерева, мы пишем `dfs(v)`.

```
void dfs(int v) {
  for (Edge e : graph[v]) { // пусть мы храним только рёбра, ориентированные вниз
    answer.push_back(edge(v,x));
    dfs(x);
    answer.push_back(edge(x,v));
  }
}
```

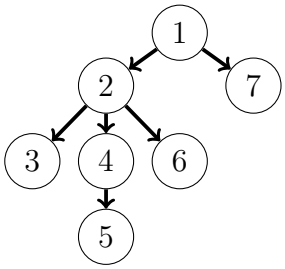
Такой обход назовём обычным или "*эйлеровым обхода дерева первого типа*".

Иногда имеет смысл хранить другую информацию после обхода:

```
void dfs2(int v) {
  index[v] = answer.size(); // сохранили для вершины v любое её вхождение в answer
  answer.push_back(v);
  for (Edge e : graph[v]) {
    dfs2(x);
    answer.push_back(v); // когда поднимаемся, проходим через вершину v => выпишем её
  }
```

[5] идея гласит "давайте предподсчитаем ответа на всех возможных маленьких тестах"

```
8  }
9  void dfs3(int v) {
10   L[v] = answer.size(); // аналог времени входа
11   answer.push_back(v); // просто сохраняем порядок обхода вершин dfs-ом
12   for (Edge e : graph[v])
13     dfs3(x);
14   R[v] = answer.size(); аналог времени выхода
15 }
```



Эйлеров обход 1-го типа (обычный)	(1,2) (2,3) (3,2) (2,4) (4,5) (5,4) (4,2) (2,6) (6,2) (2,1) (1,7) (7,1)
Эйлеров обход второго типа (± 1)	Обход: 1 2 3 2 4 5 4 2 6 2 1 7 1 Высоты: 0 1 2 1 2 3 2 1 2 1 0 1 0
Обход третьего типа	Обход: 1 2 3 4 5 6 7

Второй обход по сути тоже эйлеров.
Мы сохраняем не рёбра, по которым проходим, а вершины, в которых оказываемся при обходе.
Третий обход уже слабо напоминает эйлеров, но мы в контексте задач, где нужно выбирать между 2-м и 3-м обходами, будем иногда называть его эйлеровым.
Зачем нужны 2-й и 3-й обходы будет понятно уже сейчас, 1-й нам пригодится для ETT.

Lm 12.4.1. После третьего обхода отрезок обхода $[L_v, R_v)$ задаёт ровно поддерево вершины v.

Следствие 12.4.2. Пусть у каждой вершины дерева есть вес w_v. Тогда мы теперь умеем за $\mathcal{O}(\log n)$ делать все операции на поддереве, которые ДО умело делать на отрезке.

Lm 12.4.3. Массив высот `h[i] = height[answer[i]]` второго обхода обладает ± 1 свойством.

Lm 12.4.4. `LCA(a,b) = answer[h.RMQ[index[a],index[b]]]`

Доказательство. dfs по пути из a в b пройдёт через LCA. Это будет вершина минимальной высоты на пути, так как, чтобы попасть в ещё меньшие, dfs должен сперва выйти из LCA. ■

Замечание 12.4.5. Итого мы получили решение задачи LCA за $\langle n, 1\rangle$. Этот относительно свежий результат был получен в 2000-м Фарах-Колтоном и Бендером (два человека). Ссылка на статью.

Замечание 12.4.6. На практике популярен способом решения LCA: LCA $\to$ RMQ, а RMQ решим через Sparse Table. Это $\langle n \log n, 1\rangle$, причём у $\mathcal{O}(1)$ относительно небольшая константа.

12.5. RMQ $\to$ LCA

Чтобы свести задачу "RMQ на массиве a" к LCA, построим декартово дерево на парах (i, a_i).
Пары уже отсортированы по x, поэтому построение – проход со стеком за $\mathcal{O}(n)$.

Lm 12.5.1. RMQ на $[l, r]$ в исходном массиве равно $LCA(l, r)$ в полученном дереве.

Доказательство. Каждой вершине декартова дерева соответствует отрезок исходного массива, корнем поддерева выбирается минимум по $y_i = a_i$ на этом отрезке.
Будем спускаться от корня дерева, пока не встретим вершину, которая разделяет l и r.
Ключ, записанный в найденной вершине, обозначим i, отрезок вершины $[L_i, R_i] \Rightarrow$
$a_i = \min_{L_i \leqslant j \leqslant R_i} a_j$ и $L_i \leqslant l \leqslant i \leqslant r \leqslant R_i \Rightarrow a_i \geqslant \min_{l \leqslant j \leqslant r} a_j$ и $i \in [l, r]$.
Осталось заметить, что $i = \text{LCA}(l, r)$. ■

Следствие 12.5.2. Мы научились решать статичную версию RMQ за $\langle n, 1\rangle$. Победа!

12.6. LCA в offline, алгоритм Тарьяна

Для каждой вершины построим список запросов с ней связанных. Пусть i-й запрос – (a_i, b_i).

```
1 q[a[i]].push_back(i), q[b[i]].push_back(i)
```

Будем обходить дерево dfs-ом, перебирать запросы, связанные с вершиной, и отвечать на все запросы, второй конец которых серый или чёрный.

```
1  void dfs( int v) {
2    color[v] = GREY;
3    for (int i : q[v]) {
4      int u = a[i] + b[i] - v;
5      if (color[u] != WHITE)
6        answer[i] = DSU.get(u);
7    }
8    for (int x : graph[v])
9      dfs(x), DSU.parent[x] = v;
10   color[v] = BLACK;
11 }
```

Серые вершины образуют путь от v до корня. У каждой серой вершины есть чёрная часть поддерева, это и есть её множество в DSU. LCA(v,u) – всегда серая вершина, то есть, нужно от u подниматься вверх до ближайшей серой вершины, что мы и делаем.

В коде для краткости используется DSU со сжатием путей, но без ранговой эвристики, поэтому время работы будет $\mathcal{O}(m + n \log n)$. Если применить обе эвристики, получится $\mathcal{O}((m + n)\alpha)$, но нужно будет поддерживать в корне множества "самую высокую вершину множества".

12.7. LA (level ancestor)

Запрос "`LA(v,k)`" – подняться в дереве от вершины v на k шагов вверх.
Мы уже умеем решать эту задачу за $\langle n \log n, \log n\rangle$ двоичными подъёмами.
В offline на m запросов можно ответить dfs-ом за $\mathcal{O}(n + m)$: когда dfs зашёл в вершину v, у нас в стеке хранится весь путь до корня, и к любому элементу пути мы можем обратиться за $\mathcal{O}(1)$.

• Алгоритм Вишкина

Как и при сведении LCA $\to$ RMQ±1, выпишем высоты Эйлерова обхода первого типа.
`LA(v,k) = getNext(index[v],height[v]-k)`, где `index` – позиция в Эйлеровом обходе, а `getNext(i,x)` возвращает ближайший справа от i элемент $\leqslant x$.
Мы умеем отвечать на `getNext(i,x)` за $\langle n, \log n\rangle$ одномерным ДО снизу или сверху.

12.8. Euler-Tour-Tree

Задача: придумать структуру данных для хранения графа, поддерживающую операции

(a) `link(a, b)` – добавить ребро между a и b.
(b) `cut(a, b)` – удалить ребро между a и b.
(c) `isConnected(a, b)` – проверить связность двух вершин.

При этом в каждый момент времени выполняется условие отсутствия циклов (граф – лес).
По сути мы решаем *Dynamic Connectivity Problem* с дополнительным условием "граф – лес".

Решение: хранить обычный эйлеров обход дерева (ориентированные рёбра).
Эйлеров обход – массив, на котором мы заведём rope, например, treap по неявному ключу.

```
map<pair<int,int>, Node*> node; // по орребру умеем получать вершину treap
vector<Node*> anyEdge; // для каждой вершины графа храним любое исходящее ребро
bool isConnected(int a, int b) {
  // взяли у каждой вершины произвольное ребро, проверили, что два ребра живут в одном дереве
  return getRoot(anyEdge[a]) == getRoot(anyEdge[b]);
}
void cut(int a, int b) {
  // по орребру получаем Node*, находим его позицию в эйлеровом обходе
  Node *node1 = node[make_pair(a, b)];
  Node *node2 = node[make_pair(b, a)];
  int i = getPosition(node1), j = getPosition(node2);
  if (i > j) swap(i, j); // упорядочили (i,j)
  Node *root = getRoot(node1), *a, *b, *c;
  Split(root, a, b, i);
  Split(b, b, c, j - i); // разделили дерево на три части: (a) (i b) (j c)
  Delete(b, 0), Delete(c, 0); // собственно удаление лишнего ребра
  Merge(a, c); // в итоге теперь есть два дерева: (a c) и (b)
}
```

С операцией `link(a,b)` чуть сложнее – нужно сделать циклические сдвиги обходов: если обходы называются X и Y, а добавляемые рёбра e_1 и e_2, мы хотим представить ответ, как Xe_1Ye_2.

```
void link(int a, int b) {
  Node *pa = anyEdge[a], *pb = anyEdge[b];
  Rotate(getRoot(pa), getPosition(pa)); // Split + Merge
  Rotate(getRoot(pb), getPosition(pb));
  // Теперь первые ребра эйлеровых обходов - исходящие из a и b соответственно
  Node *e1 = createEdge(a, b), *e2 = createEdge(b, a);
  Merge(Merge(getRoot(pa), e1), Merge(getRoot(pb), e2));
}
```

12.9. (*) LA, быстрые решения

TODO

Лекция #13: HLD & LC

24 мая

13.1. Heavy Light Decomposition

Задача. Дано дерево с весами в вершинах. Нужно считать функцию на путях дерева, и поддерживать изменение весов вершин.

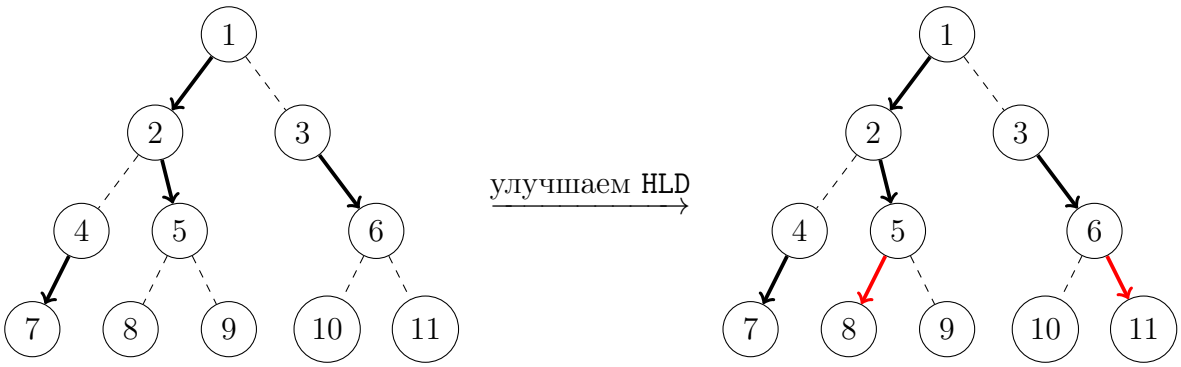
В частном случае "дерево = путь" задача уже решена деревом отрезков. Обобщим решение для произвольного дерева: разобьём вершины дерева на пути, на каждом построим дерево отрезков. Теперь любой путь разбивается на какое-то количество отрезков, на которых функцию можно посчитать деревом отрезков. Осталось выбрать такое разбиение на пути, чтобы количество отрезков всегда было небольшим. Подвесим дерево. Теперь каждое ребро или лёгкое, или тяжёлое, причём у каждой вершины не более одного тяжёлого сына.

Def 13.1.1. *Heavy-Light декомпозиция дерева (HLD) – пути образованные тяжёлыми рёбрами.*

Напомним, смысл в том, чтобы путь разбивался на как можно меньшее число отрезков. Поэтому если в пути можно включить больше рёбер, полезно это сделать. В частности каждую вершину выгодно соединить хотя бы с одним сыном.

Def 13.1.2. *Улучшенная HLD: для каждой вершины выбрали ребро в самого тяжёлого сына.*

Улучшенная HLD включает все те же рёбра, что и обычная, и ещё некоторые.



Построить разбиение на пути можно за линейное время.
Деревья отрезков на путях также строятся за линейное время.

```
1  void build_HLD(int v) {
2    int ma = -1;
3    size[v] = 1;
4    for (int x : children[v]) {
5      build_HLD(x);
6      size[v] += size[x];
7      if (ma == -1 || size[x] > size[ma])
8        ma = x;
9    }
10   path[v] = (ma == -1 ? path_n++ : path[ma]); // номер пути, на котором лежит v
11   pos[v] = len[path[v]]++; // позиция v внутри пути
12   top[path[v]] = v; // для каждого пути помним верхнюю вершину
13 }
14 build_HLD(root); // за O(n) для каждой v нашли path[v], pos[v], для каждого p len[p], top[p]
```

Lm 13.1.3. В `HLD` на пути от любой вершины до корня не более $\log n$ прыжков между путями.

Доказательство. Все прыжки между путями – лёгкие рёбра. ■

• Подсчёт функции на пути.

Можно найти `LCA` и явно разбить путь $a \to b$ на два вертикальных: $a \to$ `LCA` $\to b$. Подсчёт функции на пути $a \to$ `LCA`: поднимаемся из a, если `path[a]` $=$ `path[LCA]`, можем посчитать функцию за одно обращение к дереву отрезков: `tree[path[a]].get(pos[a], pos[LCA])`.
Иначе поднимаемся до `top[path[a]]` и переходим в `parent[top[path[a]]]`.

Теорема 13.1.4. Время подсчёта функции на пути $\mathcal{O}(\log^2 n)$.

Доказательство. Не более $2 \log n$ обращений к дереву отрезков. ■

Теорема 13.1.5. Время изменения веса одной вершины $\mathcal{O}(\log n)$.

Доказательство. Вершина лежит ровно в одном дереве отрезке. ■

Замечание 13.1.6. Аналогично можно считать функции на рёбрах.
Например, используем биекцию "вершина $\leftrightarrow$ ребро в предка".

Замечание 13.1.7. Воспользуемся функцией `isAncestor`, тогда можно обойтись без подсчёта `LCA`: прыгать от вершины a вверх, пока не попадём в предка b, и затем от b вверх, пока не попадём в предка a. Более того, таким образом с помощью `HLD` можно найти `LCA` за время $\mathcal{O}(\log n)$, используя всего $\mathcal{O}(n)$ предподсчёта.

• Другие применения.

`HLD` позволяет выполнять любые запросы на пути дерева, которые умело выполнять на отрезке дерево отрезков. Например, перекрасить путь в некоторый цвет. Или делать `+=` на пути дерева и при этом поддерживать минимум на пути в дереве.

С помощью `HLD` можно считать даже гораздо более сложные вещи: например, в дереве с весами на рёбрах поддерживать изменение веса ребра и длину диаметра дерева.

13.2. Link Cut Tree

Мы уже очень много всего умеем делать с деревьями.
С помощью `HLD` умеем обрабатывать запросы `getMax(a,b)` и `changeWeight(v)`.
С помощью `ETT` умеем обрабатывать запросы `link(a,b)`, `cut(a,b)`, `isConnected(a,b)`.

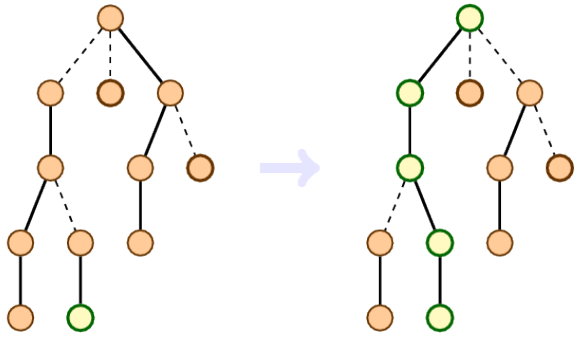
Link-Cut-Trees структура данных, которая умеет всё вышеперечисленное и не только.
Основная идея = динамически меняющаяся декомпозиция дерева на пути + для каждого пути сохранить rope (например, treap по неявному ключу).
Изначально каждая вершина – отдельный путь.

• `GetRoot(v), GetPos(v)`

В каждый момент любая вершина v лежит ровно в одном пути, а путь этот хранится в treap, который состоит из `Node*`. Давайте для v хранить ссылку на её `Node*` в treap её пути: `p[v]`. Пусть в treap есть ссылки на отцов $\Rightarrow$ от `p[v]` можно за $\mathcal{O}(\log n)$ подняться до корня её treap и параллельно насчитать "число вершин в treap левее `p[v]`", то есть, позицию v в её пути.

Если поступает любой из запросов {`getMax(a,b)`, `link(a,b)`, `cut(a,b)`, `isConnected(a,b)`}, сделаем сначала `Expose(a)`.

Def 13.2.1. *`Expose(v)` – операция, меняющая декомпозиция дерева на пути.*
Она все вершины на пути от v до корня объединяет в один большой путь.



Время работы `Expose(v)` равно $\mathcal{O}(k)$ вызовов `split` и `merge`, где k – число прыжков между путями при подъёме от v до корня. Мы докажем, что амортизированно $k \leqslant \log n$.

• `MakeRoot(v)`

Rope умеет делать `reverse`, для этого достаточно в каждой вершине treap хранить отложенную операцию `isReversed`. Если после `Expose(v)` отрезать часть пути под v и сделать "`GetRoot(p[v]).isReversed ^= 1`", вершина v станет новым корнем дерева.
Оценим время работы новой операции: $T(\texttt{MakeRoot(v)}) = T(\texttt{Expose(v)}) + \mathcal{O}(\log n)$.

• `getMax(a,b)`, `link(a,b)`, `cut(a,b)`, `isConnected(a,b)`

Красота происходящего в том, что все операции теперь коротко выражаются:

```
1  int getMax(int a, int b) {
2    MakeRoot(a), MakeRoot(b);
3    return GetRoot(a)->max;
4  }
5  void link(int a, int b) {
6    MakeRoot(a), MakeRoot(b);
7    parent[a] = b;
8  }
9  void cut(int a, int b) {
10   MakeRoot(a), MakeRoot(b);
11   split(GetRoot(b), 1); // путь состоит из двух вершин - a и b, разрежем на два
12   parent[a] = -1; // a - нижняя из двух вершин
13 }
14 bool isConnected(int a, int b) {
15   MakeRoot(a), MakeRoot(b); // a и b были в одной компоненте ⇒ теперь они в одном пути
16   return GetRoot(a) == GetRoot(b);
17 }
```

Красота функции `getMax` в том, что после двух `MakeRoot` весь путь $a \rightsquigarrow b$ – ровно один treap в "покрытии путями". И максимум на пути хранится в корне соответствующего treap.

Теорема 13.2.2. Суммарное время m операций `Expose` – $\mathcal{O}(n + m \log n)$.

Доказательство. Потенциал φ = минус "число тяжёлых рёбер, покрытых путями".
$\varphi_0 = 0, \varphi \geqslant -n \Rightarrow \sum t_i = \sum a_i + (\varphi_0 - \varphi_m) \leqslant n + \sum a_i$. Осталось оценить a_i.

Число лёгких рёбер на пути будем обозначать "`Л`", число тяжёлых "`Т`".
`Expose`: $a_i = t_i + \Delta\varphi \leqslant t_i - \texttt{Т} + \texttt{Л} \leqslant k - (k - \log n) + \log n = \mathcal{O}(\log n)$.
`MakeRoot = Expose + Reverse`. Оценим `Reverse`: $a_i = t_i + \Delta\varphi \leqslant \log n + \log n$. ■

13.3. MST за $\mathcal{O}(n)$

Алгоритм построения MST от графа из n вершин и m рёбер обозначим $F(n, m)$. Алгоритм F:

1. Сделаем 3 шага алгоритма Борувки: $n \to \frac{n}{8}$. Время работы $\mathcal{O}(n+m)$.
2. Берём случайное множество A из $\frac{m}{2}$ рёбер.
   Построим MST от A рекурсивным вызовом $F(\frac{n}{8}, \frac{m}{2})$.
   Множество рёбер полученного MST обозначим T. Рёбра из $A \setminus T$ точно не входят в MST(E).
3. Переберём рёбра из $B = E \setminus A$, оставим из них $U \subseteq B$ – те рёбра, что могут улучшить T.
   Ребро (a, b) может улучшить T, если a и b не связны в T или максимум на пути в T между a и b больше веса ребра. Рёбра из $B \setminus U$ точно не входят в MST(E).
4. MST$(E) \subseteq T \cup U$. Сделаем рекурсивный вызов от $F(\frac{n}{8}, |T| + |U|)$.

Оценим общее время работы: $|T| \leqslant \frac{n}{8} - 1$. Скоро мы покажем, что матожидание $|U| \leqslant \frac{n}{8} - 1$. Суммарное время работы $F(n, m) \leqslant (n+m) + M(\frac{n}{8}, \frac{m}{2}) + F(\frac{n}{8}, \frac{m}{2}) + F(\frac{n}{8}, \frac{n}{4})$, где $M(n, m)$ – время поиска в offline минимумов на m путях дерева из n вершин. Оказывается $M(n, m) \leqslant n+m$. Сумма параметров в рекурсивных вызовах равна $\frac{n}{2}+\frac{m}{2} \Rightarrow$ вся эта прелесть работает за $\mathcal{O}(n+m)$.

Осталось всё последовательно доказать.

Lm 13.3.1. $A_1 = A \setminus T$ и $B_1 = B \setminus U$ точно не лежат в MST(E)

Доказательство. Представим себя Краскалом.
Встретив ребро $e_a \in A_1$, мы уже имеем путь между концами e_a (рёбра из T) $\Rightarrow e_a$ не добавим.
Встретив ребро $e_b \in B_1$ у нас уже есть путь между концами e_b (рёбра из T) $\Rightarrow e_b$ не добавим. ■

Как быстро посчитать минимумы на путях в дереве можно прочитать в работе Тарьяна. У нас на экзамене этого не будет. Лучшее, что мы сейчас умеем – $\mathcal{O}(m + n \log n)$: с помощью LCA разбили пути на вертикальные, перебираем их снизу вверх, считаем минимум за линию, но со сжатием путей. Собственно работа Тарьяна показывает, как в этом подходе использовать не только сжатие путей, но полноценное СНМ.

• Random Sampling Lemma.

Lm 13.3.2. Пусть p – вероятность включения ребра из E в A на втором шаге алгоритма F (в описанном алгоритме $p = \frac{1}{2}$) $\Rightarrow$ матожидание размера множества U не больше, чем $(\frac{1}{p}-1)(n-1)$.

Доказательство. Представим себя Краскалом, строящим MST от E. Основная идея доказательства: подбрасывать монетку, решающую попадёт ребро в A или в B не заранее, а прямо по ходу Краскала, когда встречаем "интересное" ребро. Упорядочим ребра по весу, перебираем их. Если Краскал считает, что очередное ребро надо добавить в остов, то с вероятностью p добавляем (событие X), а с $1-p$ пропускаем ребро (событие Y). Оценим матожидание числа просмотренных рёбер перед первым событием типа X, включая само X. $E = 1+(1-p)E \Rightarrow E = \frac{1}{p}$. Из этих $\frac{1}{p}$ рёбер первые $\frac{1}{p}-1$ идут в U, а последнее идёт в T. Поскольку в остов можно добавить максимум $n-1$ рёбер, событие X произойдёт не более $n-1$ раз, перед каждым X случится в среднем $\frac{1}{p} - 1$ событий типа Y. Итого $E\big[|U|\big] \leqslant (n-1)(\frac{1}{p} - 1)$. ■

13.4. RMQ Offline

У нас есть алгоритм Тарьяна поиска LCA в Offline (dfs по дереву + CHM).
Мы умеем сводить RMQ-offline к LCA-offline построением декартова дерева.
Если две эти идеи объединить в одно целое, получится удивительно простой алгоритм:

```
void solve(int m, Query *q, int n, int *a) {
  vector<int> ids(n); // для каждой правой границы храним номера запросов
  for (int i = 0; i < m; i++)
    ids[q[i].right].push_back(i);
  DSU dsu(n); // инициализация CHM: каждый элемент - самостоятельное множество
  stack<int> mins;
  for (int r = 0; r < n; r++) {
    while (mins.size() && a[mins.top()] >= a[r])
      dsu.parent[mins.pop()] = r; // минимум достигается в r, объединим отрезки
    mins.push(r);
    for (int i : ids[r])
      q[i].result = dsu.get(q[i].left);
  }
}
```

В описанном выше алгоритме можно рассмотреть построение декартова дерево и параллельно обход полученного дерева dfs-ом. Проще воспринимать его иначе.
Когда мы отвечаем на запросы $[l, r]$ при фиксированном r, ответ зависит только от l.
Если $l \in [1, m_1]$, ответ – $a[m_1]$, если $l \in (m_1, m_2]$, ответ – $a[m_2]$ и т.д.
Здесь m_1 – позиция минимума на $[1, r]$, а m_{i+1} – позиция минимума на $(m_i, r]$.
Отрезки $(m_i, m_{i+1}]$ мы будем поддерживать, как множества в DSU.
Минимумы лежат в стеке: $a[m_1] \leqslant a[m_2] \leqslant a[m_3] \leqslant \ldots$
Когда мы расширяем префикс: $r \to r{+}1$, стек минимумом обновляется, отрезки объединяются.