

Первый курс, весенний семестр 2019/20

Конспект лекций по алгоритмам

Собрано 20 мая 2020 г. в 18:37

Содержание

1. Поиск в глубину (часть 2)	1
1.1. Сильная связность	1
1.2. Рёберная двусвязность	1
1.3. Вершинная двусвязность	2
1.4. Эйлеровость	3
1.5. 2-SAT	4
1.5.1. Решение 2-SAT за $O(n + m)$	5
2. Введение в теорию сложности	7
2.1. Основные классы	7
2.2. NP (non-deterministic polynomial)	8
2.3. NP-hard, NP-complete	8
2.4. Сведения, новые NP-полные задачи	9
2.5. Задачи поиска	10
2.6. Гипотезы	11
2.7. Дерево сведений	11
3. Рандомизированные алгоритмы	11
3.1. Определения: RP, coRP, ZPP	12
3.2. Примеры	13
3.3. $ZPP = RP \cap coRP$	14
3.4. Двусторонняя ошибка, класс BPP	14
3.5. Как ещё можно использовать случайные числа?	15
3.6. Парадокс дней рождений. Факторизация: метод Полларда	15
3.7. 3-SAT и random walk	17
3.8. Лемма Шварца-Зишеля	18
3.9. Random shuffle	18
3.10. Дерево игры [stanford.edu]	19
3.11. (*) Квадратный корень по модулю	20
4. Модели вычислений и диагонализация	21
4.1. Машина Тьюринга	21
4.2. Недетерминизм и рандом	21
4.3. RAM-машина	21
4.4. Строгое и слабое полиномиальное время	22
4.5. Диагонализация: доказательство теоремы о временной иерархии	22
5. Кратчайшие пути	23
5.1. Short description	24

5.2. bfs	25
5.3. Модификации bfs	25
5.3.1. 1-k-bfs	25
5.3.2. 0-1-bfs	26
5.4. Дейкстра	26
5.5. A* (A-звездочка)	27
5.6. Флойд	28
5.6.1. Восстановление пути	28
5.6.2. Поиск отрицательного цикла	28
6. Кратчайшие пути	29
6.1. Алгоритм Форд-Беллмана	30
6.2. Выделение отрицательного цикла	30
6.3. Модификации Форд-Беллмана	31
6.3.1. Форд-Беллман с break	31
6.3.2. Форд-Беллман с очередью	31
6.3.3. Форд-Беллман с random shuffle	32
6.4. Потенциалы Джонсона	33
6.5. Цикл минимального среднего веса	34
6.6. Алгоритм Карпа	34
6.7. (*) Алгоритм Гольдберга	35
6.7.1. Решение за $\mathcal{O}(VE)$	35
6.7.2. Решение за $\mathcal{O}(EV^{1/2})$	36
6.7.3. Общий случай	36
7. DSU, MST и Йен	37
7.1. DSU: Система Непересекающихся Множеств	38
7.1.1. Решения списками	38
7.1.2. Решения деревьями	38
7.1.3. Оценка $\mathcal{O}(\log^* n)$	40
7.2. (*) Оценка $\mathcal{O}(\alpha^{-1}(n))$	40
7.2.1. Интуиция и $\log^{**} n$	40
7.2.2. Введение обратной функции Аккермана	41
7.2.3. Доказательство	41
7.3. MST: Минимальное Остовное Дерево	43
7.3.1. Алгоритм Краскала	43
7.3.2. Алгоритм Прима	43
7.3.3. Алгоритм Борувки	43
7.3.4. Сравнение алгоритмов	43
7.3.5. Лемма о разрезе и доказательства	44
7.4. Алгоритм Йена	44
8. Жадность и приближённые алгоритмы	46
8.1. Приближённые алгоритмы	46
8.2. Коммивояжёр	46
8.2.1. 2-ОПТ через MST	46
8.2.2. 1.5-ОПТ через MST (Кристофидес)	46

8.3. Хаффман	47
8.3.1. Хранение кодов	48
8.4. Компаратор и сортировки	48
8.4.1. Задача про 2 станка	50
8.4.2. Выбор максимума	50
9. Приближённые алгоритмы	52
9.1. Рюкзаки	52
9.2. Схемы приближений	52
9.2.1. Partition	52
9.2.2. Knapsack	53
9.2.3. Bin packing	54
9.3. Set cover	55
10. Центроидная декомпозиция	57
10.1. Построение и минимум на пути дерева	57
10.2. Реализация	58
11. BST и AVL	60
11.1. BST, базовые операции	60
11.2. Немного кода	61
11.3. AVL (Адэльсон-Вельский, Ландис'1962)	62
11.4. Split/Merge	63
11.5. Персистентность	65
11.6. Дополнительные операции, групповые операции	65
11.7. Неявный ключ	66
11.8. Reverse на отрезке	68
11.9. Итоги	68
11.10. Персистентность: итоги	68
12. B-tree и Treap	68
12.1. B-дерево (Bayer, McCreight'1972)	69
12.1.1. Поиск по B-дереву	69
12.1.2. Добавление в B-дерево	69
12.1.3. Удаление из B-дерева	69
12.1.4. Модификации	70
12.1.5. Split/Merge	70
12.2. Производные B-деревя	70
12.2.1. 2-3-дерево (Hopcroft'1970)	70
12.2.2. 2-3-4-дерево и RB-дерево (Bayer'1972)	70
12.2.3. AA-дерево (Arne Anderson'1993)	71
12.3. Treap (Seidel, Aragon'1989)	71
12.4. Операции над Treap	72
12.5. Дополнение о персистентности	73
12.5.1. Offline	73
12.5.2. Персистентная очередь за $\mathcal{O}(1)$	73
12.5.3. Простой персистентный дек (pairing)	74

12.5.4. Treap и избавление от Y	74
12.5.5. (*) Частичная персистентность: fat nodes	75
13. Splay и корневая оптимизация	76
13.1. Rope	77
13.2. Skip-list	77
13.3. Splay tree	78
13.4. Статическая оптимальность	79
13.5. Splay Tree и оптимальность	80
13.6. SQRT decomposition	80
13.6.1. Корневая по массиву	80
13.6.2. Корневая через split/merge	81
13.6.3. Корневая через split/rebuild	81
13.6.4. Применение	82
13.6.5. Оптимальный выбор k	83
13.6.6. Корневая по запросам, отложенные операции	83
13.7. Другие деревья поиска	83
14. Дерево отрезков	84
14.1. Общие слова	85
14.2. Дерево отрезков с операциями снизу	85
14.3. Дерево отрезков с операциями сверху	86
14.4. (*) Хаки для памяти к дереву отрезков сверху	88
14.5. Динамическое дерево отрезков и сжатие координат	88
14.6. 2D-деревья	89
14.7. Сканирующая прямая	90
14.8. k -я порядковая статистика на отрезке	91
14.9. (*) Fractional Cascading	92
14.10. (*) КД-дерево	92
15. LCA & RMQ	93
15.1. RMQ & Sparse table	94
15.2. LCA & Двоичные подъёмы	95
15.3. RMQ ± 1 за $\langle n, 1 \rangle$	95
15.4. LCA \rightarrow RMQ ± 1 и Эйлеров обход	96
15.5. RMQ \rightarrow LCA	97
15.6. LCA в offline, алгоритм Тарьяна	97
15.7. LA (level ancestor)	98
15.8. Euler-Tour-Tree	98
15.9. (*) LA, быстрые решения	99
15.9.1. (*) Вишкин за $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$	99
15.9.2. (*) Ladder decomposition + четыре русских	100
16. HLD & LC	101
16.1. Heavy Light Decomposition	101
16.2. Link Cut Tree	103
16.3. MST за $\mathcal{O}(n)$	105

16.4. (*) RMQ Offline 106

Лекция #1: Поиск в глубину (часть 2)

18 января

1.1. Сильная связность

Def 1.1.1. *Орграф сильносвязен, если для \forall вершин a, b есть путь $a \rightsquigarrow b$.*

Утверждение 1.1.2. Сильная связность – отношение эквивалентности на вершинах.

Нам интересно выделить классы эквивалентности, компоненты сильной связности.

Наивный алгоритм для \forall вершины a может за $\mathcal{O}(E)$ запустить два dfs-а – по прямым и обратным рёбрам, получить компоненту вершины a . Время работы $\mathcal{O}(VE)$.

• Нахождение компонент за $\mathcal{O}(V + E)$

Нам, как и наивному решению, понадобятся оба графа – и по прямым, и по обратным рёбрам.

1. Возьмём алгоритм поиска топсорта, запустим по прямым рёбрам. Наш граф может содержать циклы, поэтому топсорта может не быть, но массив `order` всё равно полезен.

2. В порядке $v \in \text{order}$, если v не отнесена ни к какой компоненте, запустим от неё `dfs` по обратным рёбрам. Покрашенное множество вершин – компонента v . По уже покрашенным `dfs` не ходит.

• Корректность

Искали компоненту вершины u . Хотя бы всю компоненту мы обошли... почему мы не взяли слишком много? Могли взять вершину v , достижимую по обратным рёбрам, но не достижимую по прямым. Рассмотрим \forall такую v , у неё есть компонента $C(v)$, утверждается, что хотя бы одну вершину $a \in C(v)$ мы рассмотрели в `order` до u , поэтому вся $C(v)$ была уже покрашена \Rightarrow `dfs(u)` в неё не заходил.

Lm 1.1.3. Если $v \rightsquigarrow u$, но v и u в разных компонентах, $\exists a \in C(v): \text{pos}_a < \text{pos}_u$

Доказательство. Достаточно доказать для соседних компонент в конденсации. Пусть v' – первая вершина в $C(v)$, до которой дошёл `dfs`. Аналогично u' – первая в $C(u)$. Если мы сперва посетили u' , в v' из неё мы не дойдём – победа, в `order` она будет записана позже. Если мы сперва посетили v' – мы не выйдем из неё, пока не посетим все вершины $C(v) \Rightarrow$ и пройдем по ребро, ведущее в $C(u) \Rightarrow$ обойдем всю $C(u) \Rightarrow$ из u выйдем до того, как выйдем из v .

Итого: v' – искомая вершина a . ■

1.2. Рёберная двусвязность

Def 1.2.1. *Рёберная двусвязность – отношение эквивалентности на вершинах неорграфа.*

Классы эквивалентности – компоненты рёберной двусвязности. Две вершины рёберно двусвязны, если между ними есть два рёберно не пересекающихся пути.

Теорема 1.2.2. Рёберная двусвязность – отношение эквивалентности.

Доказательство. Рефлексивность и симметричность очевидны.

Транзитивность: (u, v) и (v, w) рёберно двусвязны, доказываем двусвязность (u, w) .

Возьмём два пути $u \leftrightarrow v$, они образуют рёберно простой цикл C .

Пойдём двумя путями из w в v , остановимся, когда дойдём до цикла: мы на цикле в вершинах a и b , чтобы из них попасть в u – выберем нужное направление и пойдём по циклу C . ■

Def 1.2.3. Ребро (a, b) называется мостом, если после его удаления a и b не связны.

Замечание 1.2.4. После удаления из графа мостов, компоненты связности и рёберной двусвязности будут совпадать. Доказательство: $((a, b) - \text{не мост}) \Leftrightarrow (a \text{ и } b \text{ рёберно двусвязны})$.

Тривиальный алгоритм поиска мостов за $\mathcal{O}(VE)$: заметим, что любой мост лежит в остовном дереве \Rightarrow dfs-ом найдём любой остов, каждое ребро остова проверим за $\mathcal{O}(E)$.

• Алгоритм поиска мостов за $\mathcal{O}(V + E)$

```

1 void dfs(int v, int parent) {
2     stack.push(v)
3     time[v] = min_time[v] = ++T;
4     for (int x : adjacent[v])
5         if (x != parent) {
6             if (time[x] == 0) { // древесное ребро
7                 int stack_level = stack.size();
8                 dfs(x, v);
9                 if (min_time[x] > time[v]) { // мост!
10                    vector<int> component;
11                    while (stack.size() > stack_level)
12                        component.push(stack.pop())
13                }
14                relaxMin(min_time[v], min_time[x]);
15            } else { // обратное или двойственное ему ребро в другую сторону
16                relaxMin(min_time[v], time[x]);
17            }
18        }
19    }

```

Строки 2, 7 и 10-12 нужны, чтобы восстановить компоненты в том же dfs. Если мосты находятся корректно, то в 10-12 мы откусываем компоненту, которая висит на ребре $v \rightarrow x$.

$\text{time}[v]$ – время входа в вершину v .

$\text{min_time}[v]$ – минимальное достижимое время входа из v по путям, которые спускаются по древесным рёбрам dfs, а потом делают один шаг по произвольному ребру.

Теорема 1.2.5. Алгоритм корректно находит все мосты

Доказательство. Если ребро $v \rightarrow x$ мост, то из x нельзя достичь $v \Rightarrow \text{min_time}[x] > \text{time}[v]$.

С другой стороны, в неорграфе dfs не производит перекрёстных рёбер, если $v \rightarrow x$ не мост, то из поддерева x есть путь, заканчивающийся на обратное ребро, ведущее в v или выше $\Rightarrow \text{min_time}[x] \leq \text{time}[v]$. ■

Замечание 1.2.6. Рёберная двусвязность – отношение эквивалентности \Rightarrow относительно него граф можно сконденсировать. Получится лес, вершинами которого являются компоненты рёберной двусвязности, а рёбрами – мосты.

1.3. Вершинная двусвязность

Def 1.3.1. Вершинная двусвязность – отношение эквивалентности на рёбрах неорграфа. Классы эквивалентности – множества рёбер, компоненты вершинной двусвязности. Два ребра вершинно двусвязны, если есть вершинно простой цикл, содержащий оба ребра.

Теорема 1.3.2. Вершинная двусвязность – отношение эквивалентности.

Доказательство. Рефлексивность и симметричность очевидны.

Транзитивность: (e_1, e_2) и (e_2, e_3) вершинно двусвязны, доказываем двусвязность (e_1, e_3) .

Возьмём простой цикл C через e_1 и e_2 . Пойдём двумя путями из концов e_3 в концы e_2 , остановимся, когда дойдём до цикла: мы на цикле в различных вершинах a и b , чтобы из них попасть в концы e_1 , выберем нужное направление и пойдём по циклу C . ■

Def 1.3.3. Точка сочленения – вершина, при удалении которой увеличивается число компонент связности. Утверждение: точки сочленения – ровно те вершины, у которых есть смежные рёбра из разных компонент вершинной двусвязности.

Тривиальный алгоритм поиска точек сочленения за $\mathcal{O}(VE)$: проверим каждую вершину за $\mathcal{O}(E)$. Чтобы за $\mathcal{O}(VE)$ найти компоненты, можно построить граф на рёбрах, при построении графа запретить ходить через точки сочленения.

• Алгоритм поиска точек сочленения и компонент двусвязности за $\mathcal{O}(V + E)$

```

1 void dfs(int v, int parent) {
2   time[v] = min_time[v] = ++T;
3   int count = 0;
4   for (int x : adjacent[v])
5     if (x != parent) {
6       int stack_level = stack.size();
7
8       if (ещё не ходили по ребру x,v)
9         stack.push({v, x})
10      if (time[x] == 0) {
11        dfs(x, v);
12        if (min_time[x] >= time[v]) { // новая компонента
13          vector<Edge> component;
14          while (stack.size() > stack_level)
15            component.push(stack.pop())
16          if (parent != -1 || ++count >= 2)
17            ; // точка сочленения
18        }
19        relaxMin(min_time[v], min_time[x]);
20      } else {
21        relaxMin(min_time[v], time[x]);
22      }
23    }
24 }
```

TODO

1.4. Эйлеровость

Def 1.4.1. Рёберно простой цикл/путь *эйлеров*, если содержит все рёбра графа.

Def 1.4.2. Граф *эйлеров*, если содержит эйлеров цикл.

Теорема 1.4.3. Связный неорграф эйлеров iff все степени вершин чётны.

Доказательство. Возьмём \forall цикл, удалим его рёбра, в оставшихся компонентах связности по индукции есть эйлеровы циклы, соединим всё это счастье. Чтобы найти \forall цикл, возьмём \forall вершину v , пойдём жадно вперёд, удаляя пройденные рёбра: поскольку все степени чётны, остановиться мы можем только в v . ■

• Алгоритм построения

Будем делать всё, как в доказательстве теоремы. Сперва найдём \forall цикл.

```

1 vector<set<int>> g; // наивный способ хранения, удобно удалять рёбра
2 void dfs( int v ) {
3     if (!g[v].empty()) {
4         int x = *g[v].begin();
5         g[v].erase(x), g[x].erase(v);
6         dfs(x);
7         answer.push_back(edge(v, x));
8     }
9 }

```

Слово dfs уже намекает на рекурсию. Чтобы в оставшихся компонентах связности найти эйлеровы циклы и вставить в наш в нужные места, сделаем +1 рекурсивный вызов...

```

1 vector<set<int>> g; // наивный способ хранения, удобно удалять рёбра
2 void dfs( int v ) {
3     while (!g[v].empty()) { // единственное изменение кода
4         int x = *g[v].begin();
5         g[v].erase(x), g[x].erase(v);
6         dfs(x); // dfs(x) найдёт цикл одной из компонент и вставит ровно сюда
7         answer.push_front(edge(v, x));
8     }
9 }

```

Чтобы алгоритм работал за линию, нам бы от set-а избавиться. Будем удалять лениво:

```

1 struct Edge { int a, b, isDeleted; };
2 vector<Edge> edges;
3 vector<vector<int>> ids; // каждое ребро встретится два раза

```

В цикле while вынимаем все рёбра, пропускаем с пометкой isDeleted.

Идя из вершины v по ребру e , мы попадём в вершину $e.a + e.b - v$.

Lm 1.4.4. Слабосвязный орграф эйлеров iff у всех вершин входящая степень равна исходящей.

Доказательство и алгоритм аналогичны. Реализация алгоритма даже проще: орребро не нужно удалять из вектора другой вершины \Rightarrow для хранения графа достаточно `vector<vector<Edge>`.

1.5. 2-SAT

Задача **2-SAT**: нам дана **КНФ** формула φ , в каждом клозе не более двух литералов. Нужно найти выполняющий набор x_i , или сказать, что φ противоречива. Пример:

$$\varphi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_3) \longrightarrow (x_1 = 1, x_2 = 0, x_3 = 1)$$

Примеры задач, которые можно решить, используя 2-SAT:

• 2-List-Coloring

Дан неорграф граф, для каждой вершины v есть список $l[v]$ из двух цветов.

Покрасить вершины так, чтобы соседние были покрашены в разные цвета.

Сведение: x_v – номер выбранного цвета в $l[v]$, каждое ребро (a, b) для каждого цвета $c \in l[a] \cap l[b]$ задаёт кюз $\neg(x_a = pos_{a,c} \wedge x_b = pos_{b,c}) \Leftrightarrow (x_a = \overline{pos_{a,c}} \vee x_b = \overline{pos_{b,c}})$

• Расположение геометрических объектов без наложений

Если для каждого объекта есть только два варианта расположения, то задача расположения объектов без пересечений сводится к 2-SAT. Пример: есть множество отрезков на плоскости, каждому нужна подпись с одной из двух сторон отрезка.

• Кластеризация на два кластера

Даны объекты, и матрица расстояний d_{ij} (непохожести объектов). Нужно разбить объекты на два множества: \max из диаметров $\rightarrow \min$. Решение: бинпоиск по ответу, внутри 2-SAT.

1.5.1. Решение 2-SAT за $\mathcal{O}(n + m)$

Кюз $(a \vee b) \Leftrightarrow$ двум импликациям $(\neg a \Rightarrow b)$ и $(\neg b \Rightarrow a)$.

Построим граф следствий. Вершины графа – литералы x_i и $\neg x_i$.

Для каждого кюза $(a \vee b)$ проведём рёбра $\neg a \rightarrow b$ и $\neg b \rightarrow a$.

Если n – число неизвестных, m – число кюз, мы получили $2n$ вершин и $2m$ рёбер.

Lm 1.5.1. Решение 2-SAT – для каждого i в графе следствий выбрать ровно одну из двух переменных $x_i, \neg x_i$ так, чтобы не было рёбер из выбранных в невыбранные.

Lm 1.5.2. Путь в графе следствий $x_i \rightsquigarrow \neg x_i$, означает что в любом корректном решении $x_i = 0$.

Следствие 1.5.3. $\exists i: x_i$ и $\neg x_i$ в одной компоненте сильной связности \Rightarrow формула противоречива

Теорема 1.5.4. $\exists i: x_i$ и $\neg x_i$ в одной компоненте сильной связности \Leftrightarrow формула противоречива

Доказательство. Пусть $\forall i x_i$ и $\neg x_i$ в разных компонентах. Алгоритм расстановки значений x_i :

1. Топологически отсортируем конденсацию графа.
2. $k[x_i]$ – номер компоненты в топологическом порядке. \forall ребра графа $a \rightarrow b$ верно $k[a] \leq k[b]$.
3. Для каждой переменной i выберем x_i iff $k[x_i] > k[\neg x_i]$.

Корректность решения:



Если в решение есть противоречие, то по 1.5.1 есть выбранный y , невыбранный z и ребро $y \rightarrow z$. Рёбра добавлялись парами \Rightarrow есть и ребро $\neg z \rightarrow \neg y$. Из \exists -я этих рёбер делаем вывод: $k[z] \geq k[y] \wedge k[\neg y] \geq k[\neg z]$. Из выбранности y , не выбранности z вывод: $k[y] > k[\neg y] \wedge k[\neg z] > k[z]$.

Итого мы получили решение а $\mathcal{O}(V + E)$: построить граф + найти ксс + вывести ответ. Заметьте, сами компоненты хранить не нужно, мы строим только массив $k[]$.

• Упрощение реализации

Предполагая, что ксс мы ищем через два dfs-а, можно модифицировать второй из них:

```
1 void paintComponent( int v ) {
2     used[v] = 1;
```

```
3  if (x[v / 2] == -1) // пусть вершины  $x_i$ ,  $\neg x_i$  идут парами; -1 значит неопределено
4      x[v / 2] = 1 - v % 2; // для  $x_{v/2}$  выберем не  $v$ , так как компоненту  $v$  перебираем раньше
5  for (int x : graph[v])
6      if (!used[x])
7          paintComponent(x);
8  }
```

Лекция #2: Введение в теорию сложности

20 января

2.1. Основные классы

• Алгоритмически не разрешимые задачи

Задач таких **много**¹, **much enough**², **see also**³.

Можно выделить такую общую тему, как “предсказание будущего сложного процесса”.

Например, “в игре «жизнь» достижимо ли из состояния A состояние B ?”. Или не в игре...

Наиболее каноническим примером является «проблема останова» (halting problem):

Дана программа, остановится ли она когда-нибудь на данном входе?

Теорема 2.1.1. Halting problem алгоритмически не разрешима.

Доказательство. От противного. Пусть есть алгоритм `terminates(code, x)`, всегда останавливающийся, и возвращающий `true` iff `code(x)` останавливается. Рассмотрим программу:

```
1 def invert(code):
2   if terminates(code, code): while (true)
```

Если `invert(invert)` останавливается, то должен зависнуть, и наоборот. Противоречие. ■

• Decision/search problem

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

Иначе это search problem (задача поиска). Примеры:

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

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

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

Def 2.1.2. $\text{DTIME}(f(n))$ – множество задач распознавания, для которых $\exists C > 0$ и детерминированный алгоритм, работающий на всех входах не более чем $C \cdot f(n)$.

Def 2.1.3. $P = \bigcup_{k>0} \text{DTIME}(n^k)$. Т.е. задачи, имеющие полиномиальное решение.

Def 2.1.4. $\text{EXP} = \bigcup_{k>0} \text{DTIME}(2^{n^k})$. Т.е. задачи, имеющие экспоненциальное решение.

Теорема 2.1.5. $\text{DTIME}(f(n)) \subsetneq \text{DTIME}(f(n) \log^2 f(n))$

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

Следствие 2.1.6. $P \neq EXP$

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

2.2. NP (non-deterministic polynomial)

Def 2.2.1. $NP = \{L: \exists M, \text{ работающий за полином от } |x|, \forall x ((\exists y M(x, y) = 1) \Leftrightarrow (x \in L))\}$

Неформально. NP – класс языков $L: \forall x \in L$, если нам дадут подсказку $y(x)$, мы за полином сможем убедиться, что $x \in L$. Напомним, “язык” \equiv “decision задача”.

Подсказку y так же называют свидетелем того, что x лежит в L .

Def 2.2.2. $coNP = \{L: \bar{L} \in NP\}$

Def 2.2.3. $coNP = \{L: \exists M, \text{ работающий за полином от } |x|, \forall x ((\forall y M(x, y) = 0) \Leftrightarrow (x \in L))\}$

Def 2.2.4. $coNP = \{L: \exists M, \text{ работающий за полином от } |x|, \forall x ((\exists y M(x, y) = 1) \Leftrightarrow (x \notin L))\}$

Неформально: дополнение языка лежит в NP или “ \exists свидетель того, что $x \notin L$ ”.

• Примеры:

1. **HAMPATH** = $\{G \mid G - \text{неорграф, в котором есть гамильтонов путь}\}$. Подсказка y – путь.

M получает вход $x = G$, подсказку y , проверяет, что y прост, $|y| = n \wedge \forall e \in y \Rightarrow e \in G$.

2. **k-CLIQUE** – проверить наличие в графе клики размером k .

3. **IS-SORTED** – отсортирован ли массив? Лежит даже в P .

4. **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 2.2.5. $P \subseteq NP$ (можно взять пустую подсказку).

Замечание 2.2.6. Вопрос $P = NP$ или $P \neq NP$ остаётся открытым ([wiki](#)). Предполагают, что \neq .

Def 2.2.7. **BH = BOUNDED-HALTING:** вход $x = \langle \underbrace{11 \dots 1}_k, M, x \rangle$, проверить, \exists ли такой y : $M(x, y)$ остановится за k шагов и вернёт *true*.

$BH \in NP$. Подсказка – такой y . Алгоритм – моделирование k шагов M за $\mathcal{O}(\text{poly}(k))$.

Важно, что если бы число k было записано, используя $\log_2 k$ бит, моделирование работало бы за экспоненту от длины входа, и нельзя было бы сказать “задача лежит в NP”.

2.3. NP-hard, NP-complete

Def 2.3.1. \exists полиномиальное сведение задачи A к задаче B : $(A \leq_P B) \Leftrightarrow \exists$ алгоритм f , работающий за полином, $(x \in A) \Leftrightarrow (f(x) \in B)$

Замечание 2.3.2. f работает за полином $\Rightarrow |f(x)|$ полиномиально ограничена $|x|$

Def 2.3.3. \exists сведение по Куку задачи A к задаче B : $(A \leq_C B) \Leftrightarrow \exists M$, решающий A , работающий за полином, которому разрешено обращаться за $\mathcal{O}(1)$ к решению B .

Ещё говорят “задача A сводится к задаче B ”.

В обоих сведениях мы решаем задачу A , используя уже готовое решение задачи B .

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 2.3.4. $NP\text{-hard} = NPh = \{L: \forall A \in NP \Rightarrow A \leq_P L\}$

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

Def 2.3.5. NP-complete = NPc = NPh \cap NP

NP-полные задачи – самые сложные задачи в классе NP.

Если мы решим хотя бы одну из NPc за полином, то решим все из NP за полином.

Хорошая новость: все NP-полные по определению сводятся друг к другу за полином.

Замечание 2.3.6. Когда хотите выразить мысль, что задача трудная в смысле решения за полином (например, поиск гамильтонова пути), **неверно** говорить “это NP задача” (любая из P тоже в NP) и странно говорить “задача NP-полна” (в этом случае вы имеете в виду сразу, что и трудная, и в NP). Логично сказать “задача NP-трудна”.

Lm 2.3.7. $A \leq_P B, B \in P \Rightarrow A \in P$

Доказательство. Сведение f работает за n^s , B решается за $n^t \Rightarrow A$ решается за n^{st} . ■

Lm 2.3.8. $A \leq_P B, A \in NPh \Rightarrow B \in NPh$

Доказательство. $\forall L \in NP (\exists f: L$ сводится к A функцией $f(x)) \wedge (A \leq_P B$ функцией $g(x)) \Rightarrow L$ сводится к B функцией $g(f(x))$ (за полином). ■

• NP-полные задачи существуют!

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

Теорема 2.3.9. BH = BOUNDED-HALTING \in NPc

Доказательство. NPc = NP \cap NPh. Мы уже показали 2.2.7, что BH лежит в NP.

Теперь покажем, что BH \in NPh. Для этого нужно взять $L \in NP$ и свести его к BH.

Пусть $L \in NP \Rightarrow \exists$ полиномиальный M , проверяющий подсказки для L .

Полиномиальный $\Leftrightarrow \exists P(n)$, ограничивающий время работы $M \Rightarrow \exists f(x) = \langle \underbrace{11 \dots 1}_{P(|x|)}, M, x \rangle$.

Получили полиномиальное сведение: $(x \in L) \Leftrightarrow (f(x) \in BH)$.

Заметьте, зная L , мы не умеем предъявить ни M , ни f , мы лишь знаем, что $\exists M, f$. ■

2.4. Сведения, новые NP-полные задачи

Началось всё с того, что в 1972-м Карп опубликовал список из 21 полной задачи, и дерево сведений. Кстати, в [его работе \(pdf\)](#) все сведения крайне лаконичны. Итак, приступим:

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

Чтобы доказать, что $B \in NPc$, нужно ещё не забыть проверить, что $B \in NP$.

Во всех теоремах ниже эта проверка очевидна, мы проведём её только в доказательстве первой.

• BH \rightarrow CIRCUIT-SAT \rightarrow SAT \rightarrow 3-SAT \rightarrow k -INDEPENDENT \rightarrow k -CLIQUE

Def 2.4.1. CIRCUIT-SAT. Дана схема, состоящая из входов, выхода, гэйтов AND, OR, NOT. Проверить, существует ли набор значений на входах, дающий true на выходе.

Теорема 2.4.2. CIRCUIT-SAT \in NPc

Доказательство. Подсказка – набор значений на входах \Rightarrow CIRCUIT-SAT \in NP.

Сводим ВН к CIRCUIT-SAT \Rightarrow нам даны программа M , время выполнения 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 .

$x_{0,j}$ – вход, $x_{t,output}$ – выход, $\forall i \in [1, t)$ $x_{i,j}$ зависит от $\mathcal{O}(1)$ переменных $(i-1)$ -го слоя.

Сейчас значение x_{ij} – произвольная булева формула f_{ij} от $\mathcal{O}(1)$ переменных из слоя x_{i-1} .

Перепишем f_{ij} в КНФ-форме, чтобы получить гейты вида AND, OR, NOT. Получили $\mathcal{O}(tn)$ булевых гейтов \Rightarrow по (M, t, x) за полином построили вход к CIRCUIT-SAT. ■

Теорема 2.4.3. SAT \in NPC

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

Теорема 2.4.4. 3-SAT \in NPC

Доказательство. Пусть есть кловз $(x_1 \vee x_2 \vee \dots \vee x_n), n \geq 4$.

Введём новую переменную w и заменим его на $(x_1 \vee x_2 \vee w) \wedge (x_3 \vee \dots \vee x_n \vee \bar{w})$. ■

Теорема 2.4.5. k-INDEPENDENT \in NPC

Доказательство. Наша формула – m кловзов $(l_{i1} \vee l_{i2} \vee l_{i3})$, где l_{ij} – литералы.

Построим граф из ровно $3m$ вершин – l_{ij} . $\forall i$ добавим треугольник (l_{i1}, l_{i2}, l_{i3}) (итого $3m$ рёбер).

В любое независимое множество входит максимум одна вершина из каждого треугольника.

$\forall k = 1..n$ соединим все вершины $l_{ij} = x_k$ со всеми вершинами $l_{ij} = \bar{x}_k$.

Теперь $\forall k = 1..n$ в независимое множество нельзя одновременно включить x_k и \bar{x}_k .

Итог: \exists независимое размера $m \Leftrightarrow$ у 3-SAT было решение. ■

Теорема 2.4.6. 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: $c'_{ij} = \bar{c}_{ij} \wedge (i \neq j)$. ■

2.5. Задачи поиска

Def 2.5.1. $\overline{\text{NP}}$, $\overline{\text{NPC}}$, $\overline{\text{NPh}}$ – аналогичные классы для задач поиска подсказки.

• Сведение задач минимизации, максимизации к decision задачам

Пусть мы умеем проверять, есть ли в графе клика размера k .

Чтобы найти размер максимальной клики, достаточно применить бинпоиск по ответу.

Это общая техника, применимая для максимизации/минимизации численной характеристики.

• Сведение search задач к decision задачам: выполняющий набор для SAT

Пусть $M(\varphi)$ проверяет выполнимость формулы φ .

$\varphi[x_i=e]$ – формула, полученная из φ подстановкой значения e в x_i . Индукция:

while $n > 0$:

 if $M(\varphi[x_n=0]) = 1$ then $r_n = 0$ else $r_n = 1$

$\varphi \leftarrow \varphi[x_n=r_n]$; $n--$

Теперь возьмём \forall задачу $L \in \overline{NP}$. Её decision версию можно полиномиально свести к SAT ($x \rightarrow f(x)$), для которого мы умеем восстанавливать выполняющий набор y . Попробуем из этого набора построить ответ к исходной задаче ($g(y) \leftarrow y$). **Пример:** чтобы найти решение для k -INDEPENDENT, возьмём в ответ ровно те вершины, которым соответствуют $x_i = \text{true}$.

Замечание 2.5.2. Конечно, ответ для k -INDEPENDENT можно было восстановить напрямую: пытаемся жадно взять вершину 1 и проверить $(k-1)$ -INDEPENDENT для $V \setminus N(1)$.

• **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**.

2.6. Гипотезы

Гипотеза 2.6.1. $P \neq NP$

Гипотеза 2.6.2. ETH (exponential time hypothesis): \nexists решения за $2^{o(n)}$ для 3-SAT

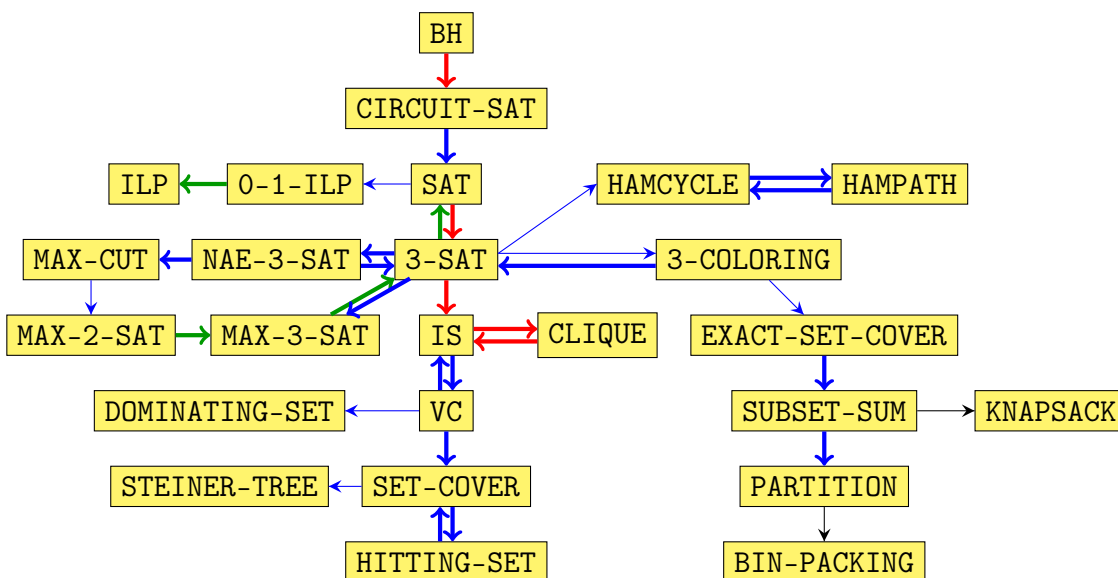
Гипотеза 2.6.3. SETH (strong ETH): $\forall \epsilon \exists k: \nexists$ решения за $(2 - \epsilon)^n$ для k -SAT

Lm 2.6.4. SETH \Rightarrow ETH $\Rightarrow P \neq NP$

2.7. Дерево сведений

Легенда:

- **Красное** – было на лекции.
- **Синее** – было в задачах практики/дз, нужно знать на экзамене.
- **Бледносинее** – было в задачах практики/дз, не будет на экзамене.
- **Зелёное** – очевидное вложение.
- Чёрное – будет доказано и использовано в будущем.



Лекция #3: Рандомизированные алгоритмы

27 января

3.1. 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:

$$\text{Def 3.1.1. } \text{RP} = \{L: \exists M \in \text{PTime} \left\{ \begin{array}{l} x \notin L \Rightarrow M(x, y) = 0 \\ x \in L \Rightarrow \Pr_y[M(x, y) = 1] \geq \frac{1}{2} \end{array} \right\}$$

x – вход, y – подсказка из случайных бит. Расшифровка: RP = randomized polynomial time.

То есть, если $x \notin L$, M не ошибается, иначе работает корректно с вероятностью хотя бы $\frac{1}{2}$.

Заметим, если для какого-то y алгоритм M вернул 1, то это точно правильный ответ, $x \in L$.

$$\text{Def 3.1.2. } \text{coRP} = \{L: \exists M \in \text{PTime} \left\{ \begin{array}{l} x \in L \Rightarrow M(x, y) = 1 \\ x \notin L \Rightarrow \Pr_y[M(x, y) = 0] \geq \frac{1}{2} \end{array} \right\}$$

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 для \forall подсказки выдадут 0. Если ответ 1, для NP-алгоритма \exists хотя бы одна подсказка, а RP-алгоритм должен корректно работать хотя бы на половине подсказок.

• Понижение ошибки

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

Lm 3.1.3. Пусть M – RP-алгоритм, ошибающийся с вероятностью p .

Запустим его k раз, если хотя бы раз вернул 1, вернём 1. Получили алгоритм с ошибкой p^k .

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

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

• ZPP (zero-error probabilistic polynomial time)

ZPP – класс задач, для которых есть никогда не ошибающийся вероятностный алгоритм с полиномиальным матожиданием временем работы.

$$\text{Def 3.1.4. } \text{ZPP} = \{L: \exists P(n), M \text{ такие, что } E_y[\text{Time}(M(x, y))] \leq P(|x|)\}$$

Про алгоритмы и для RP, и для ZPP говорят “вероятностные/рандомизированные”.

Чтобы подчеркнуть принадлежность классу, уточняют ошибку.

RP-алгоритм обладает односторонней ошибкой. ZPP-алгоритм работает без ошибки.

Мы определили основные классы только для задач распознавания, только для полиномиального времени. Можно определить аналогичные для задач поиска и для любого времени.

Важно помнить, что в RP, coRP, ZPP алгоритмы работают на *всех* тестах.

Например, вероятность ошибки в RP для любого теста не более $\frac{1}{2}$.

3.2. Примеры

• Поиск квадратичного невычета

Для любого простого p ровно $\frac{p-1}{2}$ ненулевых остатков обладают свойством $a^{\frac{p-1}{2}} \equiv -1 \pmod p$. Такие a называются “квадратичным невычетами”. *Задача:* дано p , найти невычет.

Алгоритм: пока не найдём, берём случайное a , проверяем.

Время одной проверки – возведение в степень ($\log p$ для $p \leq 2^w$).

$E_{random}[Time] = X = (\log p) + \frac{1}{2}X \Rightarrow X = 2 \log p$.

• Проверка на простоту: тест Ферма

Малая теорема Ферма: $\forall p \in \mathbb{P} \forall a \in [1..p-1] a^{p-1} \equiv 1 \pmod p$.

Чтобы проверить простоту p , можно взять случайное a и проверить $a^{p-1} \equiv 1 \pmod p$

Lm 3.2.1. $\exists a: a^{p-1} \not\equiv 1 \pmod p \Rightarrow$ таких a хотя бы $\frac{p-1}{2}$.

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

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 3.2.2. 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 p ) {
2     // p-1=2^s*t, t - нечётно
3     if ( p < 2 ) return 0; // 1 и 0 не простые
4     p --> ( s, t )
5     a = randInt( 2, p - 1 ) // [2, p - 1]
6     g = pow( a, t, p ) // возвели в степень по модулю
7     for ( int i = 0; i < s; i++ ) {
8         if ( g == 1 ) return 1;
9         if ( g mod p != -1 && g * g mod p == 1 ) return 0;
10        g = g * g mod p;
11    }
12    return g == 1; // тест Ферма
13 }
```

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

Утверждение 3.2.3. $\forall a$ вероятность ошибки не более $\frac{1}{4}$.

• Изученные алгоритмы

(*) k -я порядковая статистика учит нас:

“вместо того, чтобы выбирать медиану, достаточно взять случайный элемент”.

Поиск числа, которое встречается в массиве $>$ половины раз. 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 3.2.4. Вероятность ошибки не более $\frac{1}{2}$.

Доказательство. Пусть $AB \neq C \Rightarrow D = AB - C \neq 0$.

Посчитаем $\Pr_x [A(Bx) = Cx] = \Pr_x [Dx = 0]$. $\exists i, j: D_{i,j} \neq 0$. Зафиксируем произвольный x . $(Dx)_i = D_{i,0}x_0 + \dots + D_{i,j}x_j + \dots + D_{i,n-1}x_{n-1}$. $D_{i,j} \neq 0 \Rightarrow$ меняя значение x_j , меняем значение $(Dx)_i \Rightarrow \Pr_x [(Dx)_i = 0] = \frac{1}{2} \Rightarrow \Pr_x [Dx = 0] \leq \frac{1}{2}$. ■

3.3. ZPP = RP \cap coRP

Lm 3.3.1. Неравенство Маркова: $x \geq 0 \Rightarrow \forall a \Pr[x > aE(x)] < \frac{1}{a}$

Доказательство. Пусть $\Pr[x > aE(x)] \geq \frac{1}{a} \Rightarrow E(x) > (aE(x)) \cdot \frac{1}{a} + 0 \cdot (1 - \frac{1}{a}) = E(x) \text{ ???}$ ■

Теорема 3.3.2. ZPP = RP \cap coRP

Доказательство. Пусть $L \in \text{ZPP} \Rightarrow \exists$ алгоритм M , работающий в среднем за $P(n)$ ^{3.3.1} $\Rightarrow \Pr[\text{Time}(M) \leq 2P(n)] \geq \frac{1}{2} \Rightarrow$ запустим M с будильником на $2P(n)$ операций. Если алгоритм завершился до будильника, он даст верный ответ, иначе вернём 0 \Rightarrow RP или 1 \Rightarrow coRP.

Пусть $L \in \text{RP} \cap \text{coRP} \Rightarrow \exists M_1(\text{RP}), M_2(\text{coRP})$, работающие за полином, ошибающиеся в разные стороны, которые можно запускать по очереди $(M_1 + M_2)$. Пусть $x \notin L \Rightarrow M_1(x) = 0$, $\Pr[M_2(x) = 0] \geq \frac{1}{2}$. Сколько раз нужно в среднем запустить $M_1 + M_2$? $E \geq 1 + \frac{1}{2}E \geq 2$ (всегда запустим 1 раз, затем с вероятностью $\frac{1}{2}$ получим $M_2(x) = 1$ и повторим процесс с начала). ■

Lm 3.3.3. $\text{P} \subseteq \text{coRP} \cap \text{RP} = \text{ZPP} \subseteq \text{RP} \subseteq \text{NP}$

При этом строгие ли вложения неизвестно.

Зато известна масса задач, для которых есть простое RP-решение, но неизвестно P-решение.

3.4. Двусторонняя ошибка, класс BPP

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

Def 3.4.1. $\text{BPP} = \{L: \exists M \in \text{PTime} \left\{ \begin{array}{l} x \notin L \Rightarrow \Pr_y[M(x, y) = 0] \geq \alpha \\ x \in L \Rightarrow \Pr_y[M(x, y) = 1] \geq \alpha \end{array} \right\}, \text{ где } \alpha = \frac{2}{3}\}$.

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 3.4.2. О понижении ошибки. Пусть $\beta > 0, \alpha = \frac{1}{2} + \varepsilon, \varepsilon > 0 \Rightarrow \exists n = \text{poly}(\frac{1}{\varepsilon})$ 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 $\text{Err} = \sum_{i=0}^{i=k} p_i$, где p_i – вероятность того, что алгоритм ровно i раз вернул 1.

Выпишем в явном виде формулу для p_i , при этом удобно отсчитывать i от середины: $i = k - j$.

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

$$\text{Err} = \sum_{i=0}^{i=k} p_i \leq \sum_{i=0}^{i=k} \binom{n}{i} \left(\frac{1}{4} - \varepsilon^2\right)^k = \frac{1}{2} 2^n \left(\frac{1}{4} - \varepsilon^2\right)^k = \frac{1}{2} (1 - 4\varepsilon^2)^k$$

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

3.5. 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** такого числа не даст злоумышленнику ровно никакой информации. Боб восстановит $x = (y - r) \bmod m$.

• Вычисления без разглашения

В некой компании сотрудники стесняются говорить друг другу, какая у кого зарплата.

Зарплату i -го обозначим x_i . Сотрудники очень хотят посчитать среднюю зарплату \Leftrightarrow посчитать сумму $x_1 + \dots + x_n$. Для этого они предполагают, что сумма меньше $m = 10^{18}$, и пользуются следующим алгоритмом:

0. Первый сотрудник генерирует случайное число $r \in [0, m)$.
1. Первый сотрудник передаёт второму число $(r + x_1) \bmod m$.
2. Второй сотрудник передаёт третьему число $(r + x_1 + x_2) \bmod m$.
3. ...
- n. Последний сотрудник передаёт первому $(r + x_1 + x_2 + \dots + x_n) \bmod m$.
- *. Первый, как единственный знающий r , вычитает его и говорит всем ответ.

• Приближения

На практике будут разобраны $\frac{1}{2}$ -ОПТ приближения для MAX-3-SAT и VERTEX-COVER.

3.6. Парадокс дней рождений. Факторизация: метод Полларда

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

Lm 3.6.1. $1 - p_k \leq \frac{k(k-1)}{2n}$

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

Следствие 3.6.2. 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 3.6.3. $k = \sqrt{n} \Rightarrow 0.4 \leq p_k \leq 0.8$

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

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

$\forall k$ можно оценить p_k гораздо точнее:

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

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;

```

• 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;

```

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

• Поллард

Пусть y n минимальный делитель p , тогда $p \leq \sqrt{n}$. Сгенерируем $\sqrt[4]{n} \geq \sqrt{p}$ 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)

```

Почему так можно? Рассмотрим последовательность $x_i = f^i(x_0)$. Она заикливаясь по модулю p , что выглядит как буква «Ро». Наша цель – найти две точки друг над другом. Сделав достаточно много шагов из стартовой точки окажемся на цикле. После чего остаётся пройти ещё один раз весь цикл.

Полученный нами алгоритм 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.

Сейчас асимптотика $\mathcal{O}(n^{1/4} \cdot T(\gcd))$, чтобы получить чистое $\mathcal{O}(n^{1/4})$, воспользуемся тем, что $\gcd(a, n) = 1 \wedge \gcd(b, n) = 1 \Rightarrow \gcd(ab \bmod n, n) = 1 \Rightarrow$ можно вместо $\gcd(x-y, n)$ смотреть на группы по $\log n$ разностей: $\gcd(\prod_i (x-y_i), n)$, и только если какой-то $\gcd \neq 1$, проверять каждый y_i отдельно.

3.7. 3-SAT и random walk

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

Пусть решение X^* существует и в нём нулей больше чем единиц. Начнём с $X_0 = \{0, 0, \dots, 0\}$, 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 $X_0 = \{1, 1, \dots, 1\}$. Нужно перебрать оба варианта.

• Рандомизированное решение за 1.5^n

Упростим предыдущую идею: начнём со случайного X_0 , сделаем n шагов, выбирая каждый раз случайное из трёх направлений. Естественно вероятность успеха экспоненциально мала.

Анализ вероятностей. Если перебор за $3^{n/2}$ перебирал все варианты, то мы перебираем 1 вариант и угадываем с вероятностью $\geq p = \sum_k Pr[k] \frac{1}{3^k}$, где k – расстояние Хэмминга от X_0 до X^* , а $Pr[k] = \binom{n}{k} / 2^n$ – вероятность того, что при выборе случайного X_0 расстояние равно k .

Осталось заметить, что $p = (1 + \frac{1}{3})^n / 2^n = (\frac{2}{3})^n \Rightarrow$ повторим процесс 1.5^n раз.

Кстати, поскольку $Pr[k \leq \frac{n}{2}] = \frac{1}{2}$, достаточно делать даже не n , а $\frac{n}{2}$ шагов.

• Рандомизированное решение за 1.334^n

Schoning's algorithm (1999). Получается из предыдущего решения заменой $\frac{n}{2}$ шагов на $3n$ шагов. Вероятность успеха будет не менее $(3/4)^n \Rightarrow$ время работы $\mathcal{O}^*(1.334^n)$. [Доказательство. Статья](#). В той же статье дан более общий результат для k -SAT: $(\frac{2}{1+1/(k-1)})^n = (\frac{2(k-1)}{k})^n$.

Лучшее сегодня решение 3-SAT: [алгоритм PPSZ](#) за 1.3067^n от Paturi, Pudlak, Saks, Zani (2005).

• Поиск хороших кафешек

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

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

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

3.8. Лемма Шварца-Зиппеля

Lm 3.8.1. Пусть дан многочлен P от нескольких переменных над полем \mathbb{F} .

$$(P \neq 0) \Rightarrow \Pr_x [P(x) = 0] \leq \frac{\deg P}{|\mathbb{F}|}$$

Доказательство. На экзамене его не будет. Индукция по числу переменных.

База: многочлен от 1 переменной имеет не более $\deg P$ корней. Переход: [wiki](#). ■

Следствие 3.8.2. Задача проверки тождественного равенства многочлена нулю $\in \text{coRP}$.

Доказательство. Подставим случайный x в поле \mathbb{F}_q , где $(q - \text{простое}) \wedge (q > 2 \deg P)$. ■

• Совершенное паросочетание в произвольном графе

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

Def 3.8.3. Матрица Тамта $T: T_{ij} = -T_{ji}, T_{ij} = \begin{cases} 0 & c_{ij} = 0 \\ x_{ij} & c_{ij} = 1 \end{cases}$

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

Теорема 3.8.4. $\det T \neq 0 \Leftrightarrow \exists$ 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 году в неорграфе научились проверять наличие гамильтонова пути за $\mathcal{O}^*(1.657^n)$.

Можно почитать в оригинальной [статье](#) Бьёркланда (Björklund).

3.9. 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 3.9.1. После процедуры `Shuffle` все перестановки a равновероятны.

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.

3.10. Дерево игры [stanford.edu]

Пусть есть игра, в которую играют два игрока. Можно построить дерево игры (возможно бесконечное) – дерево всех возможных ходов. Теперь игра – спуск по дереву игры. Для некоторых простых случаев хорошо известны “быстрые” способы сказать, кто выиграет.

- **Игра на 0-1-дереве**

Задача: в листьях записаны числа 0 и 1, первый победил, если игра закончилась в 1, иначе победил второй. Пусть дерево – полное бинарное глубины n .

Решение: простой рандомизированный алгоритм «*сперва сделаем случайный ход, а второй сделаем только если не выиграли первым ходом*». Асимптотика $\mathcal{O}(1.69^n)$. Анализ в практике.

- **Игра на OR-AND-дереве**

Близкая по смыслу, но не игровая, а вычислительная задача. Дано полное бинарное дерево, в листьях 0 и 1. В промежуточных вершинах чередующиеся AND и OR гэты от детей.

Задача равносильна игре на 0-1-дереве: AND означает “чтобы первый выиграл, второй в обеих ветках должен проиграть”, OR означает “чтобы второй проиграл, первый в одной из веток должен выиграть”. Чередование AND/OR значит “ходят по очереди”.

- **Игра на min-max-дереве**

Полное бинарное дерево. В листьях записаны \mathbb{Z} числа из $[0, m)$.

Первый игрок максимизирует результат игры, второй минимизирует.

Решение #1: бинарный поиск по ответу и игра на 0-1-дереве

Решение #2: (*) **альфа-бетта отсечение**

Решение #3: (*) комбинация этих идей – алгоритм MTD-f ([статья](#), [wiki](#)). MTD-f в своё время стал прорывом в шахматных стратегиях.

3.11. (*) Квадратный корень по модулю

Задача: даны простое p и a , найти $x: x^2 = a \pmod p$.

Хорошо известны два решения: **алгоритм Тоннелли-Шенкса'1973** и **алгоритм Циполла'1907**.

Оба алгоритма вероятностны и требуют угадать квадратичный невычет.

В этой главе мы изучим только второй [1][2], как более быстрый и простой в реализации.

Алгоритм: возьмём $P(x) = (x + i)^{(p-1)/2} - 1$ и $A(x) = x^2 - a = (x - x_1)(x - x_2)$.

Посчитаем $\gcd(P(x), A(x))$, если он содержит ровно один из $x - x_1$ и $x - x_2$, мы победили.

Теорема 3.11.1. Вероятность успеха $\geq \frac{1}{2}$. Время работы – $\mathcal{O}(\log n)$ делений чисел порядка p .

Корни существуют \Leftrightarrow символ Лежандра $a^{(p-1)/2} = 1$. В реализации вместо \gcd будем вычислять $R(x) = P(x) \pmod A(x)$ – возведение в степень с умножением похожим на комплексные числа:

$$(p_1x + q_1)(p_2x + q_2) = (p_1q_2 + p_2q_1)x + (p_1p_2a + q_1q_2)$$

Пусть $R(x) = bx + c$, тогда при $b \neq 0$ пробуем корень $-cb^{-1}$.

```

1 def root(a, p):
2     if pow_mod(a, (p-1)/2, p) != 1: return -1 # символ Лежандра
3     while True:
4         i = random [0..p)
5         bx+c = pow_mod(x+i, (p-1)/2, x^2-a) # многочлен в степени по модулю
6         if b != 0:
7             z = -c/b
8             if z^2 = a \bmod p: return z

```

Доказательство. (3.11.1) Для начала посмотрим на символ Лежандра и заметим, что $(\alpha/\beta - \text{невычет}) \Leftrightarrow (\text{ровно один из } \{\alpha, \beta\} - \text{вычет})$. Теперь исследуем корни многочлена из решения $P(x) = (x + i)^{(p-1)/2} - 1$. Ими являются такие z , что $(z - i)$ – квадратичный вычет. Мы хотим оценить вероятность того, что «ровно один из $\{x_1, x_2\}$ является корнем $P(x)$ » \Leftrightarrow « $(i - x_1)/(i - x_2)$ – невычет». $\forall x_1 \neq x_2 \ i \xrightarrow{f} \frac{i-x_1}{i-x_2} \Rightarrow f$ – биекция. Невычетов $(p - 1)/2$. ■

Замечание 3.11.2. Если длина p равна n , то $E(\text{времени работы})$ при “умножении по модулю” за $\mathcal{O}(n \log n)$ получается $\mathcal{O}(n^2 \log n)$.

Замечание 3.11.3. Алгоритм можно обобщить на извлечение корня k -й степени.

Лекция #4: Модели вычислений и диагонализация

4 февраля

4.1. Машина Тьюринга

Зададим какой-то конечный алфавит Σ . Машина Тьюринга состоит из трёх лент, где одна лента состоит из бесконечной последовательности символов, проиндексированных натуральными числами. Первая лента для входных данных, вторая для выходных, а третья рабочая, на которую можно писать что угодно. Также у самой машины есть состояние q из конечного множества Q . Обычно это состояние кодирует «позицию в программе» или какое-то константное число бит памяти.

В каждый момент у машины Тьюринга есть три головки, по одной на каждую ленту. Головка указывает на конкретный символ. За один шаг машина Тьюринга читает все символы под головками и на основании этого принимает решение сдвигать ли головки, и какие символы стоит записать под их старым положением.

Формально, задано какое-то конечное множество состояний Q машины Тьюринга, а шаг вычисления описывается функцией:

$$\delta: Q \times \Sigma^3 \rightarrow Q \times \Sigma^3 \times \{Left, Stay, Right\}^3$$

Т.е. по состоянию и символам принимаем решение, что делать дальше.

Возможно это неинтуитивно, но в такой модели можно запрограммировать любую программу, которую можно было бы написать для «обычного» компьютера, причём получится разве что в полином хуже.

Это верно и для других разумных моделей, так что получается, что вне зависимости от рассматриваемой модели классы, P, NP выражают одинаковое множество алгоритмов/задач.

4.2. Недетерминизм и рандом

Как бы формализовать класс NP? Можно добавить ещё одну ленту для y . А можно определить недетерминированную машину тьюринга.

Она будет отличаться от обычной тем, что функций перехода будет две: δ_0 и δ_1 , и алгоритм как бы недетерминированно делает переход в лучшей из них (формально, в decision задаче, машина допускает слово x если есть хотя бы один способ выбрать переход каждый раз).

Несложно видеть, что это определение эквивалентно старому: если есть подсказка, то её можно интерпретировать как выбор по каким δ делать переходы. В другую сторону: в начале сделаем достаточно много шагов вычислений, которые просто записывают полученный недетерминизм в подсказку y .

Как бы формализовать рандомизированные вычисления? Да также. Хотя в случае рандома обычно говорят о вероятностной ленте. Вероятностная лента это бесконечная лента заполненная случайными битами. Каждый раз, когда нужно немного случайности, можно её просто читать вперёд.

4.3. RAM-машина

RAM-машина это более мощный и естественный способом формализовать процесс вычисления. RAM-машина состоит из бесконечного количества регистров r_0, r_1, r_2, \dots и кода, который она выполняет. Возможные инструкции выглядят следующим образом:

1. `mov ri rj`, копировать один регистр в другой
2. `mov ri const`,
3. `mov ri *rj`, разыменованье указателя, скопировать регистр с номером, лежащим в r_j .
4. `mov *ri rj`,
5. `add/mult/sub/div/mod ri rj`, арифметические операции
6. `jump 17`, перейти на другую строку программы
7. `halt`, завершить исполнение

Обычно считают, что арифметические операции выполняются за $\mathcal{O}(1)$, вне зависимости от битности чисел.

Однако в таком случае можно за полиномиальное время создать и работать с числами экспоненциальной длины, что как-то странно. Поэтому часто добавляют требование о том, что потребляемая память составляет не более, чем полином от времени работы.

4.4. Строгое и слабое полиномиальное время

Предположим, что входные данные задачи даны как последовательности целых чисел (`int`), а решаем мы задачу на RAM-машине.

Def 4.4.1. Алгоритм решения называется строго полиномиальным, если

1. Время работы можно ограничить полиномом от **количества** целых чисел во входе.
2. Алгоритм потребляет не более чем полиномиальное количество памяти (в битах), относительно входа.

Все остальные алгоритмы, работающие за полиномиальное время, называются *слабо полиномиальными*.

Пример: QuickSort работает за строго полиномиальное время, а алгоритм Евклида — нет.

Есть задачи для которых известно полиномиальное решение, но существование строго полиномиального алгоритма является нерешённой задачей.

4.5. Диагонализация: доказательство теоремы о временной иерархии

Напомним теорему:

Теорема 4.5.1. $DTime(f) \subsetneq DTime(f \log^2 f)$

Вложение очевидно, будем доказать именно неравенство. Доказывать будем для машин Тьюринга. Нам понадобится идея о том, что существуют программы, способные симулировать выполнение других программ.

Такая конструкция называется универсальной машиной тьюринга. По описанию другой машины M , её входных данных x и таймеру $t(|x|)$, она способна выполнить $t(|x|)$ шагов этой машины за время $\mathcal{O}(t(|x|) \log t(|x|))$.

И перед тем как приступить к доказательству сделаем ещё несколько замечаний.

1. Чтобы показать неравномощность нужно предъявить язык, который принадлежит правой части, но не левой. Скоро мы такой язык предъявим.
2. Любой язык можно определить его характеристической функцией, т.е.

$$L = \{x \mid f(x) = \text{true}\}$$
3. Доказывать утверждение мы будем в частном случае: $DTime(n) \subsetneq DTime(n^{1.5})$. Несложно будет увидеть, что общий случай доказывается аналогично.

Доказательство. Пронумеруем все возможные программы (машины Тьюринга) последовательными целыми числами (машину Тьюринга можно задать строкой, а множество строк счётно). Получилась последовательность машин $M_1, M_2, \dots, M_n, \dots$. Аналогично пронумеруем все возможные битовые строки, которые могут лежать в языке. Получилась последовательность $s_1, s_2, \dots, s_n, \dots$

Зададим язык характеристической функцией:

$$f(s_k) = \text{not } M_k(s_k) \text{ запущенная с будильником } t = |s_k|^{1.4}$$

Если машина в определении не успеет отработать, то скажем, что $f(s_k) = \text{false}$.

Заметим, что вычислить f мы можем за время $\mathcal{O}(n^{1.4} \log)$, а значит этот язык лежит в $DTime(n^{1.5})$. Покажем, что не существует машины, решающей его за время $\mathcal{O}(n)$ или $\leq Cn$ для какой-то C .

Основная идея: пусть машина M_k решает этот язык за время $\leq Cn$. Тогда $f(x) = M_k(x)$ для всех x . С другой стороны, $f(s_k) = \text{not } M_k(s_k)$, а значит пришли к противоречию.

К сожалению, это не совсем верно, в случае если M_k не успеет завершиться за время $t = |s_k|^{1.4}$ шагов. Т.е. если константа C достаточно велика.

Однако каждая машина встречается в списке машин бесконечное число раз (в любой код можно добавить незначащих комментариев или бессмысленных действий), т.е. $M_k = M_{i_1} = M_{i_2} = \dots$. И если пропустить достаточно много машин в этом списке, то асимптотика при $n^{1.4}$ таки задавит $\mathcal{O}(n)$, а значит аргумент выше приведёт к честному противоречию. ■

Такой трюк называется диагонализацией. Используя похожий аргумент, можно, например, **доказать**, что $|\mathbb{N}| \neq |2^{\mathbb{N}}|$.

Лекция #5: Кратчайшие пути

11 февраля

Def 5.0.1. 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.

5.1. 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	+
$w_e \geq 0$	$V^2 + E$	–	1956, Dijkstra	+
$w_e \geq 0$	$V \log V + E$	Dijkstra + fib-heap	1984, Fredman & Tarjan	+
$w_e \in \mathbb{N}$, неорграф	$V + E$	–	2000, Thorup	–
$w_e \geq 0$	$A^* \leq \text{Dijkstra}$	(A^*) A-star	1968, Nilsson & Raphael	+
любые веса	VE	–	1956, Bellman & Ford	+
$w_i \in \mathbb{Z} \cap [-N, \infty)$	$E\sqrt{V} \lceil \log(N+2) \rceil$	–	1994, Goldberg	+

Можно решать APSP запуском SSSP-решения от всех вершин. Время получится в V раз больше.

Кроме того, есть несколько алгоритмов специально для APSP:

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

К поиску в ширину и алгоритму Флойда применима оптимизация “битовое сжатие”:

$V + E \rightarrow \frac{V^2}{w}$, $V^3 \rightarrow \frac{V^3}{w}$, 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 $o(\text{размера графа})$.

Алгоритмы Гольдберга и Джонсона основаны на не известной нам пока идее потенциалов.

5.2. bfs

Ищем расстояния от s . Расстояние до вершины v обозначим $dist_v$. $A_d = \{v : dist_v = d\}$.

Алгоритм: индукция по d .

База: $d = 0, A_0 = \{s\}$.

Переход: мы уже знаем A_0, \dots, A_d , чтобы получить A_{d+1} переберём $N(A_d)$ – соседей A_d , возьмём те из них, что не лежат в $A_0 \cup \dots \cup A_d$: $A_{d+1} = N(A_d) \setminus (A_0 \cup \dots \cup A_d)$.

Чтобы за $\mathcal{O}(1)$ проверять, лежит ли вершина v в $A_0 \cup \dots \cup A_d$, используем $mark_v$.

```

1.  A0 = {s}
2.  mark ← 0, marks = 1
3.  for d = 0..|V|
4.      for v ∈ Ad
5.          for x ∈ neighbors(v)
6.              if markx = 0 then
7.                  markx = 1, Ad+1 ← x

```

Время работы $\mathcal{O}(V+E)$, так как в строке 4 каждую v мы переберём ровно один раз для связного графа. Соответственно 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 $q = A_0 A_1 A_2 \dots$, 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 $dist[x] = +\infty$.

```

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

```

Заметим, что q – очередь.

5.3. Модификации bfs

5.3.1. 1-k-bfs

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

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

Решение несколькими очередями

```

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

```

Этот же код верно работает и для весов $0-k$. Правда в таком случае мы можем вставлять новые вершины в список, по которому прямо сейчас итерируемся, что некорректно работает для некоторых контейнеров, таких как `vector<int>`.

Lm 5.3.1. Такой bfs работает за $\mathcal{O}(Vk + E)$

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

Lm 5.3.2. Каждая вершина добавляется не более $k + 1$ раза.

Доказательство. Пусть мы впервые добавили вершину v перейдя по ребру из u веса x .

Тогда $d_v = d_u + x$, а значит v лежит в списке d_v .

В какие ещё списки её могут добавить? Только в списки от d_u до $d_u + x$. ■

Lm 5.3.3. Если убрать из кода выше строчку 3, то он все ещё будет работать верно, разве что список смежности одной вершины может просматриваться $k + 1$ раз, а не 1. Получится время работы $\mathcal{O}(k(V + E))$.

5.3.2. 0-1-bfs

Возьмём обычный bfs с очередью, заменим очередь на дек. Напишем bfs как в коде, если перешли по ребру и ответ улучшился, добавим вершину в дек. При этом если прошли по ребру веса 1, то добавим в конец дека, иначе в начало.

Делает идейно то же самое, что и $0-k$ bfs в случае $k = 1$. А значит и время работы такое же: $\mathcal{O}(k(V + E)) = \mathcal{O}(V + E)$.

5.4. Дейкстра

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

Формально. Общее состояние алгоритма:

Мы уже нашли расстояния до вершин из множества A , $\forall x \notin A \ d_x = \min_{v \in A} (d_v + w_{vx})$.

Начало алгоритма: $A = \emptyset$, $d_s = 0$, $\forall v \neq s \ d_v = +\infty$.

Шаг алгоритма: возьмём $v = \operatorname{argmin}_{i \notin A} d_i$, 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 $D \geq d_a + w_{ab} \geq d_v \Rightarrow d_v$ – длина кратчайшего пути до v .

Время работы Дейкстры = $V \cdot \text{ExtractMin} + E \cdot \text{DecreaseKey}$.

(а) Реализация без куч даст время работы $\Theta(E + V^2)$.

(б) С бинарной кучей даст время работы $\mathcal{O}(E \log V)$. Обычно пишут именно так.

(в) С кучей Фибоначчи даст время работы $\mathcal{O}(E + V \log V)$.

(д) С деревом Ван-Эмбде-Боэса даст время работы $\mathcal{O}(E \log \log C)$ для целых весов из $[0, C)$.

(е) Существуют более крутые кучи, дающие время $\mathcal{O}(E + V \log \log V)$ на целочисленных весах.

Замечание 5.4.1. 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 $E \log V$.

5.5. A^* (А-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 Дейкстры, 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 .

Лм 5.5.1. Если f никогда не переоценивает расстояние ($\forall v: 0 \leq f_v \leq \text{dist}(v, t)$), то алгоритм A^* найдёт корректные расстояния. Правда при этом он может работать экспоненциально долго, так как будет доставать одну и ту же вершину из кучи много раз.

Теорема 5.5.2. Если функция f удовлетворяет неравенству треугольника: $\forall e: a \rightarrow b$ верно $f_a \leq f_b + w_{ab}$, то алгоритм A^* достанет каждую вершину из кучи не более одного раза.

Доказательство. Рассмотрим вершину a : $d_a + f_a = \min$. Возьмём $\forall v$ и путь из неё в a : $v \rightarrow u \rightarrow \dots \rightarrow c \rightarrow b \rightarrow a$. Нужно доказать, что $d_a + f_a \leq (d_v + w_{vu} + \dots + w_{cb} + w_{ba}) + f_a = F$.

Из неравенства Δ имеем $w_{ba} + f_a \geq f_b \wedge w_{cb} + f_b \geq f_a \wedge \dots \Rightarrow F \geq d_v + f_v \geq d_a + f_a$ ■

Следствие 5.5.3. Если для f верно неравенство треугольника, алгоритм можно остановить сразу после вынимания t из кучи.

Замечание 5.5.4. Заметим, что в доказательстве мы нигде не пользовались неотрицательностью весов. Для отрицательных обычно сложно найти функцию f с неравенством треугольника. Скоро мы введём технику потенциалов, узнав её, полезно ещё раз перечитать это место.

Замечание 5.5.5. “Оценка снизу” – лишь физический смысл функции f_v . Запускать алгоритм мы можем взяв любые вещественные числа.

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

Теорема 5.5.6. Алгоритм A^* на функции f , удовлетворяющий трём условиям: (а) неравенству треугольника, (b) $f_t = 0$, (c) $f_v \geq 0$ переберёт подмножество тех вершин, которые перебрала бы Дейкстра с **break**.

Доказательство. Дейкстра переберёт v : $d_v \leq d_t$.

A^* переберёт v : $d_v + f_v \leq d_t + f_t = d_t \Rightarrow d_v \leq d_t - f_v \leq d_t$. ■

5.6. Флойд

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

```

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

```

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

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

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

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

5.6.1. 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 $q[i, j] = q[i, k]$.

5.6.2. 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.

Лм 5.6.1. (\exists отрицательный цикл, проходящий через i) \Leftrightarrow (по окончании Флойда $d[i, i] < 0$).

Лм 5.6.2. (\nexists кратчайшего пути из a в b) \Leftrightarrow ($\exists i$ и пути “из a в i ” и “из i в b ”: $d[i, i] < 0$).

Замечание 5.6.3. О переполнениях. Веса всех кратчайших путей по модулю не больше $M = (V - 1)W$, где W – максимальный модуль веса ребра. Если в графе нет отрицательных циклов, Флойду хватает типа, вмещающего числа из $[-M, M]$. При наличии отрицательных циклов, могут получиться числа меньше $-M$, поэтому будем складывать с корректировкой:

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

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

Лекция #6: Кратчайшие пути

5 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 называемый “алгоритм Форд-Беллмана с очередью”, который работает за $\mathcal{O}(VE)$, а в среднем по тестам даже линейное время.

Теперь всё по порядку.

6.1. 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: $d[0, v] = (v == s ? 0 : +\infty)$.

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);

```

Оптимизируем 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”.

6.2. 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 6.2.1. \forall итерации \forall отрицательного цикла произойдёт релаксация хотя бы по одному ребру.

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

Осталось этот цикл восстановить. Пусть на n -й итерации произошла релаксация $a \rightarrow b$. 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 6.2.2. \forall момент времени \forall вершины v верно $d[p_v] + w[p_v, v] \leq d[v]$.

Доказательство. В момент сразу после релаксации 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 6.2.3. После релаксации v произошла релаксация $p_v \Rightarrow d[p_v] + w[p_v, v] < d[v]$.

Lm 6.2.4. 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 $s = v_1, v_2, \dots, v_k = b$. Последовательно для всех рёбер пути используем неравенство из леммы 6.2.2, получаем $d[s] + (\text{вес пути}) \leq d[b] \Leftrightarrow (\text{вес пути}) \leq d[b]$. Но $d[b]$ уже обновился на n -й итерации, значит строго меньше веса любого пути из не более чем $(n-1)$ ребра. Противоречие. ■

Lm 6.2.5. Вес полученного цикла отрицательный.

Доказательство. Сложим по всем рёбрам цикла неравенство из леммы 6.2.2, получим $\sum d_i + \sum w_i \leq \sum d_i \Leftrightarrow \sum w_i \leq 0$. Чтобы получить строгую отрицательность рассмотрим y – последнюю вершину цикла, для которой менялось расстояние. Тогда для вершины цикла $z: p_z = y$ неравенство из леммы 6.2.2 строгое. ■

Из доказанных лемм следует:

Теорема 6.2.6. Алгоритм восстановления отрицательного цикла корректен.

6.3. Модификации Форд-Беллмана

Теперь наш алгоритм умеет всё, что должен.

Осталось сделать так, чтобы он работал максимально быстро.

6.3.1. 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. Эмпирический факт: на случайных графах в среднем $\mathcal{O}(\log V)$ итераций \Rightarrow время работы $\mathcal{O}(E \log V)$.

6.3.2. Форд-Беллман с очередью

Сперва заметим, что внутри итерации бесполезно просматривать некоторые рёбра. Рёбро $a \rightarrow b$ 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.  B0 = {s}
3.  for (k = 0; Bk ≠ ∅; k++)
4.      for v ∈ Bk
5.          for x ∈ neighbors(v)
6.              if d[x] + w[v,x] < d[v] then
7.                  d[v] = d[x] + w[v,x], Bk+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} .

6.3.3. 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 6.3.1. 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 рёбра $a \rightarrow b$ расстояние до вершины a по индукции уже посчитано верно. ■

Lm 6.3.2. На любом тесте Форд-Беллман сделает итераций не больше, чем глубина дерева кратчайших путей.

Lm 6.3.3. \exists тест: Форд-Беллман в худшем случае сделает $V-1$ итерацию.

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

• Форд-Беллманом с random_shuffle #1

```

1 vector<int> d(n, INFINITY);
2 d[s] = 0;
3 random_shuffle(all_edges_in_graph); // перемешали рёбра только 1 раз в начале
4 while (расстояния улучшаются)
5     for (Edge e : all_edges_in_graph)
6         relax(d[e.end], d[e.start] + e.weight);

```

Lm 6.3.4. На любом тесте матожидание числа итераций не больше $\frac{V}{2}$.

Доказательство. Худший тест: дерево кратчайших путей – путь e_1, e_2, \dots, e_{V-1} ; до каждой вершины $\exists!$ кратчайший путь. Пусть $e_i: v_i \rightarrow v_{i+1}$ и расстояние до v_i найдено на k_i -й фазе. $Pr[k_i \neq k_{i+1}] = \frac{1}{2}$ (зависит только от порядка рёбер e_i и e_{i-1} в перестановке). Далее пользуемся линейностью матожидания. $\mathbb{E}[\#\{i: k_i \neq k_{i+1}\}] = \sum_i Pr[k_i \neq k_{i+1}] = \frac{V-1}{2}$. ■

• Форд-Беллманом с random_shuffle #2

Забавно, но весьма похожий код имеет уже другую оценку числа фаз.

```

1 vector<int> d(n, INFINITY);
2 d[s] = 0;
3 while (расстояния улучшаются)
4     random_shuffle(all_edges_in_graph); // переупорядочиваем рёбра каждый раз
5     for (Edge e : all_edges_in_graph)
6         relax(d[e.end], d[e.start] + e.weight);

```

Lm 6.3.5. На любом тесте матожидание числа итераций не больше $\frac{V}{e-1} \approx \frac{1}{1.72}$.

Доказательство. Пусть перед началом очередной итерации мы верно знаем расстояния до вершин из множества A . Рассмотрим $v \notin A$ и путь p_v из A до v в дереве кратчайших путей. Чтобы на текущей итерации найти расстояние до v , все рёбра p_v должны быть в перестановке в том же порядке, что и в p_v . Вероятность по всем $n!$ перестановкам этого события равна $\frac{1}{|p_v|!}$. Обозначим за k число вершин, до которых мы найдём расстояние.

Пользуемся линейностью матожидания $\mathbb{E}[k] = \sum_{v \notin A} \frac{1}{|p_v|!}$. Худший случай: «дерево кратчайших путей – бамбук», $\{|p_v|\} = \{1, 2, \dots\} \Rightarrow \mathbb{E}[k] = \sum_{i \geq 1} \frac{1}{i!} = e - 1$. ■

6.4. Потенциалы Джонсона

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

Def 6.4.1. Любой массив вещественных чисел p_v можно назвать потенциалами вершин.

При этом потенциалы задают новые веса рёбер: $e: a \rightarrow b, w'_e = w_e + p_a - p_b$.

Самое важное: любые потенциалы сохраняют кратчайшие пути.

Lm 6.4.2. $\forall s, t$ кратчайший путь на весах w_e перейдёт в кратчайший на весах w'_e

Доказательство. Рассмотрим любой путь $s \rightsquigarrow t: s = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = t$.

Его новый вес равен $w'_{v_1 v_2} + w'_{v_2 v_3} + \dots = (w_{v_1 v_2} + p_{v_1} - p_{v_2}) + (w_{v_2 v_3} + p_{v_2} - p_{v_3}) + \dots$

Заметим, что почти все p_v сокращаются, останется $W + p_s - p_t$, 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 $\forall e w'_e \geq 0$.

Для этого внимательно посмотрим на неравенство $w'_e = w_e + p_a - p_b \geq 0 \Leftrightarrow p_b \leq p_a + w_e$ 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 $\forall e w_e \geq 0$, получим $\forall v d_v = 0$, в любом случае $\forall v d_v \leq 0$.

Как найти расстояния? Форд-Беллманом.

Можно не добавлять s , а просто начать работу Форд-Беллмана с массива $\mathbf{d} = \{0, 0, \dots, 0\}$.

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-Беллмана 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.

6.5. Цикл минимального среднего веса

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

Искомый вес цикла обозначим μ . Веса рёбер w_e .

Lm 6.5.1. 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 6.5.2. 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 6.5.3. $R - L \leq \frac{1}{\sqrt{2}} \Rightarrow C$ – оптимален.

Доказательство. Рассмотрим любые два цикла, их средние веса $\frac{s_1}{k_1}$ и $\frac{s_2}{k_2}$. ($3 \leq k_1, k_2 \leq V$).

Предположим, что $\frac{s_1}{k_1} \neq \frac{s_2}{k_2}$ и оценим снизу их разность: $|\frac{s_1}{k_1} - \frac{s_2}{k_2}| = \frac{|s_1 k_2 - s_2 k_1|}{k_1 k_2} \geq \frac{1}{k_1 k_2} \geq \frac{1}{V^2} \Rightarrow$

При $R - L \leq \frac{1}{\sqrt{2}}$ на промежутке $[L, R]$ не может содержаться двух разных средних весов. ■

6.6. 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 6.6.1. 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 $(w_e - \mu - \frac{1}{\sqrt{2}})$: в графе с весами $(w_e - \mu)$ есть нулевой, мы ещё чуть уменьшили веса. Осталось доказать теорему ;-). Разобьём доказательство на несколько лемм.

Лм 6.6.2. Все w_e уменьшили на $x \Rightarrow Q$ уменьшится на x .

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

Лм 6.6.3. Если доказать теорему для случая $\mu = 0$, то она верна в общем случае

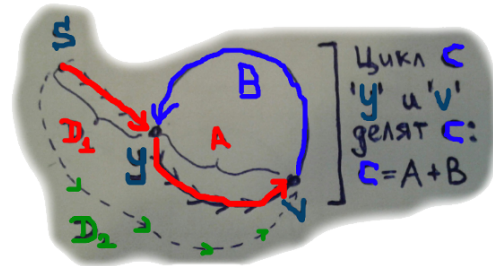
Доказательство. Уменьшив все w_e на μ по 6.5.2 перейдём к случаю $\mu' = 0$ По теореме получаем $Q' = 0$. Увеличим обратно веса на μ , по лемме 6.6.2 получаем $Q = Q' + \mu = \mu$. ■

Лм 6.6.4. $(\mu = 0) \Rightarrow (Q = 0)$

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 $d_{n,v} - \min_{k=1..n-1} d_{k,v} \geq 0$.

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 . ■



6.7. (*) 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.

6.7.1. 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 ≤ 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.

6.7.2. 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' ациклический, 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 ($d \leq 0$).

Лм 6.7.1. 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 $d_v \leq -\sqrt{k} \Rightarrow$ на пути до v хотя бы \sqrt{k} раз было ребро -1 , хотя бы столько интересных вершин. Иначе слоёв всего $\leq \sqrt{k}$, по принципу Дирихле хотя бы в одном $\geq \sqrt{k}$ 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 $s \rightsquigarrow B_d \rightsquigarrow x \rightarrow v$ 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, \dots

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 рёбра, исходящие из $A(v_1)$ веса 1. После изменения потенциала $A(v_1)$ их вес стал 0, поэтому dfs из v_2 должен по ним пройти. Чтобы быстро находить все такие рёбра, заведём структуру данных, в которую будем добавлять **все** пройденные рёбра.

Структура данных будет уметь делать три вещи: добавить ребро, уменьшить вес всех рёбер на 1, перебрать все рёбра веса 0 за их количество.

```

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

```

Хотим найти $A(v_i) \Rightarrow$ запускаем dfs от v_i и от всех концов рёбер, возвращённых `get()`.

После того, как нашли $A(v_i)$ вызываем `subtract()`.

Итого суммарное время поиска $A(v_1), A(v_2), A(v_3), \dots$ равно $\mathcal{O}(E)$.

6.7.3. Общий случай

Также, как мы от графа с весами рёбер ≥ -1 переходили к весам ≥ 0 , можно от весов рёбер $\geq -(2k+1)$ перейти к весам $\geq -k$.

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 #7: DSU, MST и Йен

25 февраля

7.1. 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 $0 \dots n-1$.

7.1.1. 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 7.1.1. Тогда суммарное число операций всех `for` во всех `join` не более $n \log n$.

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

Следствие 7.1.2. Суммарное время работы m `join`-ов $O(m + n \log n)$.

7.1.2. 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:

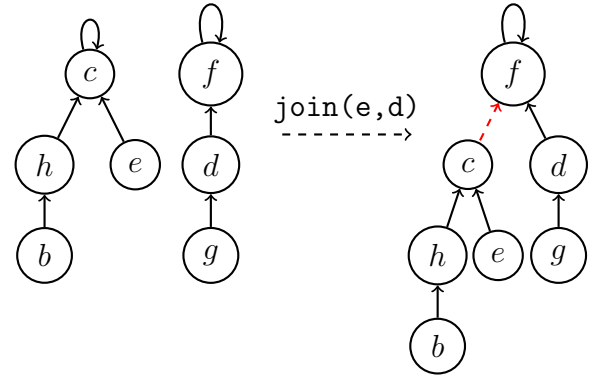
- **Ранговая сжатие путей:** все рёбра, по которым пройдёт `get` перенаправить в корень.
- **Ранговая эвристика:** подвешивать меньшее к большему. Можно выбирать меньшее по глубине, рангу, размеру. Эти идеи дают идентичный результат.

Def 7.1.3. Ранг вершины – глубина её поддеревя, 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:

```

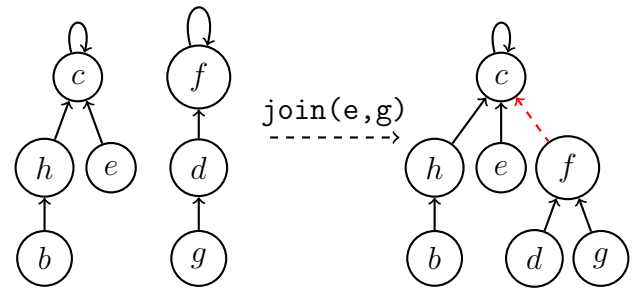
1 int p[n]; // для каждого элемента храним отца
2 void init( int n ) {
3     for (int i = 0; i < n; i++)
4         p[i] = i; // каждый элемент - сам себе корень
5 }
6 int get( int i ) {
7     // подняться до корня, вернуть корень
8     return p[i] == i ? i : get(p[i]);
9 }
10 void join( int a, int b ) {
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 (not (rank[a] <= rank[b])) swap(a, b);
12     if (rank[a] == rank[b]) rank[b]++;
13     p[a] = b; // a - вершина с меньшим рангом
14 }
    
```



Время работы `join` во всех версиях равно $\text{Time}(\text{get}) + 1$. Теперь оценим время работы `get`.

Теорема 7.1.4. 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 7.1.5. 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 7.1.6. Пусть v – вершина, p – её отец, $size$ – размер поддерева.

Ребро $p \rightarrow v$ называется тяжёлым если $size(v) > \frac{1}{2}size(p)$
лёгким если $size(v) \leq \frac{1}{2}size(p)$

Теорема 7.1.7. 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 ($v \rightarrow p$) 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. ■

7.1.3. Оценка $\mathcal{O}(\log^* n)$

Теорема 7.1.8. Сжатие путей вместе с ранговой эвристикой дадут следующую оценку: m запросов `get` в сумме пройдут не более $m(1 + 2 \log^* n) + \Theta(n)$ рёбер.

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

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

Теорема 7.1.9. Сжатие путей вместе с ранговой эвристики дадут следующую оценку: m запросов `get` в сумме пройдут не более $\Theta(m + n \log^* n)$ рёбер.

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

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

7.2.1. Интуиция и $\log^{**} n$

Теорема 7.2.1. Сжатие путей вместе с ранговой эвристикой дадут следующую оценку: m запросов `get` в сумме пройдут не более $\Theta(m(1 + \log^{**} n) + n)$ рёбер.

Доказательство. Ребро $(v \rightarrow p)$ крутизны хотя бы $\text{rank}[v]$ назовём *дважды крутым*.

Если обычное ребро $x^{\text{rank}[v]}$ раз поучаствует в сжатии, оно станет крутым \Rightarrow проходов по не крутым рёбрам $\sum_v x^{\text{rank}[v]} = \Theta(n)$.

Если крутое ребро $\text{rank}[v]$ раз поучаствует в сжатии, оно станет дважды крутым \Rightarrow проходов по не дважды крутым рёбрам $\sum_v \text{rank}[v] \leq 2n + \sum_v x^{\text{rank}[v]} = \Theta(n)$.

Осталось оценить число дважды крутых рёбер на пути. Для дважды крутого ребра имеем:

$$\text{rank}[p_v] \geq x^{x^{x^{\dots^{\text{rank}[v]}}}}$$

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

Замечание 7.2.2. Аналогично можно рассмотреть *трижды крутые* и даже k -раз крутые рёбра. Тогда суммарное время всех `get`-ов будет $\Theta((m + n)k + m \log^{**...} n)$.

7.2.2. Введение обратной функции Аккермана

Def 7.2.3. *Функция Аккермана*

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

Lm 7.2.4. $A_k(n) > n$

Lm 7.2.5. $f(t) = A_t(1)$ монотонно возрастает

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

Выпишем несколько первых функций явно:

$$A_0(t) = t + 1$$

$$A_1(t) = A_0^{(t+1)}(t) = 2t + 1$$

$$A_2(t) = A_1^{(t+1)}(t) = 2^{t+1}(t + 1) - 1 \geq 2^t \text{ (последнее равенство доказывает по индукции)}$$

$$A_3(t) = A_2^{(t+1)}(t) \geq A_2^t(2^t) \geq \dots \geq 2^{2^{\dots^t}} \text{ (высота башни } t + 1).$$

Def 7.2.6. *Обратная функция Аккермана* $\alpha(t) = \min\{k \mid A_k(1) \geq t\}$

Посчитаем несколько первых значений функции α

$$A_0(1) = 1 + 1 = 2$$

$$A_1(1) = 2 \cdot 1 + 1 = 3$$

$$A_2(1) = 2^2 \cdot 2 - 1 = 7$$

$$A_3(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2047$$

$$A_4(1) = A_3(A_3(1)) = A_3(2047) > 2^{2^{\dots^{2047}}} \text{ (башня высоты 2048).}$$

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

7.2.3. Доказательство

Для краткости записей ранг вершины e СНМ будем обозначать r_e , а родителя e в СНМ p_e .

У каждого множества в СНМ есть ровно один корень (представитель множества).

Элемент, который не является корнем, будем называть обычным.

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

Def 7.2.7. *Порядок элемента.* \forall обычного элемента e $level(e) = \max\{k \mid r_{p_e} \geq A_k(r_e)\}$.

Def 7.2.8. *Итерация порядка.* \forall обычного элемента e $iter(e) = \max\{i \mid r_{p_e} \geq A_{level(e)}^{(i)}(r_e)\}$.

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

Lm 7.2.9. \forall обычного элемента e верно, что $0 \leq level(e) < \alpha(n)$.

Доказательство. $level(e) \geq 0$, так как $A_0(r_e) = r_e + 1 \leq r_{p_e}$.

$n > r_{p_e} \geq A_{level(e)}(r_e) \geq A_{level(e)}(1) \Rightarrow n > A_{level(e)}(1) \Rightarrow level(e) < \alpha(n)$. ■

Lm 7.2.10. \forall обычного элемента e верно, что $1 \leq \text{iter}(e) \leq r_e$.

Доказательство. $r_{p_e} \geq A_{\text{level}(e)}(r_e) \Rightarrow \text{iter}(e) \geq 1$.

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

Будем доказывать время работы СНМ методом амортизационного анализа.

Def 7.2.11. *Потенциал элемента.* Определим для любого элемента e величину $\varphi(e)$.

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

Def 7.2.12. *Потенциал.* Возьмём $\varphi = \sum_e \varphi(e)$.

Обозначим через φ_t потенциал после первых t операций.

Lm 7.2.13. $\varphi_0 = 0, \forall i \varphi_i \geq 0$.

Доказательство. Изначально все элементы – представители, и все $r_e = 0$. Далее все $r_e \geq 0$, появляются обычные элементы. Используем 7.2.9 и 7.2.10, получаем неотрицательность потенциалов обычных элементов. ■

Lm 7.2.14. Если у обычной вершины v увеличить ранг отца, то $\varphi(v) \searrow$.

Теорема 7.2.15. Для операции типа *join* амортизированное время $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$.

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

Теорема 7.2.16. Для операции типа *get* амортизированное время $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$.

Доказательство. После сжатия путей по 7.2.14 потенциалы вершин не могли увеличиться. Пусть i -ая операция *get* проходит $w_i + x_i + 2$ вершин. Здесь w_i – число вершин $v: \Delta\varphi(v) < 0$, x_i – число, вершин $v: \Delta\varphi(v) = 0$, но $r_{p_v} \uparrow$, а 2 – корень и его сын. Если у вершины v не меняется потенциал, не меняются и $\text{level}(v), \text{iter}(v)$.

Осталось показать, что $x_i \leq \alpha(n)$. От противного: $(x_i > \alpha(n)) \wedge$ (по 7.2.9 \exists не более $\alpha(n)$ различных значений level) $\Rightarrow \exists$ две вершины a и b на пути: $\Delta\varphi(a) = \Delta\varphi(b) = 0, \text{level}(a) = \text{level}(b)$. После сжатия путей у нижней из $\{a, b\}$ как минимум изменится iter . Противоречие. ■

Теорема 7.2.17. Суммарное время работы m произвольных операций с СНМ – $\mathcal{O}(m \cdot \alpha(n))$.

Доказательство. $\sum_i t_i = \sum_i a_i - \varphi_m + \varphi_0$ ($\varphi_0 = 0, \varphi_m \geq 0$) $\Rightarrow \sum_i t_i \leq \sum_i a_i = m \cdot \alpha(n)$. ■

7.3. MST: Минимальное Остовное Дерево

Def 7.3.1. *Остовное дерево связного графа – подмножество его рёбер, являющееся деревом.*

Def 7.3.2. *Вес остова – сумма весов рёбер.*

Задача: дан граф, найти остов минимальной стоимости.

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

7.3.1. Алгоритм Краскала

Начнём с пустого остова. Будем перебирать рёбра в порядке возрастания веса.

Если текущее ребро при добавлении в остов не образует циклов, добавим ребро в остов.

Проверку на циклы будем делать СНМ-ом. Время работы алгоритма: $\mathcal{O}(\text{sort}(E) + (V+E)\alpha)$.

7.3.2. Алгоритм Прима

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

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

Замечание 7.3.3. Алгоритм Прима – Дейкстра с операцией “max” вместо “+”.

7.3.3. Алгоритм Борувки

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

Фаза алгоритма: для каждой вершины выберем минимальное по весу ребро.

Выбранные рёбра не образуют циклов (от противного: пользуемся тем, что веса различны).

Добавим все выбранные рёбра в остов. Сожмём компоненты связности. Удалим кратные рёбра.

Алгоритм: пока граф не сожмётся в одну вершину выполняем очередную фазу алгоритма.

Удалять кратные рёбра будем сортировкой рёбер.

Рёбра – пары чисел от 1 до $V \Rightarrow$ цифровая сортировка справится за $\mathcal{O}(V + E)$.

Lm 7.3.4. Время работы алгоритма Борувки $\mathcal{O}((V + E) \log V)$.

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

Lm 7.3.5. Время работы алгоритма Борувки $\mathcal{O}(E + V^2)$.

Доказательство. $\mathcal{O}(V + E)$ нужно на первое удаление кратных рёбер.

Теперь всегда $E \leq V^2 \Rightarrow$ алгоритм работает за $\mathcal{O}(V^2 + (\frac{V}{2})^2 + (\frac{V}{4})^2 + \dots) = \mathcal{O}(V^2)$. ■

Lm 7.3.6. Время работы алгоритма Борувки $\mathcal{O}(E \cdot \lceil \log V^2 / E \rceil)$.

Доказательство. V^2 за фазу уменьшается в 4 раза \Rightarrow за $\lceil \log_4 V^2 / E \rceil$ фаз V^2 станет не больше исходного E . По предыдущей лемме оставшаяся часть алгоритма отработает за $\mathcal{O}(E)$. ■

7.3.4. Сравнение алгоритмов

Асимптотика: из изученных нами куч и сортировок получаем, что лучше всего

Прим за $\mathcal{O}(E + V \log V)$, Прим за $\mathcal{O}(E \log_{E/V} V)$ и Краскал за $\mathcal{O}(E \log \log C)$.

- Краскал просто пишется, быстро работает.
- На очень плотных графах быстрее V^2 через Прима.
- Алгоритм Борувки мы оценили лишь сверху. В среднем по тестам он даёт линию.

Замечание 7.3.7. На всех планарных графах Борувка работает за $\mathcal{O}(V + E)$.

7.3.5. Лемма о разрезе и доказательства

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

Лм 7.3.8. Для любого разбиения множества вершин $V = A \sqcup B$, существует минимальный остов, содержащий e – минимальное по весу ребро, проходящее через разрез $\langle A, B \rangle$.

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

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

Следствие 7.3.9. X – подмножество рёбер некоего минимального остова. Зафиксируем разрез $V = A \sqcup B$ такой, что все рёбра X не пересекают разрез. Тогда \exists минимальный остов, содержащий $X \cup \{e\}$, где e – минимальное по весу ребро, проходящее через разрез $\langle A, B \rangle$.

Для доказательства описанных выше алгоритмов нужно лишь применить правильный разрез.

• Доказательство алгоритма Прима.

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

• Доказательство алгоритма Краскала.

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

• Доказательство алгоритма Борувки.

Рёбра, которые хочет добавить Борувка, разобьём на компоненты связности. Каждая компонента – дерево, рёбра компоненты будем добавлять по одному от корня к листьям. Добавляя очередное ребро ведущее в лист v , используем разрез $\langle \{v\}, V \setminus \{v\} \rangle$ и лемму о разрезе. ■

Заметим, что лемма о разрезе говорит лишь про возможность добавить к остову одно ребро \Rightarrow наше рассуждение нельзя укоротить, оставив только последнюю фразу.

7.4. Алгоритм Йена

Мы научились искать кратчайший простой путь, MST.

Идея Йена помогает искать ещё и k -й минимальный (простой путь, остовное дерево, поток...)

Оригинальный алгоритм ищет именно k -й путь.

Алгоритм. Запустим Дейкстру, получим кратчайший путь p_1 .

Второй простой путь p_2 отличается от p_1 , переберём, совпадающий префикс путей, запретим ходить по следующему ребру, запустим Дейкстру. Итого p_2 мы нашли за $|p_1|$ запусков Дейкстры.

Чтобы найти p_3 , заметим, что при поиске p_2 у нас было несколько кандидатов вида $\langle P, E, W \rangle$ – множество путей, у которых начало P , после P запрещено ходить по рёбрам из E (пока $|E| = 1$, в дальнейшем будет больше), Дейкстра из множества $\langle P, E \rangle$ нашла кратчайший, его вес – W .

p_2 – минимум по W_i , а также минимум в своём множестве путей $\langle P_i, E_i \rangle$. Остатки в $\langle P_i, E_i \rangle$ пути разделятся на множества такого же вида – пройдем P_i , зафиксируем ещё несколько (возможно, 0) вершин по p_2 и свернем (запретим следующее ребро из p_2).

Нужно хранить кучу k минимальных по W_i множеств $\langle P_i, E_i \rangle$ и k раз “выбирать следующий минимальный путь”: вытаскивать из кучи минимум, затем $\leq V$ раз запускать Дейкстру.

Итого: 2-й путь нашли за $\mathcal{O}(V \cdot \text{Dijkstra})$, а k -й за $\mathcal{O}(kV \cdot \text{Dijkstra})$.

Лекция #8: Жадность и приближённые алгоритмы

17 марта

The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind. And greed-mark my words-will save not only Teldar Paper but the other malfunctioning corporation called the USA.

Gordon Gekko [Michael Douglas], *Wall Street (1987)*

There is always an easy solution to every human problem-neat, plausible, and wrong.

H. L. Mencken, *"The Divine Afflatus", New York Evening Mail (November 16, 1917)*

8.1. 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 8.1.1. Алгоритм M называется α -оптимальным (α -OPT) для задачи P , если на любом входе для P ответ, выдаваемый M , не более, чем в α раз хуже оптимального.

8.2. Коммивояжёр

Задача коммивояжёра заключается в поиске гамильтонова цикла минимального веса. Решим задачу в случае полного графа с неравенством Δ .

Lm 8.2.1. $\forall \alpha > 1$ \nexists полиномиального α -OPT алгоритма для задачи коммивояжёра.

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

Замечание 8.2.2. Любой граф можно сделать полным, добавив вместо отсутствия ребра ребро веса $+\infty$. А вот неравенство Δ – важное свойство.

8.2.1. 2-OPT через MST

Оптимальный маршрут коммивояжёра = цикл = путь + ребро = остовное дерево + ребро \Rightarrow **MST \leq OPT**. Возьмём MST графа. Каждое ребро заменим на два ориентированных (туда и обратно), получили эйлеров орграф (число входящих равно числу исходящих).

Возьмём эйлеров обход, его вес в два раза больше MST. Если в полученном пути некоторая вершина встречается второй раз, удалим её второе вхождение.

По нер-ву Δ вес пути только уменьшится. Получили маршрут коммивояжёра веса $\leq 2 \cdot$ MST.

8.2.2. 1.5-OPT через MST (Кристофидес)

Чтобы превратить MST в эйлеров граф, добавим к нему M – совершенное паросочетание минимального веса на вершинах нечётной степени. $w(MST) \leq$ OPT, докажем что $w(M) \leq \frac{1}{2}$ OPT.

$\upharpoonright X$ – маршрут коммивояжёра только на нечётных вершинах. По нер-ву Δ $w(X) \leq$ OPT.

Цикл X можно разбить на два паросочетания – A (чётные рёбра) и B (нечётные рёбра).

$$w(A) + w(B) = w(X) \Rightarrow \min(w(A), w(B)) \leq \frac{1}{2}w(X) \leq \frac{1}{2}\text{OPT}$$

Lm 8.2.3. \exists полиномиальный алгоритм для поиска паросочетания минимального веса в произвольном графе. Кстати, умеют это делать весьма быстро, за $\mathcal{O}(E\sqrt{V} \log(VN))$. PDF 2017.

8.3. Хаффман

Для начала заметим печальный факт: идеального архиватора не существует.

Lm 8.3.1. \nexists архиватора f и целого $n: \forall x (|x| = n \Rightarrow f(x) < n)$

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

Алгоритм Хаффман – наиболее известный алгоритм сжатия текста.

Хотим придумать префиксные коды, минимизирующие величину

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

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

• Алгоритм построения префиксных кодов

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

• Реализация

Если для извлечения минимума использовать кучу, мы получим время $\mathcal{O}(\Sigma \log \Sigma)$, где Σ – размер алфавита.

Можно изначально отсортировать символы по частоте и заметить, что новые символы по частоте только возрастают, поэтому для извлечения минимума достаточно поддерживать две очереди – “исходные символы” и “новые символы”. Минимум всегда содержится в начале одной из двух очередей $\Rightarrow \mathcal{O}(\text{sort}(\Sigma) + \Sigma)$.

Теорема 8.3.2. Алгоритм корректен.

Доказательство. Если упорядочить символы $len_i \searrow_{\Sigma}$, 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.

Рассмотрим самые глубокие листы в ответе. Таких хотя бы 2, т.к. у каждого глубокого листа есть брат. Пусть таких листов k . Тогда им должны соответствовать k самых редких символов, иначе можно улучшить через своп.

Переупорядочим символы по кодам этих листов так, чтобы выбранные алгоритмом символы были рядом. Сделаем индукционный переход. ■

8.3.1. 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 = $\text{sizeof}(\text{Node}) * (2\Sigma - 1)$ 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.

8.4. Компаратор и сортировки

Решим несколько простых задач.

1. Сортировка по времени

Даны n заданий, у каждого есть время выполнения t_i . К моменту T выполнить как можно больше заданий. Решение: выполняем в порядке возрастания t_i . Доказательство: не важен порядок \Rightarrow выполняем в порядке возрастания t_i . Пусть на первом месте не t_{min} , поставим $t_{min} \Rightarrow$ сумма уменьшилась \Rightarrow сумма всё ещё $\leq T \Rightarrow$ перейдём по индукции к аналогичной задаче, но без задания t_{min} и временем $T - t_{min}$.

2. Сортировка по дедлайну

Даны n заданий, у каждого есть время выполнения t_i и дедлайн момента сдачи d_i . Нужно успеть выполнить все задания. Решение: выполнять в порядке возрастания дедлайна. Способ придумать решение: посмотрим, кого мы можем выполнить последним? Любого, у кого $d_i \geq \sum_j t_j \Rightarrow$ выполним в конце d_{max} . Доказательство: можем выполнить множество заданий I , выполнив последним $d_{last} \Rightarrow$ (можем выполнить $I_1 = I \setminus \{d_{max}\} \wedge (d_{max} \geq d_{last} \geq \sum_j t_j \Rightarrow$ можем выполнить последним $d_{max}) \Rightarrow$ возьмём по индукции решение для I_1 , допишем к нему d_{max} . На практике разобраны похожие задачи – с решением “сортировка по сумме”.

Ход мыслей в (1) и (2): как только мы показали, кого ставить на первое/последнее место, отсортируем в таком порядке, получим решение за $\mathcal{O}(\text{sort})$.

3. Сортировка по частному

Даны файлы размеров s_i и частоты обращения к ним cnt_i . Лента – хранилище данных, позволяющее к файлу, начинающемуся на позиции pos обратиться за время pos .

Расположить файлы на ленте, минимизировать суммарное время обращения $\sum_i pos(i)cnt_i$.

Решение для случая $s_i = 1$: сортируем по убыванию cnt_i .

Решение для случая $cnt_i = 1$: сортируем по возрастанию s_i .

Общий случай: сортируем по убыванию $\frac{cnt_i}{s_i}$. Чтобы придумать такое решение предположим, что задача решается сортировкой \Rightarrow у сортировки есть компаратор и компаратор решает задачу для $n = 2 \Rightarrow$ чтобы придумать компаратор полезно исследовать решение при $n = 2$. В порядке $\{1, 2\}$ получаем cnt_2s_1 , в порядке $\{2, 1\}$ получаем $cnt_1s_2 \Rightarrow less(1, 2): cnt_2s_1 < cnt_1s_2 \Leftrightarrow \frac{cnt_1}{s_1} > \frac{cnt_2}{s_2}$.

Решение придумано, доказываем: для оптимального ответа верно, что отличимые компаратором $less$ элементы идут в том же порядке, что задаёт $less$; осталось понять, что неотличимые ($\frac{cnt_1}{s_1} = \frac{cnt_2}{s_2}$) можно расставить в любом порядке, ответ $\sum_i pos(i)cnt_i$ не изменится.

Алгоритм 8.4.1. Перестановочный метод: π – искомая перестановка, $F(\pi) \rightarrow \min$.

В поисках оптимальной π пробуем поменять соседние элемента: $F(\pi) \leq F(\pi[\text{swap}(i, i+1)])$.

Полученное условие используем, как компаратор.

Часто это условие совпадает с более простым: $\text{less}(i, j) = (F([i, j]) < F([j, i]))$

Необходимые условия к компаратору

Когда можно пользоваться перестановочным методом 8.4.1 из (3)?

Чтобы передать в сортировку полученный компаратор, он должен задавать **строгий линейный порядок**, т.е. в частности должно быть антирефлексивным, антисимметричным, транзитивным и позволять сравнить любые два элемента. Проблемы обычно только с транзитивностью. Чтобы показать транзитивность в (3), мы переписывали компаратор: $(cnt_2 s_1 < cnt_1 s_2) \rightarrow (\frac{cnt_1}{s_1} > \frac{cnt_2}{s_2})$.

Под условие выше подходит компаратор, считающий все элементы равными “less(i, j): return 0” \Rightarrow нужно наложить доп.условие на равные элементы: потребуем, чтобы \forall порядка π , $\forall i \neg less(\pi_i, \pi_{i+1}) \Rightarrow F(\pi) = F(\pi[swap(i, i+1)])$, где F – функция, которую мы оптимизируем.

8.4.1. Задача про 2 станка

Задача: даны 2 станка, n деталей, время обработки деталей на первом станке a_i и время обработки деталей на втором станке b_i . Выбрать порядок обработки деталей π , обработать каждую сперва на первом, затем на втором станке. Минимизировать время окончания работ.

Решение: разбить на $P_1 = \{i: a_i \leq b_i\}$ и $P_2 = \{i: a_i > b_i\}$, отсортировать P_1 по возрастанию a , P_2 по убыванию b . $\pi = P_1 P_2$. Оригинальная **статья** (S.M.Johnson'1953).

• Вспомогательные рассуждения

Время окончания выполнения на первом станке $A_i = A_{i-1} + a_{\pi_i}$

Время окончания выполнения на втором станке $B_i = \max(A_i, B_{i-1}) + b_{\pi_i}$

Пусть x_i – ожидание второго станка перед обработкой i -й детали $\Rightarrow B_k = \sum_{i \leq k} (x_i + b_i)$.

Обозначим $F(k) = \sum_{i \leq k} x_i$. $F(n)$ – суммарный простой второго станка. Задача: $F(n) \rightarrow \min$.

При этом $x_k = \max(0, A_k - B_{k-1}) = \left(\sum_{i=1}^k a_i - \sum_{i=1}^{k-1} b_i \right) - \sum_{i=1}^{k-1} x_i = D_k - F_{k-1}$, где $D_i = \sum_{j \leq i} a_j - \sum_{j < i} b_j$.

Докажем по индукции, что $F(k) = \max_{i=1..k}(0, \max D_i)$.

База: $k = 0$. Переход: $x_k > 0 \Rightarrow x_k = D_k - F_{k-1} \wedge F_k = D_k$; $x_k = 0 \Rightarrow F(k) = F(k-1) \wedge D_k \leq F_{k-1}$.

• Корректность алгоритма

Осталось показать, что для минимизации $F(n) = \max_{i=1..n} D_i$, нужно 3 вещи:

1. Если рядом стоят $a_i > b_i$ и $a_{i+1} \leq b_{i+1}$, можно поменять местами, $F(n) \searrow$
2. Если рядом стоят $a_i \leq b_i$ и $a_{i+1} \leq b_{i+1}$: $a_i > a_{i+1}$, можно поменять местами, $F(n) \searrow$
3. Если рядом стоят $a_i > b_i$ и $a_{i+1} > b_{i+1}$: $b_i < b_{i+1}$, можно поменять местами, $F(n) \searrow$

• Неверное решение

Выведем компаратор, как в 8.4.1. $F(1, 2) = a_1 + \max(a_2, b_1) + b_2 = (a_1 + a_2 + b_1 + b_2) - \min(a_1, b_2)$.

Получили “less(i, j): return min(a[i], b[j]) > min(a[j], b[i])”.

Получили транзитивное отношение меньше, которое не является строгим линейным порядком.

Если его передать функции `std::sort`, получится почти всегда рабочий алгоритм.

Стресс-тестом можно найти контрпримеры. Подробнее на [codeforces](#) и на [github](#).

• Внешние ссылки

Можно ознакомиться с [e-maxx](#) и [ИТМО-конспектами](#).

Оба рассуждения (доказательства) не полны. При этом по обоим ссылкам рабочая реализация (хоть и переусложнённая).

8.4.2. Выбор максимума

На примере задачи «**2. Сортировка по дедлайну**».

Полученный в этом метод решает многие задачи типа «нам известен оптимальный порядок для задачи *взять всех*».

Новая задача: нам нужно успеть сделать не все, но максимальное число заданий.

Решение за $\mathcal{O}(n^2)$: если мы зафиксируем множество заданий, которое хотим выполнить, выполнять нужно в порядке $d_i \uparrow \Rightarrow$ отсортируем по d_i и напишем динамику $T[i, k]$ – минимальное суммарное время, нужное, чтобы выполнить какие-то k заданий из i первых. Переходы:

1. `relax(T[i+1, k], T[i, k]);`
2. `if d[i] >= T[i, k] + t[i] then relax(T[i+1, k+1], T[i, k] + t[i]);`

Решение за $\mathcal{O}(n \log n)$: будем жадно брать в порядке $d_i \uparrow$, если не получается взять, попытаемся заместиться на работу с максимальным t_j .

1. `sumTime = 0`
2. `for` пары $\langle d_i, t_i \rangle$ в порядке $d_i \uparrow$:
3. `if` `sumTime + t_i` $\leq d_i$ `then берём` (кладём t_i в кучу, меняем `sumTime`)
4. `else if` $t_i < \max t_j$ `then делаем замену` (меняем кучу и `sumTime`)

В строке 4, когда условие выполнено, верно и $d_i \geq d_j \Rightarrow$ мы лучше и по t , и по d ; замену точно сделать получится: вставим себя ровно на место удалённого.

• Корректность решения за $\mathcal{O}(n \log n)$

Пусть на момент рассмотрения пары $\langle d_i, t_i \rangle$ жадность считает, что нужно выполнять задания $t_{j_1} \leq t_{j_2} \leq \dots \leq t_{j_m}$. Мы специально записали эти задания в порядке $t_j \uparrow$. Докажем по индукции, что, $\forall k$ $T[i, k] = t_{j_1} + \dots + t_{j_k}$, более того множество этих k предметов есть $\{j_1, \dots, j_k\}$.

База: $i = 0 \Rightarrow m = 0$.

Переход: пользуемся корректностью динамики $T[i+1, k] = T[i, k] \vee T[i, k-1] + t_i$.

Заметим, что для всех k : $t_{j_k} \leq t_i$ первое оптимально и наше решение как раз его выберет.

А теперь заметим, что для всех k : $t_{j_k} > t_i$ второе возможно и оптимальнее первого.

Наше решение его и выберет. Для удобства считаем $t_{j_{m+1}} = +\infty$.

В сумме получаем, что $\forall k$ $T[i, k+1]$ – префикс $t_{j_1} \leq t_{j_2} \leq \dots \leq t_i \leq \dots \leq t_{j_m}$.

Лекция #9: Приближённые алгоритмы

10 марта

Книга по приближённым алгоритмам [Approximation algorithms, Vazirani'2001](#).

9.1. Рюкзаки

Будем пользоваться англоязычными обозначениями задач:

1. **Partition** – разбиение множества предметов на два равных по весу.
Даны a_i , выбрать $I: \sum_{i \in I} a_i = \frac{1}{2} \sum a_i$.
2. **Knapsack** – поместить в рюкзак размера S предметы максимальной суммарной стоимости.
Даны $a_i > 0, p_i > 0$ и S , выбрать $0 \leq x_i \leq 1: \sum a_i x_i \leq S$, при этом $\sum p_i x_i \rightarrow \max$.
 $x_i \in \mathbb{R} \Rightarrow$ задача называется непрерывным рюкзаком, решается жадной сортировкой по $\frac{p_i}{a_i}$.
 $x_i \in \mathbb{Z} \Rightarrow$ получили классический дискретный рюкзак (knapsack).
3. **Bin packing** – уложить предметы в минимальное число рюкзаков размера 1.
Даны $0 < a_i \leq 1$, выбрать $I_1, I_2, \dots, I_k: \cup I_j = \{1..n\} \wedge \forall j \sum_{i \in I_j} a_i \leq 1$. При этом $k \rightarrow \min$.

• NP-трудность

Базовой NP-трудной у нас будет задача partition.

Knapsack труден, так как можно взять все $p_i = a_i, S = \frac{1}{2} \sum a_i$ и решить partition.

Bin packing труден, так как если научимся отличать ответы 2 и 3, решим partition.

Следствие 9.1.1. $\forall \varepsilon > 0$ для bin packing $\nexists (1.5 - \varepsilon)$ -ОПТ алгоритма.

9.2. Схемы приближений

Для некоторых трудных задач доступны сколь угодно точные приближения за полиномиальное время. **PTAS/FPTAS**.

Def 9.2.1. Алгоритм $A(\varepsilon, x)$ – PTAS (polynomial-time approximation scheme) для задачи M , если $\forall \varepsilon > 0$ он отработает за полином от $|x|$ и выдаст решение F такое, что:

$$\begin{cases} F \geq (1 - \varepsilon)OPT & \text{если } M \text{ – задача максимизации} \\ F \leq (1 + \varepsilon)OPT & \text{если } M \text{ – задача минимизации} \end{cases}$$

Def 9.2.2. FPTAS (fully PTAS) обязывает A работать за полином и от $|x|$, и от ε^{-1} .

9.2.1. Partition

Задача NP-трудна. Эквивалентная ей – **subset sum**.

Оптимизационная версия partition: разбить на два максимально близких по сумме множества.

Обе задачи мы умеем решать за $\mathcal{O}^*(2^{n/2})$ методом meet-in-the-middle.

\nexists приближённых по величине $D = |\sum a_i - \sum b_i|$ решений (NP-трудно отличить $D = 0$ от $D > 0$).

Но делать что-то надо, поэтому строят приближённые решения по величине $\max(\sum a_i, \sum b_i)$.

Тривиальное жадное решение за $\mathcal{O}(n \log n)$: перебирать предметы в порядке $a_i \downarrow$, класть очередной предмет в множество с меньшей суммой. При $n \leq 4$ решение даст оптимальный ответ. Полученная разность D или оптимальна, или не более a_5 . Более того, это $\frac{7}{6}$ -ОПТ решение.

PTAS схема: переберём, куда кладем $2k$ максимальных по весу предметов, остаток разложим, следуя предыдущей жадности. Получили $(1 + \frac{1}{k})$ -ОПТ решение. Время $\mathcal{O}(2^k n) \Rightarrow$ не FPTAS.

Упражнение 9.2.3. Доказательство осталось на практику

Алгоритм Кармаркар-Карпа (1982) решает эту задачу лучше нашей базовой жадности:

```

1 solve(a):
2   while size(a) > 1:
3     x = a.extractMax()
4     y = a.extractMax()
5     a.push(x - y)
6   return a.extractMax()

```

Описанный алгоритм ещё называют LDM (largest difference method).

На n случайных равномерно распределённых величинах в $\mathbb{R} \cap [0, 1]$ жадность даст $E(D) = \Theta(\frac{1}{n})$, Кармаркар-Карп $n^{-\Theta(\log n)}$. Ту же асимптотику матожидания разности Кармаркар-Карп даст на любых независимых одинаково распределённых случайных величинах.

Доказательство (Yakir'1996). Более глубокий анализ: (Mertens'1999), (Boettcher, Mertens'2008).

9.2.2. Knapsack

Здесь можно почитать про PTAS и FPTAS для **knapsack**.

Задача: даны $a_i > 0, p_i > 0$, выбрать $0 \leq x_i \leq 1$: $\sum a_i x_i \leq S \wedge \sum p_i x_i \rightarrow \max$.

Если $x_i \in \mathbb{R}$ задача называется непрерывной и решается жадно сортировкой по $\frac{p_i}{a_i}$.

Если $x_i \in \mathbb{Z}$ задача называется дискретной или 0-1 и жадное решение ничего не приближает.

Пример: $a_1 = 1, p_1 = 2, a_2 = S, p_2 = S \Rightarrow$ жадность возьмёт первый предмет, оптимально брать второй. Ошибка в $S/2$ раз \Rightarrow не ограничена.

Тем не менее рюкзак легко приблизить и PTAS, и FPTAS.

- **PTAS**: переберём, какие k предметов возьмём за $\binom{n}{k}$, оставшиеся наберём жадностью.

Теорема 9.2.4. Мы получили $(1 - \frac{1}{k})$ -ОПТ решение, работающее за $\mathcal{O}(n^{k+1})$

Доказательство. Обозначим оптимальное решение I . Рассмотрим момент, когда мы угадали k max по p_i предметов, присутствующих в I : $i_1 \dots i_k$. Посмотрим теперь, как жадность лихо добавляет лучшие по $\frac{p_i}{a_i}$ предметы и остановимся на первом событии «жадность не взяла предмет $j \in I$ ». В этот момент остановим жадность, полученное решение обозначим I' . $I' \cup \{j\}$ не хуже ОПТ: i_1, \dots, i_k, j – общая часть, остальные предметы I' мажорируют I по $\frac{p_i}{a_i}$. Осталось заметить, что $p_j \leq \frac{1}{k}(p_{i_1} + \dots + p_{i_k}) = \frac{1}{k}\text{ОПТ} \Rightarrow P(I') \geq (1 - \frac{1}{k})P(I)$, где $P(I) = \sum_{i \in I} p_i$. ■

- **FPTAS**: возьмём псевдополиномиальную динамику `minWeight[i, sumP]`, и чтобы она работала за строгий полином, сделаем все $p_i \in \{0, 1, \dots, k\}$: $p'_i = \lfloor \frac{p_i}{\max_j p_j} k \rfloor = \lfloor \frac{1}{x} p_i \rfloor$, где $x = \frac{\max_j p_j}{k}$. Решение работает за $\mathcal{O}(n^2 k)$, оценим ошибку.

$$\text{Пусть } P(\text{set}) = \sum_{i \in \text{set}} p_i, \text{ а } P'(\text{set}) = \sum_{i \in \text{set}} p'_i = \sum_{i \in \text{set}} \lfloor \frac{p_i}{x} \rfloor.$$

Сразу заметим, что $P'(\text{set}) \leq \frac{P(\text{set})}{x} \leq P'(\text{set}) + n$, так как мы теряем на округлении, но не более, чем единицу на каждый предмет.

Пусть opt это оптимальное решение, а мы нашли sol . Тогда $P'(\text{sol}) \geq P'(\text{opt})$, так как мы нашли самое лучшее решение в профитах p' .

$$\begin{aligned}
 P(\text{sol}) &\geq x P'(\text{sol}), P'(\text{sol}) \geq P'(\text{opt}), P'(\text{opt}) \geq \frac{P(\text{opt}) - nx}{x} \\
 \implies P(\text{sol}) &\geq x P'(\text{sol}) \geq x P'(\text{opt}) \geq P(\text{opt}) - nx
 \end{aligned}$$

Получили решение, которое хуже не более чем на nx . Хотим: $nx \leq \varepsilon P(\text{opt})$.

Знаем, что $P(\text{opt}) \geq \max_j p_j$. Выберем x , так что $\varepsilon(\max_j p_j) \geq nx$, тогда получаем требуемое.

$$x \leq \frac{\varepsilon(\max_j p_j)}{n} \implies k \geq \frac{n}{\varepsilon}$$

9.2.3. Bin packing

Возможно, интересные ссылки:

- [PTAS](#) (более-менее случайная презентация)
- [Hoberg'2015](#) про $OPT + \mathcal{O}(\log OPT \log \log OPT)$ приближение.
- [Korf'1999](#) про решение задачи на практике.
- [Janson et al.'2010](#) про FPT-решение за $\mathcal{O}(2^{k \log^2 k} + n)$, ошибающееся \leq чем на 1 корзину.

Распределять по k корзинам сложнее, чем по двум (partition). Динамикой мы умеем решать задачу лишь для полиномиально ограниченных весов и то за $\mathcal{O}(n^{\mathcal{O}(k)})$.

• First fit

Перебираем предметы в произвольном порядке.

Для каждого предмета перебираем корзины, кладём предмет в первую подходящую.

Теорема 9.2.5. First fit – 2-ОПТ алгоритм для bin packing

Доказательство. Если наш ответ m , то $m - 1$ корзина заполнена больше чем на половину. Итого $\sum a_i > \frac{1}{2}(m - 1)$. С другой стороны $OPT \geq \sum a_i \Rightarrow 2OPT > m - 1 \Rightarrow 2OPT \geq m$. ■

Конечно, есть более удачные порядки перебора предметов чем “произвольный”.

И в теории, и на практике хорошо работает порядок “по убыванию”.

Теорема 9.2.6. First fit decreasing (FFD) для задачи bin packing даёт ответ $\leq \frac{11}{9}OPT + 1$

Доказательство. [Yue Minyi'1991](#). Чуть более сильный $(\frac{11}{9}OPT + \frac{6}{9})$ результат [Dosa'2007](#). ■

При этом выше (в 9.1.1) мы уже говорили, что нет $(1.5 - \varepsilon)$ -ОПТ алгоритма.

• Best fit

Эвристика: класть в корзину, в которой после добавления предмета останется поменьше места.

• Практически эффективное решение

Обе жадности уже хороши тем, что работают за $\mathcal{O}(n \log n)$.

Есть чуть менее быстрое, но более эффективное решение:

- зафиксируем число корзин k (если задача минимизации k , сделаем бинпоиск по ответу);
- начинём со случайного распределения предметов по k корзинам
- минимизируем максимальный размер корзины локальными оптимизациями двух типов – пытаемся перекинуть предмет из одной корзины в другую; пытаемся поменять два предмета в двух корзинах.
- для эффективной реализации второй оптимизации перебираем предметы в порядке $a_i \uparrow$ и пытаемся менять местами только соседние – a_i и a_{i-1} .

```

1 // p[i] - номер корзины по предмету, sumA[j] - суммарный вес в корзине
2 while optimizing // одна фаза оптимизации работает за O(n)
3   min_index <-- номер корзины с минимальным весом
4   for i = 1..n

```

```

5   if sumA[p[i]] - a[i] > sumA[min_index]
6       p[i] = min_index, recalc sumA
7   for i = 2..n // a[i] возрастают
8       if sumA[p[i]] - a[i] > sum[p[i-1]] - a[i-1]
9           swap(p[i], p[i-1]), recalc sumA

```

• PTAS для bin packing

Напомним задачу: даны $0 < a_i \leq 1$, разбить их на min число корзин размера 1.

• План решения

1. Сперва рассмотрим ситуацию, когда все $a_i \geq \varepsilon$ и различных a_i не более K
2. Отбросим ограничение “не более K ” (округлим все a_i к одному из K значений).
3. Отбросим ограничение $a_i \geq \varepsilon$ (в начале выкинем $a_i < \varepsilon$, в конце жадно добавим).

1 Итак, все $a_i \geq \varepsilon$ и различных a_i не более $K \Rightarrow$ в каждую корзину влезает не более $M = \frac{1}{\varepsilon}$ предметов \Rightarrow различных типов корзин не более $R = K^M$. Нам не нужно больше n корзин, есть не более $P = \binom{n+R-1}{R}$ вариантов выбрать n корзин R типов.

Заметим, что $1/\varepsilon, K, M, R$ – константы. Огромные, но константы.

$P \leq \frac{1}{R!}(n+R)^R \Rightarrow P$ – полином от n , переберём все P вариантов, выберем оптимальный.

2 Выберем $K = \lceil \frac{1}{\varepsilon^2} \rceil$. Отсортируем a_i , разобьём их на K групп, в каждой не более $\lfloor n\varepsilon^2 \rfloor$ элементов. Изменим в каждой группе все числа на максимальное, теперь различных $\leq K$. Исходная задача – A , новая – A' . A' поэлементно больше $A \Rightarrow \forall$ распределение для A' является распределением для A . Рассмотрим A'' , полученную из A' удалением $\lfloor n\varepsilon^2 \rfloor$ максимальных. A'' поэлементно меньше $A \Rightarrow k(A'') \leq k(A) \Rightarrow k(A') \leq k(A'') + \lfloor n\varepsilon^2 \rfloor \leq k(A) + \lfloor n\varepsilon^2 \rfloor$.

$\forall i a_i \geq \varepsilon \Rightarrow OPT \geq n\varepsilon \Rightarrow k(A') \leq k(A) + \varepsilon \cdot OPT = (1 + \varepsilon)OPT$.

3 I – исходная задача, I' – задача без $a_i < \varepsilon$. Мы уже получили решение для I' не хуже $(1 + \varepsilon)k(I')$. Добавим, используя любую жадность, например FF, все $a_i < \varepsilon$.

Если ответ не изменился, отлично $k(I') \leq k(I) \Rightarrow (1 + \varepsilon)k(I') \leq (1 + \varepsilon)k(I)$.

Если ответ изменился, то все корзины кроме последней заполнены больше чем на $1 - \varepsilon \Rightarrow (1 - \varepsilon)(k(I') - 1) + 1 \leq \sum_{\text{все}} a_i \leq k(I) \Rightarrow k(I') \leq \frac{1}{1 - \varepsilon}(k(I) - 1) + 1 \leq (1 + 2\varepsilon)k(I) + 1$.

Замечание 9.2.7.

В части (1) мы получили точное решение.

В части (2) мы получили $(1 + \varepsilon)$ приближении, и лишь

В части (3) приближение стало $(1 + 2\varepsilon)$ -ОПТ + 1.

9.3. Set cover

Задача: даны множества A_1, A_2, \dots, A_m , хотим покрыть $U: |U| = n$ минимальным числом A_i .

Шаг жадности: добавить в ответ множество A_i , покрывающее максимальное число ещё не покрытых элементов U .

Упражнение 9.3.1. Придумать реализацию за $\mathcal{O}(n + \sum |A_i|)$.

Теорема 9.3.2. Получили $(\ln n)$ -ОПТ алгоритм.

Доказательство. Пусть оптимальный ответ равен k .

За один шаг жадность покроет $\geq \frac{1}{k}n$ элементов $U \Rightarrow$

размер U уменьшится в $\frac{k-1}{k}$ раз \Rightarrow за t шагов размер U уменьшится в $(\frac{k-1}{k})^t$ раз.
Осталось взять t : $(\frac{k-1}{k})^t < \frac{1}{n} \Leftrightarrow (\frac{k}{k-1})^t > n \Leftrightarrow t > \frac{1}{\ln k - \ln(k-1)} \ln n > k \ln n$. ■

Упражнение 9.3.3. Найдите $(\ln n)$ -ОРТ приближение взвешенной задачи – вес A_i равен w_i .

Лекция #10: Центроидная декомпозиция

10 марта

10.1. Построение и минимум на пути дерева

Пусть дано дерево с весами на рёбрах или в вершинах. Рассмотрим три задачи вида «построить структуру данных, которая умеет за $\mathcal{O}(1)$ вычислять минимум...»

1. на пути от любой вершины до корня;
2. на любом пути, проходящем через корень;
3. на любом пути

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

Def 10.1.1. *Центроид* – вершина, при удалении которой, размеры всех компонент $\leq \frac{n}{2}$.

Чтобы решить третью задачу найдём за $\mathcal{O}(n)$ центроид v (задача с практики); все пути, проходящие через v обработаем, используя (2); удалим v из дерева и рекурсивно построим структуру от оставшихся компонент.

В итоге каждая вершина ровно один раз будет найдена, как центроид. У каждого центроида v есть центроид-предок p_v в дереве рекурсии, глубина в дереве рекурсии d_v и компонента связности $C(v)$, в которой он был найден, как центроид.

Def 10.1.2. *Центроидной декомпозицией* будем называть два массива – p_v и d_v . Первый из массивов задаёт так называемое “дерево центроидной декомпозиции”.

Замечание 10.1.3. Зная только глубины d_v , можно восстановить компоненты $C(v)$. Для этого нужно от центра v запустить dfs, который проходит только по вершинам большей глубины. Поэтому $C(v)$ хранить **не нужно**.

Lm 10.1.4. Для центроидной декомпозиции нужно $\Theta(n)$ памяти.

Lm 10.1.5. Глубина дерева центроидной декомпозиции не более $\log_2 n$.

Lm 10.1.6. Суммарный размер компонент $\mathcal{O}(n \log n)$.

Следствие 10.1.7. В задаче поиска min на пути суммарный размер предподсчёта $\mathcal{O}(n \log n)$.

Чтобы предподсчитать все минимумы от v до вершин $C(v)$, запускаем dfs:

```

1 void precalcMin(int v, int u, int parent, int curMinimum) { // v - центроид
2   curMinimum = min(curMinimum, w[u]); // пусть веса в вершинах
3   f[depth[v]][u] = curMinimum; // на уровне depth[v] каждая вершина u встретится ≤ 1 раза
4   for (int x : graph[u])
5     if (depth[x] >= depth[v] && x != parent)
6       precalcMin(v, x, u, curMinimum);
7 }
8 precalcMin(v, v, -1, INT_MAX);

```

Lm 10.1.8. Для любого пути $[a..b]$ на дереве, есть такой центр $v: v \in path[a..b]$ и $a, b \in C(v)$.

Чтобы найти такой центр v рассмотрим пути от a и b вверх в дереве центроидной декомпозиции и возьмём наименьшего общего предка (LCA) a и b – самый нижний такой центр, что $a, b \in C_v$.

$$\text{минимум на пути}[a..b] = \min(f[d_v, a], f[d_v, b])$$

Сейчас мы умеем LCA искать только за \mathcal{O} (глубины дерева). В нашем случае это $\mathcal{O}(\log n)$.

На практике научимся за $\mathcal{O}(\log \log n)$, а в скором будущем вообще за $\mathcal{O}(1)$.

10.2. Реализация

```

1 // Эта функция нужна только, чтобы посчитать размер компоненты связности
2 int calc_size( int v, int parent = -1 ) {
3     int size = 1;
4     for (int x : graph[v])
5         if (x != parent && d[x] == -1)
6             size += calc_size(x, v);
7     return size;
8 }
9 // Зная размер дерева n, считаем размеры поддеревьев, параллельно ищем центроид
10 int get_centroid( int v, int parent, int n, int &centroid ) {
11     int size = 1;
12     for (int x : graph[v])
13         if (x != parent && d[x] == -1)
14             size += get_centroid(x, v, n, centroid);
15     if (size * 2 >= n && centroid == -1)
16         centroid = v;
17     return size;
18 }
19 // Главная функция построения, v - любая вершина компоненты
20 void build( int v, int parent, int depth ) {
21     int centroid = -1;
22     get_centroid(v, -1, calc_size(v), centroid);
23     d[centroid] = depth, p[centroid] = parent;
24     dfs(centroid, -1, 0);
25     for (int x : graph[centroid])
26         if (d[x] == -1)
27             build(x, centroid, depth + 1);
28 }
29 d = vector<int>(n, -1);
30 p = vector<int>(n, -1);
31 build(0, -1, 0);

```

На самом деле, чтобы решать задачи, нам не нужна прям центроидная декомпозиция, нам нужно выбирать вершину `centroid` так, чтобы глубина дерева декомпозиции была $\leq \log_2 n$.

Благодаря этому, можно обойтись без `calc_size`:

```

1 void build( int v, int parent, int depth, int n ) {
2     int centroid = -1;
3     get_centroid(v, -1, n, centroid); // n - оценка сверху на размер компоненты
4     if (centroid == -1) centroid = v; // если оценка наглая, if мог не сработать
5     ...
6     build(x, centroid, depth + 1, (n + 1) / 2);
7 }

```

И, наконец, часть, которая ищет минимум. Нужно для каждого центроида насчитать dfs-ом минимумы на путях до корня.

```
1 f = vector<vector<int>>(LOG, vector<int>(n));
2 void calc_min( int depth, int curMin, int v, int parent ) {
3     curMin = min(curMin, weight[v]); // пусть веса будут в вершинах
4     f[depth][v] = curMin; // на каждой глубине v встретится не более раза
5     for (int x : graph[v])
6         if (x != parent and d[x] == -1) // не ходим в уже помеченные вершины
7             calc_min(depth, curMin, x, v);
8 }
9 int get_min( int a, int b ) {
10     int lca = get_lca(a, b); // в дереве центроидной декомпозиции, заданной p[]
11     return min(f[d[lca]][a], f[d[lca]][b]);
12 }
```


Лекция #11: BST и AVL

9 апреля

11.1. BST, базовые операции

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

Def 11.1.1. *BST – binary search tree (бинарное дерево поиска). В каждой вершине пара $\langle x, y \rangle$. Левое поддерево содержит пары $\langle x, y \rangle$ со строго меньшими x . Правое поддерево содержит пары $\langle x, y \rangle$ со строго большими x .*

Как видно из определения, мы пока предполагаем, что все x_i различны.

y_i – просто дополнительные данные. Например $\langle x_i, y_i \rangle$ – студент, x_i – имя студента.

Хранить вершины дерева будем, как `struct Node { int x, y; Node *p, *l, *r; }`;

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

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

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

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

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

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

`next(v)`: если есть правый сын, спуститься в него и до упора влево, иначе идти вверх, пока не найдём меньший ключ.

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.

Пока преимуществом над хеш-таблицей является только `next`.

Вскоре появятся и другие операции, например, на практике сделаем `Node* lower_bound(x)`.

Все описанные операции: `add`, `del`, `find`, `next`, `lower_bound` работают за глубину дерева.

Чтобы в этом был толк, скоро изучим способы поддерживать глубину $\mathcal{O}(\log n)$.

• Ускорение некоторых операций

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

◦ `find` за $\mathcal{O}(1)$: хеш-таблица, для каждого x хранящая `Node* v`, содержащую x .

◦ `next/prev` за $\mathcal{O}(1)$: добавляем на вершинах двусвязный список, это называют *прошивкой*.

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

• Равные 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 сильнее ослабить определение – равные разрешать хранить и слева, и справа.

11.2. Немного кода

```

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

```

• Добавление

```

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

```

Более короткая версия, использующая ссылку на указатель:

```

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 }

```

Алгоритм 11.2.1. Персистентная версия.

```

1 Node* add(Node* v, int x) {
2   if (!v)
3     return new Node(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 уже созданные Node-ы никогда не изменяются (*immutable*). Поэтому можно писать так: `Node *root2 = add(root1, x)`, после чего у вас есть два корректных дерева – старое с корнем в `root1` и новое с корнем в `root2`.

• Для любителей экзотики

Реклама Node**. Полюбуйтесь на нерекурсивные `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; // подцепили 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 }

```

11.3. AVL (Адэльсон-Вельский, Ландис'1962)

Def 11.3.1. 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 – $O(\log n)$.

Def 11.3.2. Высота вершины v – максимальное из расстояний от v до листьев в поддереве v .

Def 11.3.3. BST называют AVL-деревом, если оно удовлетворяет AVL-инварианту: в каждой вершине разность высот детей не более одного: $\forall v |h(v.l) - h(v.r)| \leq 1$.

Lm 11.3.4. Глубина AVL-дерева $O(\log(n))$

Доказательство. При спуске по любому ребру высота поддерева уменьшается или на один, или на два. Значит, если высота AVL-дерева h , то на первых $\lfloor h/2 \rfloor$ уровнях полное дерево. Значит $h \leq C \log(n)$. ■

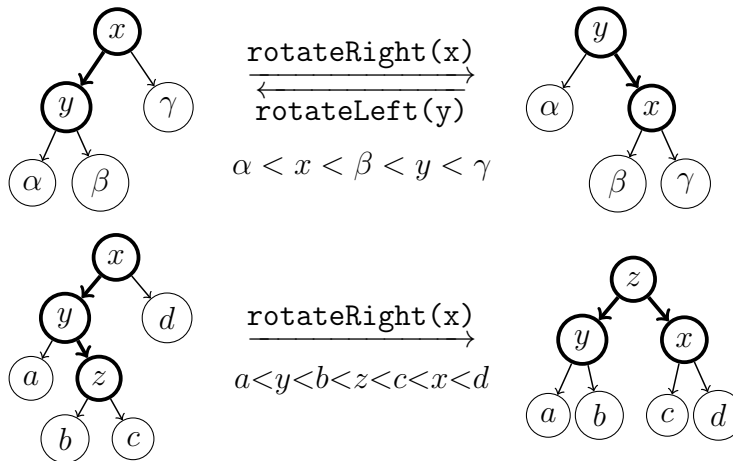
Добавление в AVL-дерево сперва делается также, как в обычное BST.

А затем 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 $(v \rightarrow v.l)$.
- (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 ребрам $(v \rightarrow v.l \rightarrow v.l.r)$.



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 11.3.5. При добавлении в AVL-дерево происходит $\mathcal{O}(1)$ присваиваний указателей.

К сожалению, высоты могут меняться 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.

Подробнее про удаление из AVL на практике.

Замечание 11.3.6. И в добавлении, и в удалении при подъёме вверх и перебалансировке можно пользоваться ссылками на родителя. Но удобнее всю перебалансировку делать именно на обратном ходу рекурсии.

Замечание 11.3.7. Раз отцы не нужны, дерево можно сделать персистентным (см. [разд. 11.5](#)). Персистентное вращение напишем на практике.

11.4. Split/Merge

Def 11.4.1. $split(t, x)$ делит t на $l = \{a \mid a < x\}$ и $r = \{a \mid a \geq x\}$.

Def 11.4.2. 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 11.4.3. *merge* – операция, обратная *split*.

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

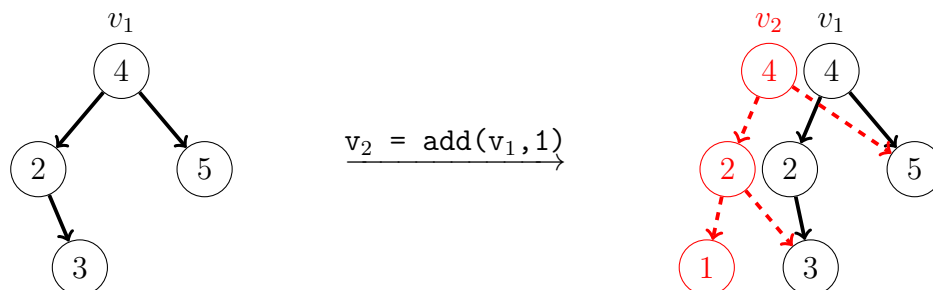
Замечание 11.4.4. По умолчанию *split* и *merge* ломают деревья, получаемые на вход. Но их, как и все другие процедуры работы с BST, можно сделать персистентными.

11.5. Персистентность

Def 11.5.1. *Персистентной* называется структура данных, которая при модификации сохраняет доступ ко всем предыдущим версиям.

У нас уже есть персистентный `add` в обычное BST 11.2.1. Отличие от не персистентной версии в том, что на обратном ходу рекурсии вместо того, чтобы менять старые вершины, мы создаём новые. По аналогии можно реализовать `del`.

Замечание 11.5.2. Каждая отдельная версия персистентного дерева – дерево. Все версии в совокупности образуют ациклический граф.



Замечание 11.5.3. У вершины теперь несколько “отцов”, поддерживать их не получается.

Замечание 11.5.4. Есть несколько способов сделать структура персистентной. Рассмотренный нами называется `path copying`. Также полезна `fat nodes`.

11.6. Дополнительные операции, групповые операции

Можно поддерживать в вершине v размер поддерева $v \rightarrow \text{size}$.

Всегда, когда поддерево вершины v меняется, считаем $v \rightarrow \text{size} = v \rightarrow \text{l} \rightarrow \text{size} + v \rightarrow \text{r} \rightarrow \text{size} + 1$

Чтобы не обрабатывать отдельно случай “ $v \rightarrow \text{l} = \emptyset$ ”, 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 некоторое число y_v .

Научимся для примера обрабатывать запрос “`getMin(l,r) = min{y_v | l ≤ x_v ≤ 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     ( l <= v->x && v->x <= r ) ? v->y : INT_MAX
10  );
11 }
12 int result = getMin(root, INT_MIN, INT_MAX, l, r);

```

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

• Модификация на отрезке

Хотим со всеми $\{y_v | l ≤ x_v ≤ r\}$ сделать $y_v += \Delta$ (групповая операция).

Если мы захотим честно обновить все нужные y_v , это займёт линейное время.

Чтобы получить время $\mathcal{O}(\log n)$ на запрос, воспользуемся отложенными операциями:

у каждой вершины v будем хранить `v->add` – число, которое нужно прибавить ко всем вершинам в поддереве v . Запрос “прибавления на отрезке” (`rangeInc`) теперь обрабатывается по аналогии с `getMin`, есть три случая: (1) $[vl, vr] \cap [l, r] = \emptyset$, (2) $vl, vr \subset [l, r]$ и (3) “отрезки пересекаются”.

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

```

1 void pushTo( 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   pushTo(v->l, v->add);
7   pushTo(v->r, v->add);
8   v->add = 0; // при этом минимум не изменился
9 }

```

11.7. Неявный ключ

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

Возьмём дерево по ключу “индекс в массиве”: $x_v = i, y_v = a_i$.

Тогда, если в каждой вершине хранить размер поддерева и минимум y_v в поддереве, у нас уже есть операции `getValue(i)`, `add(i, data)`, `del(i)`, `getMin(l,r)`.

Нужно только после `add(i, data)` и `del(i)` пересчитывать ключи (индексы в массиве)...

Но зачем их пересчитывать и вообще хранить, если мы ими *нигде* не пользуемся?

Идея дерева по неявному ключу: можно просто не хранить ключ.

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

11.8. Reverse на отрезке

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

```

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

Техника “высплитим отрезок $[l,r)$ ” позволяет простой отложенной операцией `v->isReversed` реализовать `reverse(l, r)` (перевернуть отрезок массива).

Замечание 11.8.1. `getMin`, `rangeInc` можно было делать спуском по дереву, сделать `reverse` без выспличивания отрезка не получится.

11.9. Итоги

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

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

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

11.10. Персистентность: итоги

Любое дерево поиска, где не обязательно нужны отцы, можно сделать персистентным. В частности AVL. Время работы останется пропорциональным глубине дерева, для AVL это $\mathcal{O}(\log n)$.

Следствие 11.10.1. \exists персистентное дерево по неявному ключу с операциями за $\mathcal{O}(\log n)$.

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

Следствие 11.10.2. Любую структуру данных можно сделать персистентной, с замедлением в $\mathcal{O}(\log n)$ раз. Просто поменяем все массивы на персистентные массивы.

Основной минус персистентных массивов – \forall операция создаёт $\mathcal{O}(\log n)$ новых вершин дерева.

Лекция #12: B-tree и Treap

16 апреля

12.1. B-дерево (Bayer, McCreight'1972)

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

Def 12.1.1. 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 12.1.2. При $k \geq 2$ 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.

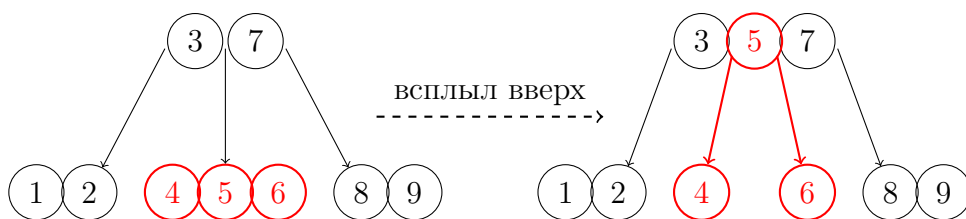
12.1.1. Поиск по B-дереву

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 $10^9 \dots 10^{10}$ записей. Тогда в оперативную память такая база не помещается, и при чтении из базы происходит чтение с жесткого диска. При чтении с жесткого диска за одну операцию чтения мы читаем не 1 байт, а сразу блок, например из 4 килобайт. B-дерево, не ухудшая процессорное время, минимизирует число обращений к диску.

12.1.2. Добавление в B-дерево

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



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

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

12.1.3. 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 дерева.

Замечание 12.1.3. В удалении и добавлении за $\mathcal{O}(\log n)$ мы пользуемся тем, что дерево поиска умеет делать split и merge за логарифмическое время.

12.1.4. 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](#).

12.1.5. Split/Merge

Делаются за высоту дерева. Будут в практике, как упражнения. У деревьев должно быть одинаковое k , но могут быть произвольные высоты.

12.2. Производные B-дерева

12.2.1. 2-3-дерево (Hopcroft'1970)

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

12.2.2. 2-3-4-дерево и RB-дерево (Bayer'1972)

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

2-3-4-дерево ничем не лучше 2-3-дерева. Зато оно эквивалентно красно чёрному дереву (RB-дерево). А RB-дерево это быстрая и популярная реализация сбалансированного дерева поиска. Например, используется в C++ .STL внутри `set<>`.

Def 12.2.1. Рассмотрим *BST*, в котором каждая вершина покрашена в красный или чёрный. Также в каждое место “отсутствия сына” мысленно добавим фиктивного сына. Дерево называется *красно чёрным*, если:

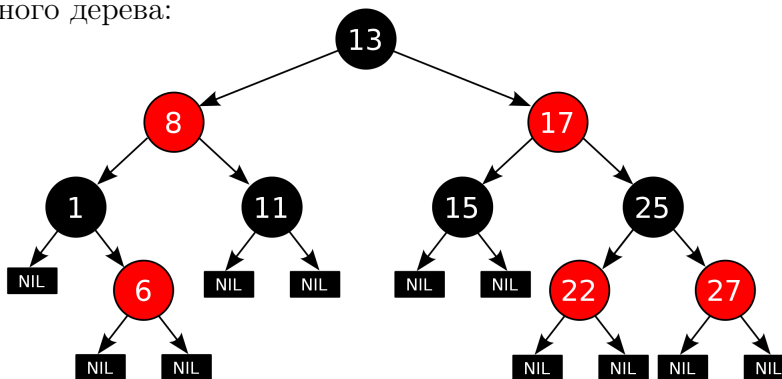
- На пути от корня до любой фиктивной вершины одинаковое число чёрных вершин.
- Сыном красной вершины не может быть красная вершина.
- Корень – чёрная вершина.

Lm 12.2.2. Есть преобразование между красночёрными и 2-3-4-деревьями.

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

Замечание 12.2.3. Это почти биекция. В случае “один красный сын” при сливании его с корнем теряется информация правый он или левый.

Пример красно-чёрного дерева:



12.2.3. AA-дерево (Arne Anderson'1993)

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

Def 12.2.4. *RB-дерево называется AA-деревом, если любая красная вершина является правым сыном своего отца.*

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

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

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

12.3. Treap (Seidel, Aragon'1989)

Def 12.3.1. *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 12.3.2. 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 12.3.3. *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 12.3.4. 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 12.3.5. 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 $E(\sum size_i) \leq 2n \ln n$, 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 12.3.6. Декартово дерево (treap) от множества ключей $\{x_i\}$ – декартово дерево (cartesian tree) на $\langle x_i, random \rangle$.

Lm 12.3.7. Треап, как BST, является RBST

Следствие 12.3.8. Матожидание средней глубины вершины в Треап – $\mathcal{O}(\log n)$

Lm 12.3.9. Матожидание средней глубины вершины в Треап – $\mathcal{O}(\log n)$

Lm 12.3.10. Матожидание высоты treap $\mathcal{O}(\log n)$

12.4. Операции над Треап

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 “ $\geq x$ ”. Merge(L, R) – обратная к Split операция. В частности Merge, что все ключи в L меньше ключей в R .

Lm 12.4.1. 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 старое 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 $t \rightarrow x$ попал в левую половину, то элементы меньше x – это корень, всё его левое поддерево и какая-то часть правого. Поэтому вызовем рекурсивно от $t \rightarrow r$, левую из образовавшихся частей подвесить к корню, правую запишем в r .

12.5. 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 массив персистентным мы умеем двумя способами:

1. Дерево поиска по неявному ключу. Оно кроме всего прочего будет Copye.
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 предпочтительней – он быстрее, его реализация проще.

12.5.1. Offline

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

12.5.2. 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:

```

1 struct Queue {
2   Stack L, R; // L для pop, R для push
3   Stack L1, R1, tmp; // вспомогательные стеки, итого 5 стеков
4   int state, copied;
5
6   Queue Copy() const {
7     return Queue(L.copy(), R.copy(), L1.copy(), R1.copy(), tmp.copy(), state, copied);
8   }
9   int Front() const {
10    return L.front(); // очередь не пуста => L не пуст!
11  }
12  pair<Queue, int> Pop() const {
13    Queue res = Copy();
14    int data = res.L.pop();
15    forn(i, 3) res.Step();
16    return make_pair(res, data);
17  }
18  Queue Push(int data) const {
19    Queue res = Copy();
20    res.R.push(data);
21    forn(i, 3) res.Step();
22    return res;
23  }
24
25  void Step() { // основной шаг переворачивания; этот метод не const!
26    if (state == DO_NOTHING) {
27      // if у на достаточно большой pop-стек then рано волноваться
28      if (L.size > R.size) return;
29      // В этот момент L.size == R.size
30      R1 = R, R = new Stack(), tmp = new Stack();
31      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.

12.5.3. Простой персистентный дек (pairing)

```

1 struct Deque<T> {
2     T l; // элемент или null
3     Deque<pair<T,T>> m; // все остальные разбиты на парах
4     T r; // элемент или null
5
6     Deque<T> push_back( T x ) { // возвращаем новую версию
7         if (r == null) return {l, m, x};
8         return Deque(l, m.push_back(pair(r, x)), null);
9     }
10    pair<T, Deque<T>> pop_back() { // возвращаем то, что достали, и новую версию
11        if (r != null) return pair(r, Deque(l, m, null));
12        if (m == null) return pair(l, Deque(null, null, null));
13        <x, m1> = m.pop_back();
14        return pair(x.second, Deque(l, m1, x.first));
15    }
16 }

```

12.5.4. Треар и избавление от Y

В структуре Treap мы пользовались игреками для того, чтобы оценить полученную структуру как RBST. Попробуем избавиться от хранения y , но всё ещё сохранить случайность дерева.

Обсудим только операции Split и Merge, раз остальные выражаются через них. Функция Split не использовала y , поэтому не поменяется. Как поменять функцию Merge?

Если мы хотим сmerzжить дерево A и дерево B , то кинем монетку: с вероятностью $\frac{|A|}{|A|+|B|}$ выберем в качестве корня корень A , с вероятностью $\frac{|B|}{|A|+|B|}$ выберем в качестве корня корень B .

Почему именно такие вероятности? Каждый из ключей в A является корнем A с вероятностью $1/|A|$, каждый из ключей B является корнем B с вероятностью $1/|B|$. Получается, что каждый ключ объединения является корнем с вероятностью $\frac{1}{|A|+|B|}$.

По индукции, корни всех оставшихся поддеревьев тоже будут выбраны случайно.

Таким способом можно написать Treap и в обычном случае, хотя это и менее эффективно. Основное достоинство отсутствующих u это возможность делать персистентный merge вершины с самой собой. Например, так можно реализовать операцию копирования подотрезка массива, если массив хранится как персистентный Treap по неявному ключу.

12.5.5. (*) Частичная персистентность: fat nodes

Def 12.5.1. Структура данных частично персистентна, если допускает модификацию последней версии и get-запрос к любой.

Замечание 12.5.2. Частично персистентный массив тривиально сделать за $\mathcal{O}(1)$ памяти/времени на модификацию и $\mathcal{O}(\log n)$ времени на обращение: храним независимо для каждой ячейки вектор пар $\langle \text{время модификации}, \text{значения} \rangle$; обращение = бинпоиск.

Теорема 12.5.3. Если у нас есть ссылочная структура (BST, список, двусвязный список) с ограниченной входящей степенью (во всех перечисленных структурах ≤ 2), можно получить амортизированную оценку $\mathcal{O}(1)$ времени и на модификацию, и на обращение.

Доказательство. Техника fat-nodes.

Можно было бы воспользоваться тривиальным решением и в каждой вершине хранить вектор $\langle \text{time}_i, \text{version}_i \rangle$, при каждом обращении к каждой вершине пришлось бы делать бинпоиск.

Оптимизация: будем для каждой вершины хранить не более k версий. Когда у вершины приходится создавать $(k+1)$ -ую версию, создадим копию вершины из одной новой версии, всем предкам вершины рекурсивно создадим новую версию, направив исходящее ребро в копию.

```

1 # Vertex : { parents, versions }
2 def newVersion(t, v, newVersion):
3     if len(v.versions) < k:
4         v.versions.append((t, newVersion))
5         return
6     u = Vertex(v.parents, [(t, newVersion)])
7     for p in v.parents:
8         newVersion(t, p, redirectLink(p, v, u))

```

Осталось выбрать число k . Возьмём любое, чтобы $\forall v |v.parents| < k$. Время работы и память оцениваются одинаково. Потенциал $\varphi = \sum |v.versions|$, где сумма берётся только по вершинам, содержащим последнюю версию. Когда мы создаём новую вершину, мы уменьшаем потенциал на k и увеличиваем на $|v.parents| \Rightarrow$ уменьшаем. ■

• Примеры:

- Treap. Обычная операция split. Создаст $\mathcal{O}(\log n)$ новых вершин, отработает за $\mathcal{O}(\log n)$. Её fat-node-персистентный аналог создаст асимптотически столько же вершин за то же время.
- AVL по явному ключу. Операция add модифицирует ссылки у $\mathcal{O}(1)$ вершин. Высоты – промежуточная информация, её можно смело портить у старых версий. Значит n персистентных операций add в AVL используют $\mathcal{O}(n)$ памяти и $\mathcal{O}(n \log n)$ времени.
- AVL по неявному ключу. Нужно хранить размеры \Rightarrow поменять $\Theta(\log n)$ вершин.
- Дерево отрезков. Одна модификация в точке изменяет $\log n$ вершин $\Rightarrow n$ модификаций изменят $n \log n$ вершин \Rightarrow создадут $\mathcal{O}(n \log n)$ новых.

Следствие 12.5.4. Если взять scanline с деревом отрезков или, мы не получили преимущества перед обычной персистентностью. Если взять scanline с BST, то важно, какое именно BST использовать, и какие операции с ним. Пусть используем AVL с явным ключом, операции только add \Rightarrow profit.

Пример задачи: online проверка, лежит ли точка внутри невыпуклого многоугольника (обобщение: в какой грани планарного графа лежит точка). Научились её решать за $\langle \mathcal{O}(n), \mathcal{O}(\log n) \rangle$.

Лекция #13: Splay и корневая оптимизация

24 апреля

13.1. Rope

Def 13.1.1. *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.

13.2. 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 13.2.1. Матожидание времени работы $find$ – $\mathcal{O}(\log n)$.

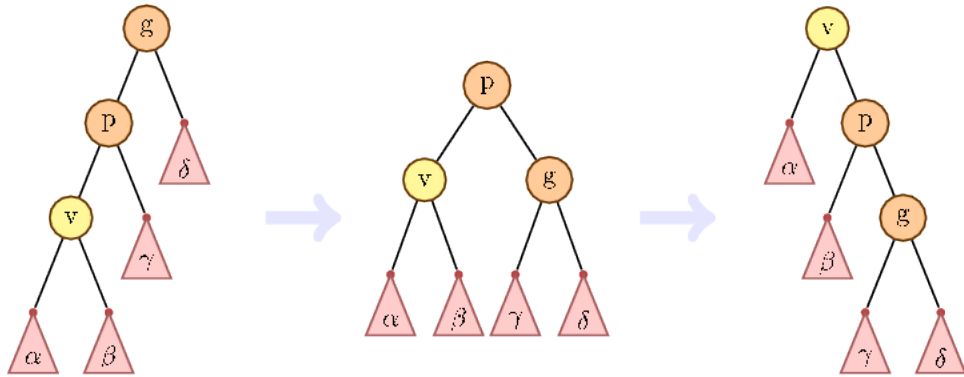
Доказательство. На каждом шаге `for` матожидание число шагов $\mathcal{O}(1)$. ■

Чтобы удалить i -элемент, сделаем $\text{find}(i)$, 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.

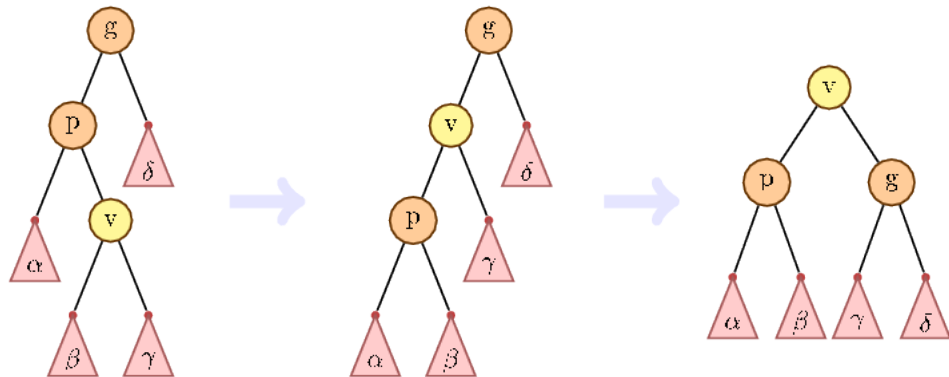
13.3. Splay tree

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 $\text{Splay}(v)$, которая последовательными вращениями протолкнёт v до корня.

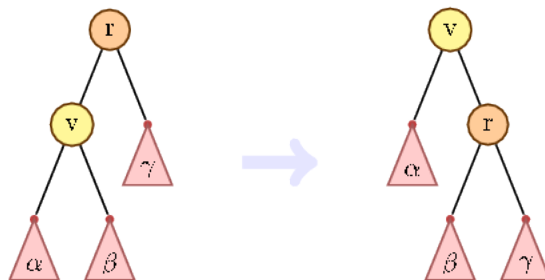
- **Zig-zig** вращение



- **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 13.3.1. $x, y > 0, x + y = 1 \Rightarrow \log x + \log y \leq -2$

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

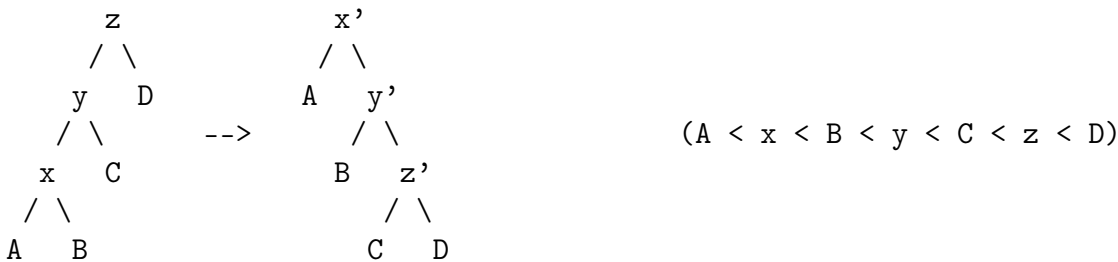
Lm 13.3.2. $x, y > 0, x + y = C \Rightarrow \log x + \log y \leq 2 \log C - 2$

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

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

Теорема 13.3.3. $\forall v, u \ a_{v \rightarrow u} \leq 3(R_u - R_v) + 1 = 3 \log \frac{\text{size}_u}{\text{size}_v} + 1$

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



Мы хотим показать $F \leq 3(R_{x'} - R_x) \Leftrightarrow R_{z'} \leq 2R_{x'} - R_x - 2 \Leftrightarrow R_{z'} + R_x \leq 2R_{x'} - 2$.

Теперь вспомним, что $R_{z'} = \log(C + D + 1)$, $R_x = \log(A + B + 1) \stackrel{\text{лемма}}{\Rightarrow} R_{z'} + R_x \leq 2 \log(A + B + C + D + 2) - 2 \leq 2R_{x'} - 2$. ■

Следствие 13.3.4. Среднее время одной операции в splay-дереве – $\mathcal{O}(\log n)$.

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

Замечание 13.3.5. В теорему вместо size_v можно подставить любой взвешенный размер: $\text{size}_v = w_v + \text{size}_l + \text{size}_r$, где w_v – вес вершины.

Если $w_v \geq 1$, то потенциал в каждый момент неотрицательный.

Полученная оценка на время работы операции Splay позволяет получать более крутые оценки на операции во многих случаях.

13.4. Статическая оптимальность

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

Def 13.4.1. BST называется статически оптимальным, если минимизирует матожидание времени запроса: $\sum_i p_i d_i \rightarrow \min$.

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: $x_1 < x_2 < \dots < x_n$.

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

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

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 $i[l, r-1] \leq i[l, r] \leq i[l+1, r]$, 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.

13.5. Splay Tree и оптимальность

Теорема 13.5.1. Пусть к splay-дереву поступают только запросы $\text{find}(v)$, k_v – количество обращений к вершине v , $m = \sum k_v$. Тогда суммарное время всех find равно $\mathcal{O}(m + \sum_v k_v \log \frac{m}{k_v})$.

Доказательство. Если покажем, что $(3 \log \frac{m}{k_v} + 1)$ – время на один запрос к вершине v , получим требуемую сумму.

Эта оценка на время получается из 13.3.3 подстановкой веса вершины $w_v = k_v$.

Тогда $size_{root} = m$ и время $splay(v) \leq 3 \log \frac{size_{root}}{size_v} + 1 \leq 3 \log \frac{size_{root}}{k_v} + 1$. ■

Для задачи статически оптимального дерева поиска также известно жадное C -ОПТ решение. Из существования этого решения следует, что оценка на Splay дерево из предыдущей теоремы также отличается от статически оптимального BST всего лишь в константу раз.

Также есть гипотеза о динамической оптимальности Splay-дерева, которая формулируется следующим образом:

Пусть есть какой-то абстрактный алгоритм работы с BST. К нему поступают запросы поиска, которые работают за $d(key) + 1$. Также между запросами к BST можно вращать, за 1 времени на каждый поворот. Не существует такого сценария запросов, в котором Splay дерево будет работать хуже, чем $\mathcal{O}(n + t_{opt})$.

В **практике** мы также доказали теорему о времени работы бора и Static Finger теорему.

В той же практике предлагается более быстрая top-down реализация splay-дерева.

13.6. SQRT decomposition

13.6.1. Корневая по массиву

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

• Сумма на отрезке и изменение в точке

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

```

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 $\mathcal{O}(\sqrt{n})$.

13.6.2. Корневая через split/merge

Дополним предыдущие две задачи операциями `insert(i,x)` и `erase(i)`.

Будем хранить `vector` или `list` кусков. Здесь i -й кусок – это отрезок $[l_i, r_i)$ нашего массива, мы храним его как `vector`. Сам массив мы не храним, только его разбиение на куски.

Когда приходит операция `insert/erase`, ищем, про какой она кусок за $\mathcal{O}(\sqrt{n})$. Теперь сделаем эту операцию в найденном куске за $\mathcal{O}(\sqrt{n})$. При этом кусок мог уменьшиться/увеличиться.

Кусок размера меньше \sqrt{n} сmergeм 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 – $\mathcal{O}(\sqrt{n})$.

13.6.3. Корневая через split/rebuild

Храним то же, что и в прошлой задаче.

Операция `split(i)` – сделать так, чтобы i -й элемент был началом куска (если это не так, кусок, в котором лежит i , нужно разделить на два). В задачах про сумму и минимум `split` = $\mathcal{O}(\sqrt{n})$.

Ещё удобнее, если `split` возвращает номер куска, началом которого является i -й элемент.

Любой запрос на отрезке $[l, r)$ теперь будем начинать со `split(r)`, `split(l)`.

И вообще хвостов нигде нет, всегда можно посплитить.

Тогда код любой функции теперь прекрасен своей лаконичностью:

```

1 vector<Part> p;
2 void insert(int i, int x) {
3     i = split(i);
4     a[n++] = x; // добавили x в конец исходного массива
5     p.insert(i, Part(n - 1, n));
6 }
7 void erase(int i) {
8     split(i + 1);
9     p.erase(split(i));
10 }
11 int get_sum(int l, int r) { // [l,r)
12     int sum = 0;
13     for (r = split(r), l = split(l); l < r; l++)
14         sum += p[l].sum;
15     return sum;
16 }

```

Есть небольшая проблема – число кусков после каждого запроса вырастет на $\mathcal{O}(1)$.

Давайте, если число кусков $\geq 3\sqrt{n}$, 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.

13.6.4. Применение

Задачи про минимум, сумму мы уже умели решать через BST, все операции за $\mathcal{O}(\log n)$.

Из нового мы пока научились только запросы сумма/изменение “перекашивать”:

$\langle \mathcal{O}(\log n), \mathcal{O}(\log n) \rangle \rightarrow \langle \mathcal{O}(\sqrt{n}), \mathcal{O}(1) \rangle$ и $\langle \mathcal{O}(1), \mathcal{O}(\sqrt{n}) \rangle$.

На самом деле спектр задач, решаемых корневой оптимизацией гораздо шире. Для примера приведём максимально ужасную задачу. Выполнять нужно следующие запросы:

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

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

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

```

1 void reverse(l, r) {
2     r = split(r), l = split(l);
3     reverse(p + l, p + r);
4     for (; l < r; l++)
5         p[l].is_reversed ^= 1;
6 }

```

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”.

13.6.5. Оптимальный выбор k

Не во всех задачах выгодно разбивать массив ровно на \sqrt{n} частей по \sqrt{n} элементов.

Обозначим число кусков k . В каждом куске $m = \frac{n}{k}$ элементов.

На примере последней задачи оптимизируем k .

• `split/rebuild`

Время `inner_split` куска: $\mathcal{O}(m \log m)$ ¹, так как нам нужно сортировать.

Время `split(i)`: $\mathcal{O}(k) + \text{inner_split} = \mathcal{O}(k + m \log m)$.

Время `reverse` и `add_to_all`: $\mathcal{O}(k) + \text{split}(i) = \mathcal{O}(k + m \log m)$.

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: $T(\text{split}) + T(\text{insert}) + T(\text{sum}) + \dots + \frac{1}{k}T(\text{rebuild}) = \Theta((k + m) \log m + \frac{1}{k}n \log n) \rightarrow \min$

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 $T(\text{split}) + T(\text{insert}) + T(\text{sum}) + \dots = \Theta(m + k \log m)$

Заменили $\log m$ на $\log n$, сумму на максимум \Rightarrow решаем $\frac{n}{k} = k \log n$. Итого $k = \sqrt{n / \log n}$.

13.6.6. Корневая по запросам, отложенные операции

Задача: даны числа, нужно отвечать на запросы `lower_bound`. Самое простое и быстрое решение – отсортировать числа, на сортированном массиве вызывать стандартный `lower_bound`.

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](#).

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

13.7. Другие деревья поиска

В мире есть ещё много деревьев поиска, которые не охватывает курс.

Отдельное внимание хочется обратить на

Finger tree – чисто функциональный горел, умеющий обращаться к концам за $\mathcal{O}(1)$.

Finger Search tree – дерево поиска, которое почти всё делает за амортизированное $\mathcal{O}(1)$.

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

-- Size Balanced Tree.

Лекция #14: Дерево отрезков

30 апреля

Дерево отрезков (range tree) – это и структура данных, и мощная идея. Как структура данных, она не даёт почти ничего асимптотически нового по сравнению с BST. Основные плюсы дерева отрезков – малые константы времени и памяти, а также простота реализации.

Поэтому наш рассказ начнётся с самой эффективной реализации дерева отрезков – “снизу”.

14.1. Общие слова

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

$$\text{root} = 1, \text{parent}[v] = v / 2, \text{leftSon}[v] = 2 * v, \text{rightSon}[v] = 2 * v + 1.$$

Если вершине v соответствует отрезок $[vl, vr]$ ², её детям будут соответствовать 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 функция 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.

14.2. Дерево отрезков с операциями снизу

```

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

Лм 14.2.1. Время работы `get` на отрезке $[l, r]$ равно $\mathcal{O}(1 + \log(r - l + 1))$

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

²Можно писать на полуинтервалах. Я сам пишу и рекомендую именно полуинтервалы.

³Если vl и vr могут быть отрицательными или больше $\frac{1}{2}\text{INT_MAX}$, вычислять vm следует, как $vl+(vr-vl)/2$.

```

1 int change(int i, int x) {
2   t[i += n] = x; // обновили значение в листе
3   // пересчитали значения во всех ячейках на пути до корня (все отрезки, содержащие i)
4   for (i /= 2; i >= 1; i /= 2)
5     t[i] = min(t[2 * i], t[2 * i + 1]);
6 }

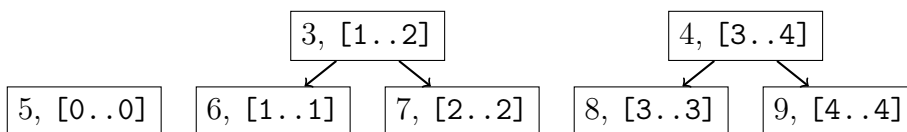
```

Время работы – $\mathcal{O}(\log n)$. И снова без рекурсии.

Lm 14.2.2. `get` и `change` корректно решают задачу “минимум на меняющемся массиве”

Доказательство. 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, он хранится в ячейках $[l_0, r_0) = [n, 2n)$. Из слоя $[l_i, r_i)$ получаем слой отцов $[l_{i+1}, r_{i+1}) = [\lfloor \frac{l_i}{2}, \lfloor \frac{r_i}{2} \rfloor)$. По индукции получаем $r_i \leq 2l_i$ и $r_i \leq l_{i+1}$. ■

Дерево отрезков для $n = 5$:



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

14.3. 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. 14.1).

Все функции работы с деревом, включая построение, – рекурсивные функции спуска. Пример:

```

1 int getMin( int v, int vl, int vr, int l, int r ) {
2   if (vr < l || r < vl) return INT_MAX; // не пересекаются
3   if (l <= vl && vr <= r) return t[v]; // вершина целиком внутри запроса
4   int vm = (vl + vr) / 2;
5   int fl = getMin(2 * v, vl, vm, l, r);
6   int fr = getMin(2 * v + 1, vm + 1, vr, l, r);
7   return min(fl, fr);
8 }
9 int result = getMin(1, 0, n - 1, l, r); // 1 = root

```

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

Операцию `change` можно сделать рекурсивно сверху.

А можно округлить n вверх до 2^k и сделать `get` сверху, `change` снизу. Правда преимуществ нет...

Чтобы пользоваться деревом “сверху” осталось его построить, а в построении – главное выделить массив нужной длины. При $n = 2^k$ получается полное бинарное дерево из $2^{k+1} = 2n$ вершин. Если не округлять n до 2^k , все индексы лишь уменьшатся.

Lm 14.3.1. $4n$ ячеек достаточно, более точно выделить $2^{1+\lceil \log_2 n \rceil}$ ячеек.

Теперь оценим время работы запроса `getMin`.

Lm 14.3.2. `getMin` посетит не более $4 \log n$ вершин дерева.

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

Lm 14.3.3. `getMin` разбивает любой отрезок $[l, r]$ на не более чем $2 \log n$ вершин дерева отрезков.

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

Последняя лемма верна и для реализации “снизу”. По тем же причинам.

Решая задачи, часто удобно думать про дерево отрезков так:

*Мы даём дереву отрезков $[l, r]$, а оно разбивает отрезок $[l, r]$
на $\leq 2 \log n$ вершин, для которых уже посчитана полезная функция.*

Минусы по сравнению с реализацией “снизу”:

- (a) Памяти нужно $4n$ вместо $2n$.
- (b) `get` почти всегда за $\mathcal{O}(\log n)$, даже для отрезков длины $\mathcal{O}(1)$.
- (c) Из-за рекурсии больше константа.

Зато есть много плюсов, главный из них: также, как и в BST, можно делать “модификацию на отрезке”. “Все элементы на отрезке $[l, r]$ увеличить на x ”. “Всем элементам присвоить x ”.

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

```

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

```

Ещё некоторые плюсы реализации сверху:

- (a) Дерево отрезков сверху – реально дерево!
- (b) Дерево отрезков сверху можно сделать персистентным.
- (c) Дерево отрезков сверху можно сделать динамическим (следующий раздел).

14.4. (*) Хаки для памяти к дереву отрезков сверху

• Проблема

Если $n = 2^k$ – проблем нет. Памяти $2n$.

Проблема, это, например, $n = 2^k + 2$, тогда дети корня имеют размер $2^{k-1} + 1 \Rightarrow$ глубину $k \Rightarrow$ правый из детей будет использовать ячейку $2^{k+1} + 2^k \approx 3n$.

Бывает ещё хуже. Пусть $n = 2^k + 4 \Rightarrow$ дети $2^{k-1} + 2 \Rightarrow$ внуки $2^{k-2} + 1 \Rightarrow$ нужна ячейка для правого внука $2^{k+1} + 2^k + 2^{k-1} \approx 3.5n$.

Lm 14.4.1. Обычная реализация сверху при выборе $v_m = (v_l + v_r) / 2$ и с округлением вверх, и с округлением вниз может привести к использованию ячеек с номером $4n - \Theta(\sqrt{n})$.

Доказательство. Рассмотрим $n = 2^k + 2^i \Rightarrow$ на нижнем уровне используем ячейку $2^{k+1} + 2^k + \dots + 2^{k-i+1} = 2^{k+2} - 2^{k-i+1}$. Возьмём $i = k/2$, получим $n \rightarrow 4n - \Theta(\sqrt{n})$. ■

• Решение

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 14.4.2. Метод деления $w \rightarrow \max(2^k < w \text{ и } w - 2^k)$ гарантирует $v < 3n$.

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

14.5. Динамическое дерево отрезков и сжатие координат

Пусть наш массив длины $M = 10^{18}$ и изначально заполнен нулями.

Есть два способа реализовать на таком массиве дерево отрезков.

Первый способ элегантен, 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 14.5.1. Первый способ работает и для версии снизу, и для версии сверху.

Второй способ предлагает от хранения в массиве перейти к ссылочной структуре:

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

Вершина дерева такая же, как и в BST. Вершины можно создавать лениво.

Изначально всё дерево отрезков состоит из `Node* root = NULL`.

Все запросы спускаются сверху вниз и, если попадают в `NULL`, создают на его месте `Node`.

Lm 14.5.2. Время работы любого запроса $\mathcal{O}(\log M)$.

Lm 14.5.3. После k запросов создано $\mathcal{O}(\min(M, k \log M))$ `Node`-ов.

Lm 14.5.4. После k запросов создано $\mathcal{O}(k \cdot \max(1, \log M - \log k))$ `Node`-ов.

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

• Сжатие координат

Динамическое дерево отрезков засчёт большей глубины и ссылочной структуры медленней обычного. Использует больше памяти \Rightarrow реже кэшируется. Поэтому, когда есть возможность, для решения исходной задачи применяют не его, а *сжатие координат*.

Задача: в offline обарботать n запросов “изменение в точке i_j ” и “сумма на отрезке $[l_j, r_j]$ ” над массивом длины M .

Сжатие координат.

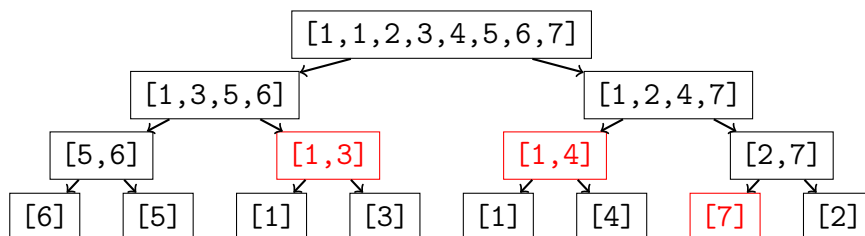
Offline \Rightarrow все запросы известны заранее, на $[0, M)$ не более $3n$ интересных точек: i_j, l_j, r_j .

Давайте, сложим их в массив, отсортируем и заменим на позиции в этом массиве.

То есть, перенумеруем числами из $[0, 3n)$. Свели исходную к задаче на массиве длины $3n$.

14.6. 2D-деревья

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



Как мы помним (14.3.3), дерево отрезков разбивает любой отрезок на $\mathcal{O}(\log n)$ вершин дерева отрезков. Красным выделены вершины, на которые распадётся отрезок $[2, 7)$.

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

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

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

Теорема 14.6.1. 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 $X_1 \leq x_i \leq X_2$ можно превратить в равносильное $L \leq i \leq R$. ■

Обе задачи решаются деревом отрезков отсортированных массивов за $\mathcal{O}(\log^2 n)$ на запрос⁴.

Решим задачу #1, вторая к ней сводится. Отрезок $[L, R]$ разделится на $\mathcal{O}(\log n)$ вершин, на каждой из них сделаем два бинпоиска, вернём \sum по вершинам “upper_bound(U) - lower_bound(D)”.

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

⁴В разд. 14.9 мы оптимизируем время до $\mathcal{O}(\log n)$

- k -я статистика на отрезке за $\mathcal{O}(\log^3 n)$.

Запрос `get(L,R,k)` – вернуть `sorted(a[L..R])[k]`.

Сделаем бинпоиск по ответу. Внутри бинпоиска нам дают x и нужно узнать сколько чисел на $[L, R]$ не более x , что мы только что научились за $\mathcal{O}(\log^2 n)$. Заметим, что бинпоиск по ответу можно реализовать, как бинпоиск по `sorted(a)`, поэтому бинпоиск сделает $\mathcal{O}(\log n)$ итераций.

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

- 3D-запросы

В вершине дерева отрезков можно хранить вообще всё, что душе угодно. Например, 2D-дерево. Рассмотрим 3D-запрос на плоскости: даны n точек (x_i, y_i, z_i) , нужно отвечать на запросы $\#\{i: X_1 \leq x_i \leq X_2 \wedge Y_1 \leq y_i \leq Y_2 \wedge Z_1 \leq z_i \leq Z_2\}$. 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.

14.7. 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

- Прямоугольник начался: сделаем `count[y1..y2] += 1`;
- Прямоугольник закончился: сделаем `count[y1..y2] -= 1`;
- 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 $\{(x, y): X_1 \leq x \leq X_2 \wedge Y_1 \leq y \leq Y_2\}$ равно разности количеств в областях $\{(x, y): x \leq X_2 \wedge Y_1 \leq y \leq Y_2\}$ и $\{(x, y): x \leq X_1 - 1 \wedge Y_1 \leq y \leq Y_2\}$.

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 $\sum_{y \in [y_1..y_2]} \text{count}[y]$.

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)` обрабатываем 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 14.6.1 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.

14.8. 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`. Общим вершинам соответствует отрезок `[v1..vr]`,

если $(a \rightarrow l \rightarrow \text{sum} - b \rightarrow l \rightarrow \text{sum} \geq k)$, есть хотя бы k чисел со значениями $[v_l..v_m]$ и мы в обоих деревьях спустимся налево, иначе мы теперь хотим найти $(k - (a \rightarrow l \rightarrow \text{sum} - b \rightarrow l \rightarrow \text{sum}))$ -е число и в обоих деревьях спустимся направо.

14.9. (*) Fractional Cascading

У нас есть дерево отрезков сортированных массивов. Отвечать на запрос можно сверху или снизу. На скорость это не влияет, т.к. и там, и там мы делаем одни и те же $\mathcal{O}(\log n)$ бинпоисков.

Хотим отвечать на запрос $\#\{i: L \leq i \leq R \wedge a[i] < x\}$.

Будем отвечать на запрос сверху, но бинпоиск сделаем только в корне $root = 1$.

Вместо того, чтобы делать бинпоиски в детях, воспользуемся ответом для отца.

Для этого нам понадобится предподсчёт, который получается лёгкой модификацией merge:

```

1 void build( int n, int *a ) {
2     assert(n & (n - 1) == 0) // n = 2^k
3     vector<vector<int>> t(2 * n), l(2 * n);
4     for (int i = 0; i < n; i++)
5         t[n + i] = vector<int>(1, a[i]); // листья дерева
6     for (int i = n - 1; i > 0; i--) {
7         int A = t[2 * i].size(), B = t[2 * i + 1].size();
8         t[i].resize(A + B), l[i].resize(A + B);
9         for (size_t a = 0, b = 0, c = 0; a < L || b < R; ) { // собственно merge
10            l[i][c] = a; // среди первых c элементов t[i] ровно a ушли в левого сына
11            if (a == L || (b < R && t[2 * i][a] > t[2 * i + 1][b]))
12                t[i][c++] = t[2 * i + 1][b++];
13            else
14                t[i][c++] = t[2 * i][a++];
15        }
16    }
17 }

```

Если бинпоиск в вершине i вернул x , в детях нам вернут $l[i][x]$ и $t[i].size() - l[i][x]$.

Следствие 14.9.1. $\forall k \geq 2$ мы улучшили время k -мерного ортогонального запроса с $\log^k n$ до $\log^{k-1} n$, добавив во внутреннее $2D$ дерево fractional cascading.

На самом деле fractional cascading – более общая идея: за $\mathcal{O}(k + \log n)$ сделать бинпоиск сразу по k сортированным массивам. Мы сейчас решили частный случай этой задачи для $\log n$ массивов специального вида. [wiki](#) даёт общее описание и ссылки. Общая идея, как в skip list – половину элементов “нижнего списка” толкать вверх в следующий список.

14.10. (*) КД-дерево

Даны n точек на плоскости, каждая задаётся тройкой (x_i, y_i, w_i) – координатами и весом. Построим за $\mathcal{O}(n \log n)$ структуру данных, которая будет занимать $\mathcal{O}(n)$ памяти и будет уметь для точек в прямоугольной области $(lx_j \leq x_i \leq rx_j, ly_j \leq y_i \leq ry_j)$ за $\mathcal{O}(\sqrt{n})$ делать все те же операции, что дерево отрезков на отрезке. Добавлять новые точки можно будет корневой по запросам. Удалять, если нет необратимых запросов типа «min на прямоугольнике», также.

Структура: бинарное дерево. В корне прямоугольник $[-\infty, +\infty] \times [-\infty, +\infty]$. На чётном уровне делим вертикальной прямой n точек на $\lfloor \frac{n}{2} \rfloor$ и $\lceil \frac{n}{2} \rceil$, на нечётном делим горизонтальной прямой.

Построение за $\mathcal{O}(n \log n)$. Отсортировали и по x , и по y , передали в рекурсию два порядка – и по x , и по y . Время работы кроме исходной сортировки $T(n) = \mathcal{O}(n) + 2T(n/2)$.

Запрос за $\mathcal{O}(\sqrt{n})$. Как и в дереве отрезков 3 варианта – прямоугольник запрос и прямоугольник вершина-дерева могут не пересекаться, вкладываться, нетривиально пересекаться. Время работы $2^{\frac{1}{2} \log n}$, так как на каждом втором уровне рекурсии мы не будем ветвиться.

Лекция #15: LCA & RMQ

14 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 $\mathcal{O}(g(n))$.

15.1. RMQ & Sparse table

Def 15.1.1. *RMQ = Range Minimum Query = запросы минимумов на отрезке.*

Задачу RMQ можно решать на не меняющемся массиве (static) и на массиве, поддерживающем изменения в точке (dynamic). Запросы минимума на отрезке будем обозначать “get”, изменение в точке – “change”. Построение структуры – “build”.

Мы уже умеем решать задачу RMQ несколькими способами:

1. Дерево отрезков: build за $\mathcal{O}(n)$, get за $\mathcal{O}(\log n)$, change за $\mathcal{O}(\log n)$
2. Центроидная декомпозиция: build за $\mathcal{O}(n \log n)$, get за $\mathcal{O}(LCA)$, static.
3. Корневая: build за $\mathcal{O}(n)$, get за $\mathcal{O}(\sqrt{n})$, change за $\mathcal{O}(1)$.
4. Корневая: build за $\mathcal{O}(n)$, get за $\mathcal{O}(1)$, change за $\mathcal{O}(\sqrt{n})$.

Lm 15.1.2. \nexists структуры данных, поддерживающей build за $\mathcal{O}(n)$, change и get за $o(\log n)$.

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

А вот static версию задачи (только get-запросы) мы скоро решим за время $\langle n, 1 \rangle$.

• Sparse Table

Пусть $f[k, i]$ – минимум на отрезке $[i, i+2^k)$. Массив $f[]$ можно предподсчитать за $n \log n$: $f[0]$ – исходный массив, $f[k, i] = \min(f[k-1, i], f[k-1, i+2^{k-1}])$.

Теперь, get на $[l, r) = \min(f[k, l], f[k, r-2^k])$, где $2^k \leq r - l < 2^{k+1}$.

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++

7	5	2	4	6	1	8	4	3	7	8	3	10	5	10	6
$b_1=2$		$b_2=1$		$b_3=3$			$b_4=5$								

Разобьём исходный массив a на куски длины $\log n$, минимум на i -м куске обозначим b_i .

\forall отрезок a , который содержит границу двух кусков, разбивается на отрезок b и два “хвоста”.

Минимум на хвосте – это минимум на префиксе или суффиксе одного куска, все такие частичные минимумы предподсчитаем за $\mathcal{O}(n)$. Чтобы, находить минимум на отрезке массива b , построим на b Sparse Table, который весит $\frac{n}{\log n} \log \frac{n}{\log n} \leq n$. Получили $\langle n, 1 \rangle$ решение.

Осталось решить для отрезков, попадающих целиком в один из кусков.

Давайте на каждом куске построим структуру данных для решения RMQ.

Наивный цикл	$\langle n, \log n \rangle$	константа сверхмала
Sparse Table	$\langle n \log \log n, 1 \rangle$	$\frac{n}{\log n} \cdot (\log n \log \log n)$
Дерево отрезков	$\langle n, \log \log n \rangle$	
Рекурсивно построить себя	$\langle n \log^* n, \log^* n \rangle$	$T(n) = 1 + T(\log n), M(n) = n + (n/\log n)(\log n + \dots)$

15.2. 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];
```

Def 15.2.1. $LCA(a, b)$ – общий предок вершин a и b максимальной глубины.

LCA = least common ancestor = наименьший общий предок.

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

Соптимизируем эту идею – научимся $\forall v, k$ из вершины v прыгать сразу на 2^k вверх.

$up[k, v] = up[k-1, up[k-1, v]]$ – прыжок на 2^k равен двум прыжкам 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 – уравнивать глубины и прыгать вверх:

```
1 int K = ⌈log2N⌉, up[K][N];
2 int LCA(int a, int b) {
3   if (depth[a] < depth[b]) swap(a, b);
4   a = jump(a, depth[a] - depth[b]);
5   for (int k = K - 1; k >= 0; k--)
6     if (up[k][a] != up[k][b])
7       a = up[k][a], b = up[k][b];
8   return a == b ? a : up[0][a];
9 }
```

Здесь `jump(v, d)` за $\mathcal{O}(\log n)$ прыгает вверх из v на d , для этого d нужно представить, как сумму степеней двойки. Время и память предподсчёта – $\Theta(n \log n)$, время поиска LCA – $\Theta(\log n)$.

Можно уменьшить константу времени работы, используя `isAncestor`:

```
1 int LCA(int a, int b) {
2   for (int k = K - 1; k >= 0; k--)
3     if (!isAncestor(up[k][a], b))
4       a = up[k][a];
5   return isAncestor(a, b) ? a : up[0][a];
6 }
```

15.3. RMQ±1 за $\langle n, 1 \rangle$

Def 15.3.1. Говорят, что массив обладает ± 1 свойством, если $\forall i |a_i - a_{i+1}| = 1$

Наша цель – решить RMQ на ± 1 массиве.

Подкрутим уже имеющуюся у нас идею из Sparse Table ++.

“Разобьём исходный массив a на куски длины $k = \frac{1}{2} \log n$, минимум на i -м куске обозначим b_i .”

$$\min(a_1, a_2, \dots, a_k) = a_1 + \min(0, a_2 - a_1, a_3 - a_1, \dots, a_k - a_1) = a_1 + X.$$

Поскольку $\forall i |a_{i+1} - a_i| = 1$, X – минимум на одной из 2^{k-1} последовательностей.

$2^k = 2^{\log n / 2} = \sqrt{n} \Rightarrow$ можно за $o(n)$ предподсчитать ответы для всех последовательностей.

15.4. LCA \rightarrow RMQ ± 1 и Эйлеров обход

Напомним, эйлеров обход графа – цикл, проходящий по каждому ребру ровно один раз.

Если у дерева каждое ребро заменить на два ориентированных, мы получим эйлеров оргграф.

Чтобы построить эйлеров обход дерева, пишем `dfs(v)`.

```

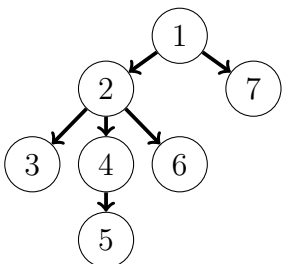
1 void dfs(int v) {
2   for (Edge e : graph[v]) { // пусть мы храним только рёбра, ориентированные вниз
3     answer.push_back(edge(v,x));
4     dfs(x);
5     answer.push_back(edge(x,v));
6   }
7 }
```

Такой обход назовём обычным или “эйлеровым обхода дерева первого типа”.

Иногда имеет смысл хранить другую информацию после обхода:

```

1 void dfs2(int v) {
2   index[v] = answer.size(); // сохранили для вершины v любое её вхождение в answer
3   answer.push_back(v);
4   for (Edge e : graph[v]) {
5     dfs2(x);
6     answer.push_back(v); // поднимаемся, проходим через вершину v
7   }
8 }
9 void dfs3(int v) {
10  L[v] = answer.size(); // аналог времени входа
11  answer.push_back(v); // просто сохраняем порядок обхода вершин dfs-ом
12  for (Edge e : graph[v])
13    dfs3(x);
14  R[v] = answer.size(); // аналог времени выхода
15 }
```



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

Второй обход по сути тоже эйлеров.

Мы сохраняем не рёбра, по которым проходим, а вершины, в которых оказываемся при обходе.

Третий обход уже слабо напоминает эйлеров, но мы в контексте задач, где нужно выбирать между 2-м и 3-м обходами, будем иногда называть его эйлеровым.

Зачем нужны 2-й и 3-й обходы будет понятно уже сейчас, 1-й нам пригодится для ЕТТ.

Lm 15.4.1. После третьего обхода отрезок обхода $[L_v, R_v)$ задаёт ровно поддерево вершины v .

Следствие 15.4.2. 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 15.4.3. Массив высот $h[i] = \text{height}[\text{answer}[i]]$ второго обхода обладает ± 1 свойством.

Lm 15.4.4. $\text{LCA}(a, b) = \text{answer}[h.\text{RMQ}[\text{index}[a], \text{index}[b]]]$

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 15.4.5. 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 15.4.6. На практике популярен способ решения LCA: LCA \rightarrow RMQ, а RMQ решим через Sparse Table. Это $\langle n \log n, 1 \rangle$, причём у $\mathcal{O}(1)$ относительно небольшая константа.

15.5. RMQ \rightarrow 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 15.5.1. RMQ на $[l, r]$ в исходном массиве равно $\text{LCA}(l, r)$ 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 $[L_i, R_i] \Rightarrow a_i = \min_{L_i \leq j \leq R_i} a_j$ и $L_i \leq l \leq i \leq r \leq R_i \Rightarrow a_i \geq \min_{l \leq j \leq r} a_j$ и $i \in [l, r]$.

Осталось заметить, что $i = \text{LCA}(l, r)$. ■

Следствие 15.5.2. Мы научились решать статичную версию RMQ за $\langle n, 1 \rangle$. Победа!

15.6. 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”.

15.7. 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 $LCA \rightarrow RMQ \pm 1$, выпишем высоты Эйлера обхода второго типа.

$LA(v, k) = getNext(index[v], height[v] - k)$, где $index$ – позиция в Эйлеровом обходе, а $getNext(i, x)$ возвращает ближайший справа от i элемент $\leq x$.

Мы умеем отвечать на $getNext(i, x)$ за $\langle n, \log n \rangle$ одномерным ДО снизу или сверху.

15.8. Euler-Tour-Tree

Задача: придумать структуру данных для хранения графа, поддерживающую операции

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

При этом в каждый момент времени выполняется условие отсутствия циклов (граф – лес).

По сути мы решаем *Dynamic Connectivity Problem* с дополнительным условием “граф – лес”.

Решение: хранить обычный эйлеров обход дерева (ориентированные рёбра).

Эйлеров обход – массив. Заведём заведём на нём горе. Например, $treap$ по неявному ключу.

```

1 map<pair<int,int>, Node*> node; // по орребру умеем получать вершину treap
2 vector<Node*> anyEdge; // для каждой вершины графа храним любое исходящее ребро
3 bool isConnected(int a, int b) {
4     // взяли у каждой вершины произвольное ребро, проверили, что два ребра живут в одном дереве
5     return getRoot(anyEdge[a]) == getRoot(anyEdge[b]);
6 }
7 void cut(int a, int b) {
8     // по орребру получаем Node*, находим его позицию в эйлеровом обходе
9     Node *node1 = node[make_pair(a, b)];
10    Node *node2 = node[make_pair(b, a)];
11    int i = getPosition(node1), j = getPosition(node2);
12    if (i > j) swap(i, j); // упорядочили (i,j)
13    Node *root = getRoot(node1), *a, *b, *c;
14    Split(root, a, b, i);
15    Split(b, b, c, j - i); // разделили дерево на три части: (a) (i b) (j c)
16    Delete(b, 0), Delete(c, 0); // собственно удаление лишнего ребра
17    Merge(a, c); // в итоге теперь есть два дерева: (a c) и (b)
18 }

```

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 .

```

1 void link(int a, int b) {
2     Node *pa = anyEdge[a], *pb = anyEdge[b];
3     Rotate(getRoot(pa), getPosition(pa)); // Rotate = Split + Merge
4     Rotate(getRoot(pb), getPosition(pb));
5     // теперь первые рёбра эйлеровых обходов - исходящие из a и b соответственно
6     Node *e1 = createEdge(a, b), *e2 = createEdge(b, a);
7     Merge(Merge(getRoot(pa), e1), Merge(getRoot(pb), e2));
8 }

```

Можно реализовать `pair<Node*,int> goUp(Node*)`, которая заменит и `getRoot`, и `getPosition`.

15.9. (*) LA, быстрые решения

15.9.1. (*) Вишкин за $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$

В разд. 15.7 у нас уже построен Эйлеров обход и мы знаем, что $LA(v, k) = getNext(index[v], height[v] - k) = getNext(i, a[i] - k)$.
Осталось научиться вычислять `getNext` за $\mathcal{O}(1)$.

Давайте $\forall i$ предподсчитаем `getNext(i, a[i] - j)` для всех j до $3 \cdot 2^k$, где 2^k – максимальная степень двойки, которая делит i . Суммарный размер предподсчёта = $\sum_k \frac{2n}{2^k} (3 \cdot 2^k) = \Theta(n \log n)$, где $2n$ – длина Эйлерова обхода. Предподсчёт делается через LA-offline за то же время.

Ответ на запрос: выберем $s: 2^s \leq k < 2^{s+1}$.

Поскольку $getNext(i, a[i] - k) \geq i + k \geq i + 2^s$, перейдём к $j = i - (i \bmod 2^s) + 2^s$.

Заметим, что `getNext(j, a[i] - k)` предподсчитан, так как $a[j] - a[i] \leq 2^s \wedge k \leq 2 \cdot 2^s$.

15.9.2. (*) Ladder decomposition + четыре русских

• Longest-Path-Decomposition и $\langle \mathcal{O}(n), \mathcal{O}(\sqrt{n}) \rangle$

Longest-Path-Decomposition – разбиение вершин дерева на пути, путь от вершины продолжается вниз в сторону самого глубокого сына. Внутри одного пути LA считается за $\mathcal{O}(1)$. Иначе будем прыгать вверх по путям. Прыжков $\mathcal{O}(\sqrt{n})$, так как, на k -м подъёме у нас должен быть брать длины $\geq k \Rightarrow$ всего в дереве $1 + \dots + k = \Theta(k^2)$ вершин.

• Ladder-Decomposition и $\langle \mathcal{O}(n), \mathcal{O}(\log n) \rangle$

Возьмём пути из прошлого решения.

Пусть k_i – длина пути, насчитаем продолжение пути вверх на k_i .

Размер и время предподсчёта всё ещё линейны.

Прыгаем вверх. Пусть мы сейчас в вершине v и прошли снизу расстояние $x \Rightarrow$ из v есть путь вниз длины $\geq x \Rightarrow$ вверх тоже $\geq x \Rightarrow$ каждый раз расстояние удваивается.

• Ladder-Decomposition и $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$

Добавим двоичных подъёмов. Отсюда и предподсчёт $\mathcal{O}(n \log n)$. С их помощью, желая прыгнуть на k , за $\mathcal{O}(1)$ прыгнем на 2^s : $2^s \leq k < 2^{s+1}$. Из вершины, в которой мы оказались, путь ladder decomposition позволяет подняться за $\mathcal{O}(1)$ на оставшиеся $k - 2^s \leq 2^s$.

• Эйлеров обход, четыре русских и $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$

Нужно насчитать двоичные подъёмы не ото всех вершин, а лишь от $\Theta(\frac{n}{\log n})$ каких-то...

Предподсчитывать динамикой $f[k, v] = f[k-1, f[k-1, v]]$ не получится \Rightarrow используем LA-offline.

Берём ± 1 Эйлеров обход, выбираем k и от каждой k -й вершины считаем подъёмы.

Чтобы ответить на запрос $LA(v, k)$ берём $i = pos[v]$ в Эйлеровом обходе, $j = i - (i \bmod k) + k$.

Если $h[j] \leq h[i] + k$, то отвечаем за $\mathcal{O}(1)$ от j , иначе ответ лежит на $(j - k, j]$. Ответ зависит только от битовой строки длины k , чисел $j - i$ и k . Итого $2^k k^2$ различных задач, ответы на которые мы предподсчитаем. Пусть $k = \frac{1}{2} \log n$.

Лекция #16: HLD & LC

21 мая

16.1. Heavy Light Decomposition

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

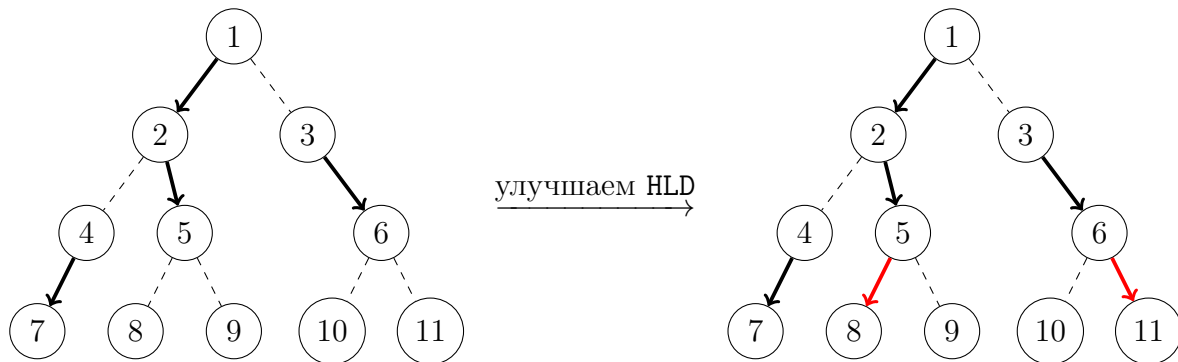
В частном случае “дерево = путь” задача уже решена деревом отрезков. Обобщим решение для произвольного дерева: разобьём вершины дерева на пути, на каждом построим дерево отрезков. Теперь любой путь разбивается на какое-то количество отрезков, на которых функцию можно посчитать деревом отрезков. Осталось 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 16.1.1. *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 16.1.2. *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) { // внутри пути нумерация снизу вверх
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]

```

Лм 16.1.3. 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 $a \rightarrow b$ на два вертикальных: $a \rightarrow \text{LCA} \rightarrow b$. Подсчёт функции на пути $a \rightarrow \text{LCA}$: поднимаемся из a , если $\text{path}[a] = \text{path}[\text{LCA}]$, можем посчитать функцию за одно обращение к дереву отрезков: $\text{tree}[\text{path}[a]].\text{get}(\text{pos}[a], \text{pos}[\text{LCA}])$. Иначе поднимаемся до $\text{top}[\text{path}[a]]$ и переходим в $\text{parent}[\text{top}[\text{path}[a]]]$.

Теорема 16.1.4. Время подсчёта функции на пути $\mathcal{O}(\log^2 n)$.

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

Теорема 16.1.5. Время изменения веса одной вершины $\mathcal{O}(\log n)$.

Доказательство. Вершина лежит ровно в одном дереве отрезке. ■

Замечание 16.1.6. Аналогично можно считать функции на рёбрах.

Например, используем биекцию “вершина \leftrightarrow ребро в предка”.

Замечание 16.1.7. Воспользуемся функцией `isAncestor`, тогда можно обойтись без подсчёта LCA: прыгать от вершины a вверх, пока не попадём в предка b , и затем от b вверх, пока не попадём в предка a . В конце a и b лежат на одном пути, учтём эту часть тоже. Более того, таким образом с помощью HLD можно найти LCA за время $\mathcal{O}(\log n)$, используя всего $\mathcal{O}(n)$ предподсчёта.

• Другие применения.

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

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

• Функция на поддереве.

Сгенерируем пути чуть иначе.

```

1 int bound = 0; // левая граница текущего пути
2 int k = 0, a[n]; // Эйлеров обход дерева
3 void build_HLD( int v ) {
4     left[v] = bound, pos[v] = k, a[k++] = v;
5     if (children[v].empty())
6         return;
7     int i = (x in children[v] : size[x] = max);
8     buildPos(i, v); // сперва идём в тяжёлого сына, продолжаем текущий путь
9     for (int x : c[v])
10        if (x != i)
11            bound = k, build_HLD(x); // сдвинуть границу - начать новый путь
12 }
```

$\text{pos}[v]$, $\text{left}[v]$ – позиции вершины и верхушки её пути в Эйлеровом обходе.

Получается, что каждый путь HLD – отрезок Эйлерова обхода $a[]$. Из этого два вывода:

- Можно параллельно применять массовые операции и на путях, и на поддеревьях.
- Можно хранить одно дерево отрезков на массиве $a[]$, содержащее сразу все пути.

16.2. 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 ЕТТ 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 горе (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]`.

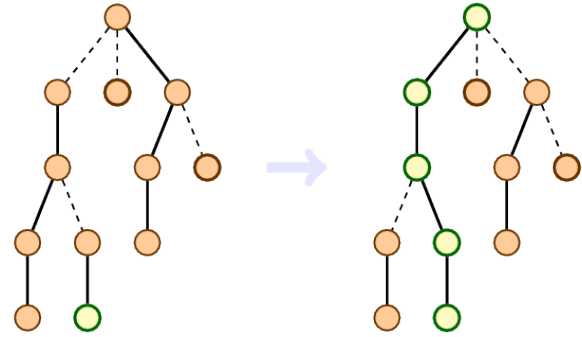
Пусть в `treap` есть ссылки на отцов \Rightarrow от `p[v]` можно за $\mathcal{O}(\log n)$ подняться до корня её `treap` и параллельно насчитать “число вершин в `treap` левее `p[v]`”, то есть, позицию v в её пути.

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

Def 16.2.1. $\text{Expose}(v)$ – операция, меняющая декомпозиция дерева на пути.

Она все вершины на пути от v до корня объединяет в один большой путь.

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



- $\text{MakeRoot}(v)$

Роре умеет делать reverse , для этого достаточно в каждой вершине treap хранить отложенную операцию isReversed . Если после $\text{Expose}(v)$ отрезать часть пути под v и сделать “ $\text{GetRoot}(p[v]).\text{isReversed} \hat{=} 1$ ”, вершина v станет новым корнем дерева.

Оценим время работы новой операции: $T(\text{MakeRoot}(v)) = T(\text{Expose}(v)) + \mathcal{O}(\log n)$.

- $\text{getMax}(a,b), \text{link}(a,b), \text{cut}(a,b), \text{isConnected}(a,b)$

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 16.2.2. Суммарное время m операций $\text{Expose}/\text{MakeRoot}$ – $\mathcal{O}(n + m \log n)$.

Доказательство. Потенциал φ = минус “число тяжёлых рёбер, покрытых путями”.

$\varphi_0 = 0, \varphi \geq -n \Rightarrow \sum t_i = \sum a_i + (\varphi_0 - \varphi_m) \leq n + \sum a_i$. Осталось оценить a_i .

Число лёгких рёбер на пути будем обозначать “Л”, число тяжёлых “Т”.

Expose : $a_i = t_i + \Delta\varphi \leq t_i - T + L \leq k - (k - \log n) + \log n = \mathcal{O}(\log n)$.

$\text{MakeRoot} = \text{Expose} + \text{Reverse}$. Reverse меняет тяжёлость только рёбрам на пути (v, root) . И до reverse , и после лёгких на пути не более $\log n \Rightarrow \Delta\varphi \leq \log n \Rightarrow a_i = t_i + \Delta\varphi = \mathcal{O}(\log n)$. ■

Замечание 16.2.3. На практике мы также оценим собственно Link/Cut и добавим splay -дерево.

16.3. 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: $n \rightarrow \frac{n}{8}$. 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 $\text{MST}(E)$.
3. Переберём рёбра из $B = E \setminus A$, оставим из них $U \subseteq B$ – те рёбра, что могут улучшить T .
Рёбро (a, b) может улучшить T , если a и b не связаны в T или максимум на пути в T между a и b больше веса ребра. Рёбра из $B \setminus U$ точно не входят в $\text{MST}(E)$.
4. $\text{MST}(E) \subseteq T \cup U$. Сделаем рекурсивный вызов от $F(\frac{n}{8}, |T| + |U|)$.

Оценим общее время работы: $|T| \leq \frac{n}{8} - 1$. Скоро мы покажем, что матожидание $|U| \leq \frac{n}{8} - 1$.
Суммарное время работы $F(n, m) \leq (n + m) + M(\frac{n}{8}, \frac{m}{2}) + F(\frac{n}{8}, \frac{m}{2}) + F(\frac{n}{8}, \frac{n}{4})$, где $M(n, m)$ – время поиска в offline минимумов на m путях дерева из n вершин. Оказывается $M(n, m) \leq n + m$.
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 16.3.1. $A_1 = A \setminus T$ и $B_1 = B \setminus U$ точно не лежат в $\text{MST}(E)$

Доказательство. Представим себя Краскалом.

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

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

• Random Sampling Lemma.

Lm 16.3.2. 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 $E[|U|] \leq (n - 1)(\frac{1}{p} - 1)$. ■

16.4. (*) RMQ Offline

У нас есть алгоритм Тарьяна поиска LCA в Offline (dfs по дереву + СНМ).

Мы умеем сводить RMQ-offline к LCA-offline построением декартова дерева.

Если две эти идеи объединить в одно целое, получится удивительно простой алгоритм:

```

1 void solve(int m, Query *q, int n, int *a) {
2     vector<int> ids(n); // для каждой правой границы храним номера запросов
3     for (int i = 0; i < m; i++)
4         ids[q[i].right].push_back(i);
5     DSU dsu(n); // инициализация СНМ: каждый элемент - самостоятельное множество
6     stack<int> mins;
7     for (int r = 0; r < n; r++) {
8         while (mins.size() && a[mins.top()] >= a[r])
9             dsu.parent[mins.pop()] = r; // минимум достигается в r, объединим отрезки
10        mins.push(r);
11        for (int i : ids[r])
12            q[i].result = dsu.get(q[i].left);
13    }
14 }
```

В описанном выше алгоритме можно рассмотреть построение декартова дерева 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: $a[m_1] \leq a[m_2] \leq a[m_3] \leq \dots$

Когда мы расширяем префикс: $r \rightarrow r+1$, стек минимумом обновляется, отрезки объединяются.