

# Algorithms and Data Structures

---

## Algorithms and Data Structures

22-09-07 : Dynamic programming

*Task 1: Getting POSUDA from a STUDENT*

Approaches to compute dynamically

DP on (sub)sets : TSP

Coloring a graph

22-09-21

Трюки с масками

reverse

Младший бит

Старший бит

bitset

Meet in the middle

Пример на рюкзаке

BST

find, next

add

delete

print

AVL-дерево /todo про теорему

rotate

1. Добавление в  $v.l.l$  или в  $v.r.r$

2. Добавление в  $v.l.r$  или в  $v.r.l$

Код

22-09-28

RBST

*Th. (эквивалентность определений)*

*Th. (матожидание для глубины) \todo*

Декартово дерево

*Th. (связь между Treap и RBST)*

Merge

Split

22-10-05

Улучшения BST

1. Массовые операции

(a) getSum(x1, x2)

(b) assignValue(x1, x2, y)

2. Дерево поиска по неявному ключу \todo

(a) get\_kth\_element(k)

22-10-19

Динамическое дерево отрезков

Способ 1: меняем `vector` на `map`

Способ 2: не массив, а указатели

Способ 3: сжатие координат

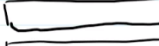

- Двумерные (многомерные) запросы
- Дерево mergeSort
- 22-10-26
  - Sparse table
  - LCA & Двоичные подъемы
    - isAncestor
    - LCA
  - LA (level ancestor)
    - Offline
  - Алгоритм Вишкина
- 22-11-02
  - RMQ  $\pm 1$
  - ФКБ
- 22-11-09
  - MST
  - Th. (лемма о разрезе)
  - Алгоритм Прима
  - Алгоритм Краскала
    - Корректность алгоритма
- 22-11-18
  - DSU
    - DSU на списках
      - Th. (суммарное время работы join)
    - DSU на деревьях
      - Оптимизации
      - Th. (про размер поддеревя)
      - Th. (время работы join и get)

## 22-09-07 : Dynamic programming

### Task 1: Getting POSUDA from a STUDENT

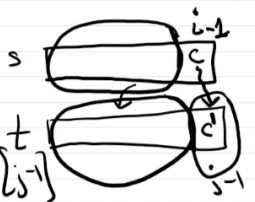
Dynamic Programming

Edit distance

$s$    $\rightarrow$   $t$  

min # of oper.

- 1).  $d[i][j]$  - min # of oper.,  $s[:i] \rightarrow t[:j]$
- 2). answer:  $d[|s|][|t|]$
- 3).  $d[i][j] = \min \begin{cases} d[i-1][j] + 1 \\ d[i][j-1] + 1 \\ d[i-1][j-1] + 1, s[i-1] \neq t[j-1] \\ d[i-1][j-1], s[i-1] = t[j-1] \end{cases}$



From a word  $S$  we want to get a word  $T$  by inserting, removing or replacing characters.

$d[i][j]$  == min amount of operations to change  $S[1..i]$  into  $T[1..j]$

**Base:**

`d[0][j] = j` (just insert letters into S and get T)

`d[i][0] = i`

**Transition:**

$$d[i][j] = \min \begin{cases} d[i-1][j] + 1 (\text{insert a char.}) \\ d[i][j-1] + 1 (\text{remove a char.}) \\ d[i-1][j-1] + 1 (\text{replace a char.}) \end{cases}$$

**Approaches to compute dynamically**1. Recursion + Memorization *aka* Lazy DP

```

1  int f(i, j) {
2      if (dp[i][j] is computed) // <----- that is where memorization
    is used
3          return dp[i][j];
4      return f(i - 1, j) + 1;
5      ...
6  }
```

## 2. Using a for-cycle

```

1  for i = 1..n {
2      for j = 1..m // provided that dp[1..i-1][1..j-1] is computed
    correctly
3      {
4          dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);
5          dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);
6          ...
7      }
8  }
```

**DP on (sub)sets : TSP**

TSP -- travelling Salesman Problem -- find a path visiting all vertexes only once.

Weight of the path should be minimal.

$d[S, v]$  -- min weight of a path which ends in  $v$  and visits vertices that are in subset  $S$  once.

Answer is in  $d[\{1, \dots, n\}, v]$

**Base:**

- $\forall v: S = \{v\} \quad d[S, v] = 0 \quad d[S, u] = +\infty \text{ for } u \neq v$
- $d[S, v] = +\infty \text{ for } v \notin S$

**Transition:**

$$d[S, v] = \min_{u \in S} \{ d[S \setminus \{v\}, u] + w_{uv} \}$$

**Time:**  $O(2^n n^2)$

**Memory:**  $O(2^n n)$

```
1 // Base cases
2 for (v = 0..n-1) {
3     S = 1 << v;          // S = {v}
4     for (u = 0..n-1) {
5         if (v == u)
6             d[S][u] = 0;
7         else
8             d[S][u] = +inf;
9     }
10 }
11
12 // переход
13 for ( S = 0 .. (1<<n)-1 ) {
14     for (v = 0..n-1)
15     {
16         if (S >> v mod 2 == 0) {    // check if v \notin S
17             d[S][v] = +inf;
18             continue;
19         }
20         for (uv \in E) {            // looking at edges between v and other
vertices
21             d[S][v] = min(d[S][v], d[S - (1 << v)][u] + weight[u][v])    //
S \ {v}
22         }
23     }
24 }
```

## Coloring a graph

The goal is to minimize amount of different colors. The graph is *undirected*.

$V = \{0, \dots, n-1\}$  -- all vertices.  $S \subseteq V$

$d[S] == \min \#$  of colors used to color subgraph on vertices from  $S$

Answer is in  $d[V]$

**Base:**

- $d[\text{\textbackslash varnothing}] = 0$
- $d[i] = 1 \text{ \textbackslash forall } i = 0..n-1$

**Transition:**

$$d[S] = \min_{T \subseteq S} \{ d[T] + 1 \} \quad (T \text{ is an independent set. '+1' for coloring it})$$

```
1 for (S = 0..2^n-1) {  
2     for (t = S..1)  
3 }
```

## 22-09-21

---

## Трюки с масками

### reverse

16-битные числа.

Рассмотрим число  $n$ .

- откусим у него последний бит (  $n \gg 1$  ). В начале числа появился ведущий ноль, так как кол-во битов фиксированное
- полученную штуку перевернем (  $rev[n \gg 1]$  )
- уберем ведущий ноль, который теперь оказался в конце числа (  $rev[n \gg 1] \gg 1$  )
- в начало результата добавим чиселко, которое мы откусили в первом пункте (  $| ((n \& 1) \ll 15)$  )

```
1 for n = 1..(1 << 16) - 1 :  
2     rev[n] = (rev[n >> 1] >> 1) | ((n & 1) << 15);
```

## Младший бит

```
1 | n & -(n - 1)
```

## Старший бит

Предподсчитаем старший бит для чисел от 0 до  $(1 \ll 16) - 1$

```
1 msb = 0;
2 for (int m = 1; m < (1 << n); ++m) {
3     if (m == (1 << (msb + 1))) // случай, если старший бит устарел
4         ++msb;                // (например, к числу 11..1 прибавили 1)
5     ....
6 }
```

## bitset

`bitset<N> b`

- фиксированной длины
- занимает  $\frac{N}{w}$  байт, где  $w = 8$  -- машинное слово

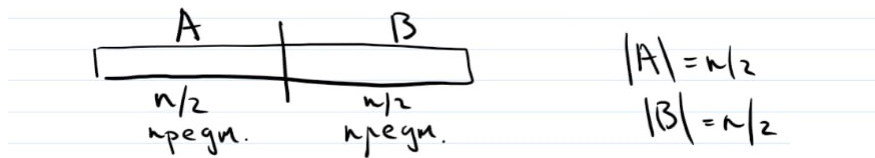
```
1 #include <bitset>
2 #include <iostream>
3
4 int main() {
5     std::bitset<10> bs1;
6     std::bitset<10> bs2("11010");
7     std::bitset<10> bs3(103);
8
9     int x = bs2[3];
10
11
12     bs1[4] = 1; // 0000010000
13     bs1.set(4, 0); // 0000000000
14     bs1.set(3); // 0000001000
15     bs1.set(); // 1111111111
16
17     bs1.reset(1); // 1111111101
18     bs1.reset(); // 0000000000
19
20     bs1.flip(3); // 0000001000
21     bs1.flip(); // 1111110111
22 }
```

```
1 #include <bitset>
2 #include <iostream>
3
4 int main() {
5     std::bitset<10> bs; // 0000000000
6
7     std::cout << bs.any(); // 0
8
9     bs.flip(0); // 0000000001
10    std::cout << bs.any(); // 1
11
12
13    std::cout << bs.none(); // 0
14    bs.reset(); // 0000000000
15    std::cout << bs.none(); // 1
16
17
18    bs.set(3);
19    bs.set(5); // 0000101000
20    std::cout << bs.count(); // 2
21
22 }
```

# Meet in the middle

## Пример на рюкзаке

Решение за  $O(2^{\frac{n}{2}}n)$



1.  $O(2^{n/2})$ : Выписываем всевозможные наборы из  $A$ .  $a \in A$ .
2.  $O(2^{n/2} \frac{n}{2})$ : Отсортируем пары  $\{weight_a, cost_a\}$  по возрастанию весов
3.  $O(2^{n/2})$ : Посчитаем максимумы  $cost_a$  на префиксах: `maxC[i+1] = std::max(maxC[i], cost[i+1])`
4. Перебираем всевозможные наборы из  $B$

## BST

```
1 struct Node {
2     Node* left;
3     Node* right;
4     Node* parent;
5     int x;
6     auto it;          // указатель на место в списке
7 }
8
9 using pNode = Node*;
```

## find, next

```
1 bool find(Node* v, int x) {    // O(n)
2     if (v == nullptr)
3         return nullptr;
4     if (!v->left && !v->right)
5         return false;
6     if (v->x == x)
7         return true;
8     if (v->x < x)
9         find(v->right, x);
10    find(v->left, x);
11 }
12
13 Node* next(Node* v) {        // O(n)
```

```

14     if (v->right) {
15         v = v->right;
16         while (v->left)
17             v = v->left;
18     }
19     return v;
20 }

```

## add

Доходим до момента когда у вершинки нет левого ребёнка

```

1  Node* add(Node* v, int x) {    // O(n)
2
3      if (!v->left && !v->right)
4          return new Node(x);
5      else if (x < v->x)
6          v->left = add(v->left, x);
7      else if (x > v->x)
8          v->right = add(v->right, x);
9  }

```

## delete

Находим вершинку  $u$ , у которой  $u \rightarrow x > v \rightarrow x$ . У  $u$  нет левого ребёнка.

Перевешиваем  $u$  на место  $v$ , меняем значение указателей, потом удаляем  $v$ .

```

1  void del(Node* v) {    // O(n)
2      Node* u = next(v);
3
4      u->x = v->x;
5      // u->parent->left = v->right; мб не надо, непонятно
6      u->parent = v->parent;
7      u->left = v->left;
8      u->right = v->right;
9
10     v = u;
11 }

```



## print

```
1 void print(Node*v) { // O(n)
2     if (!v->right && !v->left)
3         return;
4     print(v->left);    // выведется в отсорт.порядке:
5     std::cout << v->x; // сначала все <x, потом x, потом >x
6     print(v->right);
7 }
```

**Если равные ключи:**

1.  $x \rightarrow \text{pair}(x, \text{id})$ ;
2. Считать количество  $x$  в структуре
3.  $x < v \rightarrow \text{left}$ ;  $x \geq v \rightarrow \text{right}$

## AVL-дерево /todo про теорему

Сбалансированное BST. Придумано в 1962 году советским математиками Адельсоном-Вельским (AV) и Ландисом (L).

**Условие:**  $|h(v.l) - h(v.r)| \leq 1, \forall v$

**Th.:** глубина AVL-дерева  $O(\log n)$

**Док-во:**

$S_h$  -- min число вершин высоты  $h$  (здесь  $h$  = кол-во вершин для данной высоты)

$S_0 = 0, S_1 = 1, S_2 = 2, S_3 = 4$  и т.д.

Докажем, что  $S_h = S_{h-1} + S_{h-2} + 1$

Вершина имеет высоту  $h$ , тогда когда максимальное из высот детей равно  $h - 1$

Получается, высота растёт экспоненциально.

Рассмотрим дерево на  $n$  вершинах.  $h$  -- высота.

$$n \geq S_n \geq c \cdot \phi^h$$

$$\log_{\phi} n \geq \log_{\phi} c + h$$

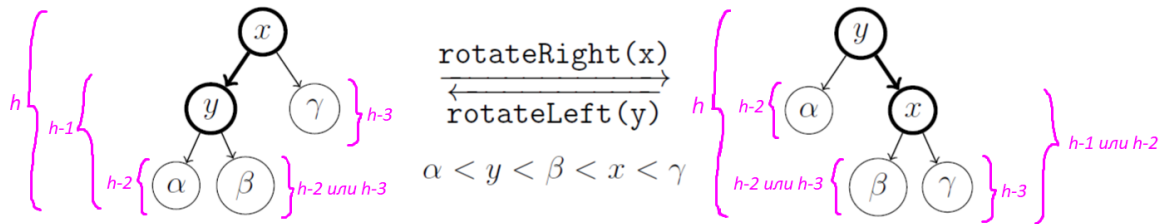
$$h \leq \log_{\phi} n - \log_{\phi} c = O(\log n)$$

## rotate

### 1. Добавление в $v.l.l$ или в $v.r.r$

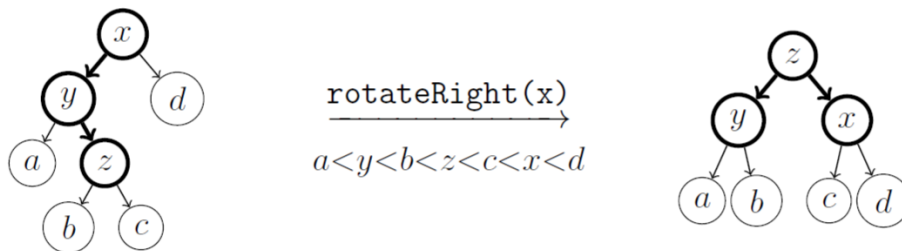
Пусть  $h(v.l) = h(v.r) + 2$  -- добавили в левого внука; левое поддерево выше, чем правое

Точно знаем, что левое поддерево непустое.



### 2. Добавление в $v.l.r$ или в $v.r.l$

Нужно сделать два маленьких вращения: сначала поддерево с корнем  $v.l$  влево, потом дерево с корнем  $v$  вправо.



## Код

```

1  add (Node* v, int x) {    // O(logn)
2      if (!v)
3          return new Node(x)
4      if (v.x <= x)
5          v.r = add(v.r, x)
6      else
7          v.l = add(v.l, x)
8      rebalance(v) // O(1): просто перевешиваем указатели
9      return v
10 }
```

Чтобы `rebalance()` работал за  $O(1)$ , в нодах будем хранить дополнительное поле `height`.

### RBST

Randomized Binary Search Tree

**Def 1:** *BST* может быть *RBST*, если любой ключ может быть корнем с одинаковой вероятностью.

**Def 2:** *RBST* -- это *BST*, полученное путем добавления всех ключей в изначально пустое дерево в случайной порядке.

#### **Th. (эквивалентность определений)**

Определения эквивалентны

*Док-во:* в случайной перестановке любой элемент будет корнем с вер-тью  $\frac{1}{n} \Leftrightarrow$  любой элемент может быть корнем с одинаковой вер-тью. То же самое верно для поддеревьев.

#### **Th. (матожидание для глубины) \todo**

*RBST* построено по ключам  $\{x_1, \dots, x_n\}$ .  $v_i$  -- узел ключа  $x_i$ . Тогда  $E[d(v_i)] = O(\log n)$

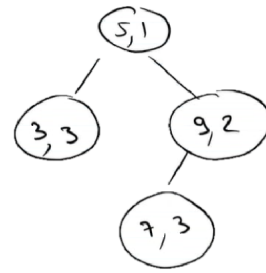
*Док-во:*

Выберем элемент из массива.  $x_i$  стал корнем с вероятностью  $\frac{1}{n}$ .

### Декартово дерево

aka *Treap* aka *Cartesian tree* =  $\{ (x_1, y_1), \dots, (x_n, y_n) \}$ .

- дерево поиска по координатам  $x$
- бинарная куча по координатам  $y$
- координаты  $y$  выбираются рандомно



### Th. (связь между Treap и RBST)

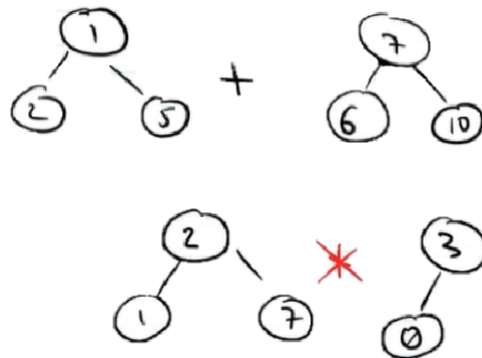
Treap является RBST по координатам  $x$

Док-во:

Корень =  $(x_i, y_i)$  где  $y_i = \min y$ . Координаты  $y$  случайные  $\Rightarrow$  любая вершина может стать корнем с одинаковой вероятностью  $\frac{1}{n}$ . Аналогично в поддеревьях.

### Merge

Мержим деревья  $a$  и  $b$ . Требование:  $\forall x \in a \leq \forall y \in b$ .



```

1 Node* merge(Node* a, Node* b) {
2     if (!a)
3         return b
4     if (!b)
5         return a
6     if (a.y <= b.y) {
7         a.r = merge(a.r, b)
8         return a
9     }
10    b.l = merge(a, b.l)
11    return b
12 }

```

## Split

```

1 pair<Node*, Node*> split(Node* t, int x) { // O(logn)
2     if (!t)
3         return (nullptr, nullptr)
4     if (t.x <= x) {
5         (a, b) = split(t.r, x)
6         t.r = a
7         return (t, b)
8     }
9     if (t.x > x) {
10        (a, b) = split(t.l, x)
11        t.l = b
12        return (a, t)
13    }
14 }

```

## 22-10-05

---

## Улучшения BST

### 1. Массовые операции

### (a) getSum(x1, x2)

Дано множество пар  $\langle x, y \rangle$ .

$$\text{getSum}(x_1, x_2) = \sum_{x \in [x_1, x_2]} y$$

Решение 1:

Храним *RBST*. Ключ --  $x$ , доп. информация -- сумма  $y$  в поддереве.

$$\text{split}(t, x_1) \rightarrow t_{<x_1}; t_{\geq x_1}$$

$$\text{split}(t_{\geq x_1}, x_2) \rightarrow t_{[x_1, x_2)}; t_{\geq x_2}$$

$$\text{answer} = t_{[x_1, x_2)}.\text{sum}$$

$$t_3 = \text{merge}(t_{[x_1, x_2)}, t_{\geq x_2})$$

$$t = \text{merge}(t_{<x_1}, t_3)$$

Решение 2:

```
1  getSum(t, qx1, qx2, cx1, cx2)
2  {
3      if ([qx1, qx2] && [cx1, cx2] == 0) { // нет пересечения
4          return 0;
5      }
6
7      if ([cx1, cx2] && [qx1, qx2] == [cx1, cx2]) { // [cx1, cx2] внутри [qx1,
qx2])
8          return t->sum;
9      }
10
11     return getSum(t->l, qx1, qx2, cx1, t->x) + getSum(t->r, qx1, qx2, t->x,
cx2);
12 }
13
14 int main() {
15     getSum(root, qx1, qx2, -inf, +inf);
16 }
```

Работает за  $O(\log n)$ : на каждом уровне будет посещено не больше 4 вершин, потому что разветвляемся максимум от двух вершин. Тогда время работы =  $4 \cdot \text{height} = 4 \log n$ .

### (b) assignValue(x1, x2, y)

$$\text{split}(t, x_1) \rightarrow t_{<x_1}; t_{\geq x_1}$$

$$\text{split}(t_{\geq x_1}, x_2) \rightarrow t_{[x_1, x_2)}; t_{\geq x_2}$$

Лениво меняем значения в поддереве  $t_{[x_1, x_2)}$  с помощью **push**:

```

1 Node::push()
2 {
3     if (toAssign == -inf) {
4         return;
5     }
6     if (l) {
7         l->toAssign = toAssign
8         l->y = toAssign
9     }
10    if (r) {
11        r->toAssign = toAssign
12        l->y = toAssign
13    }
14    toAssign = -inf;
15 }

```

Значения в поддереве с корнем  $t_{[x_1, x_2]}$  меняются во время первого `merge`.

## 2. Дерево поиска по неявному ключу \todo

(a) `get_kth_element(k)`

(b)

# 22-10-19

## Динамическое дерево отрезков

### Способ 1: меняем `vector` на `map`

Возьмём обычное ДО, заменим `vector<int> ST`, в котором значения дерева, на `unordered_map<ll, int> ST`.

За  $q$  запросов создадим  $\leq \min(q \log n, n)$ .

Платим временем, потому что мапа нкуда не спешит.

### Способ 2: не массив, а указатели

Храним ДО не на массиве, а на указателях. Создаём вершины, только когда они нужны. Вершины можно создавать в запросах `get`.

Вершины нет  $\Rightarrow$  значение = 0

### Список 3: сжатие координат

Требуем, чтобы задача была в оффлайне.

$q$  запросов. Сортируем  $2q$  чисел, присваиваем им координаты  $0, 1, 2, \dots$

## Двумерные (многомерные) запросы

Задача 1:

Дан массив длины  $n$ . Запросы вида `get(l, r, d, u)` -- найти #чисел на  $[l, r)$  : их величины  $\in [d, u)$

Задача 2:

Дано  $n$  точек на плоскости. Запросы `get(l, r, d, u)` -- #точек в множестве  $\{i : x_i \in [l, r), y_i \in [d, u)\}$

**Утверждение:** задача 1 и задача 2 -- это одно и то же.

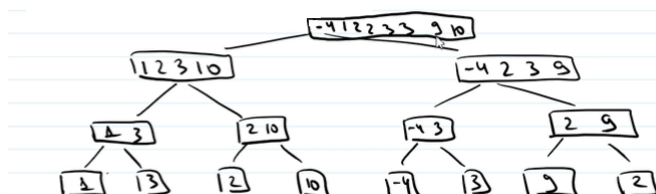
**Док-во:**

$1 \Rightarrow 2 : a[i] \rightarrow (i, a[i])$

$2 \Rightarrow 1 :$  сделаем сжатие точек по  $x_i$ . Получаем координаты  $\{x_i\} = \{0, 1, 2, \dots\}$  -- это индексы для массива

## Дерево mergeSort

Дерево отрезков сортированным массивов



Построение за  $O(n \log n)$  с помощью `mergeSort` (обычное ДО за  $O(n)$ ).

Задача 1:  $\text{Время} = \underbrace{\# \text{кусков}}_{O(\log n)} \cdot \underbrace{T(\text{ответ в куске})}_{2 \text{ бинар. поиска}} = O(\log^2 n)$



## 22-10-26

### Sparse table

Разреженная таблица -- структура данных, позволяющая отвечать на запросы минимума на отрезке за  $O(1)$  с препроцессингом за  $O(n \log n)$  времени и памяти.

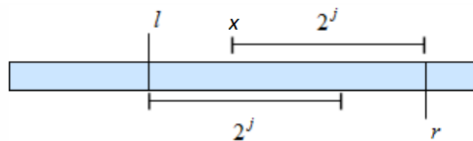
По сути, считаем минимумы на каждом отрезке длины  $2^k$ .

$$dp[i][k] = \min\{a_i, a_{i+1}, \dots, a_{i+2^k-1}\}$$

```
1 for i = 0..n:
2     dp[i][0] = parent[i]
3
4 for k = 1..logn:
5     for v = 0..n:
6         dp[v][k] = min(dp[v][k-1], dp[v + 2^{k-1}][k-1])
```

Запрос `get(l, r)`:

Находим степень  $j$  такую, что  $2^j \leq r - l$ . Тогда минимум на  $[l, r] = \min(dp[l][j], dp[x][j])$  ( $x = r - 2^j$ ).



Чтобы запрос работал за  $O(1)$ , преподсчитаем степени двойки. Типа  $k = \log[r - l]$ .

### LCA & Двоичные подъемы

#### isAncestor

Запускаем dfs от корня и запоминаем время входа/выхода. Тогда

```
1 isAncestor(a, b) {
2     if (tin[a] <= tin[b] && tout[a] >= tout[b])
3         return true;
4     return false;
5 }
```

*Док-во корректности:*

Рассмотрим вершинку . Вершинки, у которых  $\text{tin} \geq \text{tin}[a]$  , находятся в более правых ветках и у детей . Вершинки, у которых  $\text{tout} \leq \text{tout}[a]$  , находятся в более левых ветках и у детей . В пересечении только дети  $a$ .

## LCA

```
1 lca(v, u) {
2     if (isAncestor(v, u)) {
3         return v
4     }
5     for (int k = logn; k >= 0; --k) {
6         tmp = up[v][k]
7         if (!isAncestor(tmp, u)) {           // если не прыгнули слишком высоко,
8             v = tmp                           // то меняем v
9         }
10    }
11    return up[v][0]
12 }
```

## LA (level ancestor)

Запрос  $\text{LA}(v, k)$  – подняться в дереве от вершины  $v$  на  $k$  шагов вверх.

Уже умеем решать за  $< n \log n, \log n >$  двоичными подъёмами.

## Offline

Пройдем dfs-ом и будем запоминать пройденный путь. Для вершины сохранен путь от корня, можно за  $O(1)$  узнать предка на  $k$ -ом уровне.

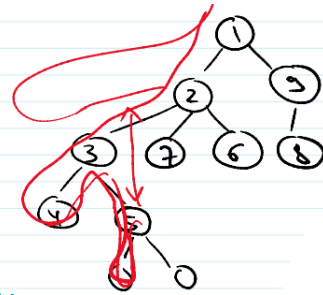
## Алгоритм Вишкина

Работает за  $< n, \log n >$ .

Выпишем высоты Эйлера обхода второго типа. Получим массив `height`.

Для запроса  $\text{LA}(v, k)$  знаем индекс  $j$  : `height[j] = v`. Найдем индекс  $\max i$  : `i <= j` && `height[i] = height[j] - k`.

v | 1 2 3 4 3 5 3 2 7 2 6 2 1 9 8 9 1  
 h | 0 1 2 3 2 3 2 1 2 1 2 1 0 1 2 1 0



$$\frac{V=5 \quad k=2}{h[v]=3}$$

**22-11-02**

**RMQ  $\pm 1$**

$$\forall i : |a_i - a_{i+1}| = 1$$

**ФКБ**

Хотим **RMQ** за  $\langle n, 1 \rangle$ .

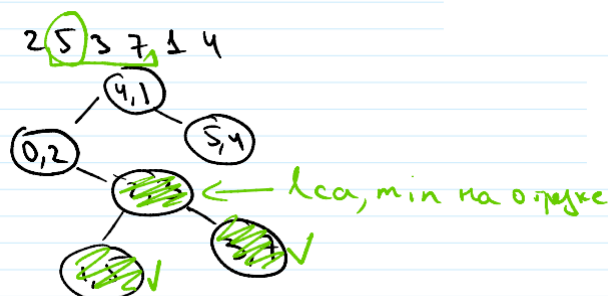
$$\text{RMQ} \xrightarrow{1} \text{LCA} \xrightarrow{2} \text{RMQ} \pm 1$$

## 1. RMQ $\rightarrow$ LCA

Строим декартач по массиву за  $O(n)$ .  $a[i] \rightarrow \langle i, a[i] \rangle$

По индексам это BST (?) (aka приоритет), по элементам -- бин куча.

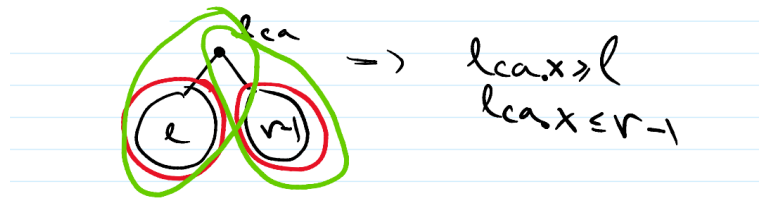
Как построить за  $O(n)$ ? Проходимся последовательно по массиву. В дереве будем всегда двигаться по самому правому пути (запоминать его конец). Если значение текущего элемента меньше, чем начало пути, то добавляем новую ветку справа от всех веток начала пути (теперь начало самого правого пути поменялось).



LCA для вершинок  $v$  и  $u = \min$  на отрезке  $[v.key, u.key]$

Док-во:

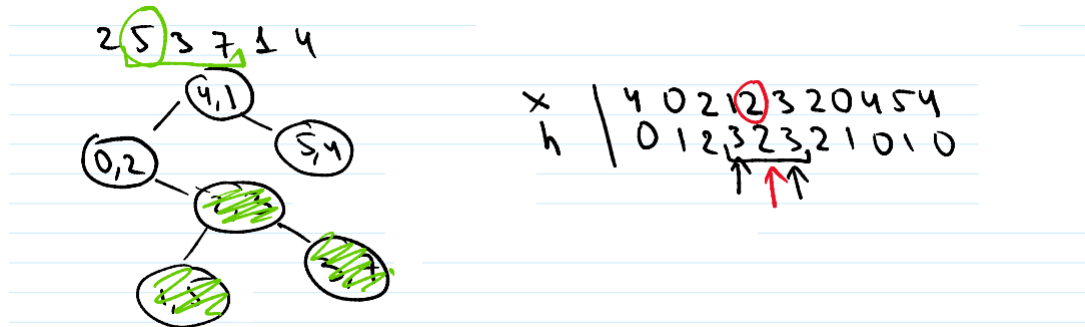
- $LCA \in$  отрезку



- $\forall i, i \in [l, r] : (i, a[i]) \in subtree(lca) \Rightarrow lca.value \leq a[i]$   
 $\Rightarrow lca = \min$  на отрезке

## 2. LCA $\rightarrow$ RMQ $\pm 1$

(1) Эйлеров обход второго типа (выписываем вершину и высоту)



Получаем массив  $h$ , в котором разница между элементами  $= \pm 1$ .

Выбираем в нем любое из вхождений вершинок, мин число между ними  $= lca$  этих вершинок.

(2) В  $h$  любой отрезок однозначно кодируется как  $(x, \pm 1, \dots, \pm 1)$ , где  $x$  -- число в начале отрезка.

Хотим для любого отрезка  $[l, r)$  быстро узнавать тип и быстро находить минимум.

$k = \frac{\log n}{2}$ . Строим sparse table.

Всего есть  $2^0 + 2^1 + \dots + 2^k \leq 2^{k+1} = O(\sqrt{n})$  различных отрезков.

Переберем все отрезки и найдем в них минимумы.

```
1 for (mask = 0..2^{k+1})
2   for (l = 0..k)
3     for (r = 0..k)
4       precalc[type][l][r] = min elem in [l, r];
```

## 22-11-09

### MST

Minimal Spanning Tree

### Th. (лемма о разрезе)

$V$  -- множество вершин,  $E$  -- множество ребер в исходном графе.

$V = A \sqcup B$  -- разрез графа

$T \subset E$  :  $T$  не пересекает разрез ( $\nexists e \in T : e = (ab), a \in A, b \in B$ )

Тогда  $\exists MST T_0 : T \subset T_0$

*Док-во:*

$e$  -- ребро минимального веса из разреза (

$e = (ab), a \in A, b \in B \wedge \forall e' = (a'b'), a' \in A, b' \in B : weight(e') \geq weight(e)$ )

Тогда  $e$  -- безопасное ребро для  $T$ , т. е.  $\exists MST T_1 : T \cup \{e\} \subset T_1$

Рассмотрим  $T_0$ .

1.  $e \in T_0 \Rightarrow T_1 := T_0$

2.  $e \notin T_0$ . Тогда в  $T_0$  есть как минимум одно ребро  $e' = (a'b')$ , пересекающее разрез (иначе  $T_0$  не связное, противоречие).

$T_1 := T_0 / \{e'\} \cup \{e\}$

$$\left. \begin{aligned} weight(T_1) &= weight(T_0) - weight(e') + weight(e) \\ weight(e') &\geq weight(e) \end{aligned} \right\} \Rightarrow weight(T_1) \leq weight(T_0)$$

$T_1$  -- дерево ( $|T_1| = |T_0| - 1 + 1 = n - 1$ )

Путь между  $a'$  и  $b'$  все ещё существует и равен  $a' \rightsquigarrow a \xrightarrow[e]{} b \rightsquigarrow b'$ . Если после замены нет пути из  $a'$  в  $a$ , то изначально  $T_0$  не был связным, противоречие.

3. Утв. 1:  $T_1$  — MST ( для (1) очев, для (2) следует из того, что размер =  $n - 1$  и  $T_1$  связное )

Утв. 2:  $T \cup \{e\} \subset T_1$

Для (1) очев, для (2):

$$e \in T_1$$

В  $T$  не было ребер из разреза по условию  $\Rightarrow e' \notin T \Rightarrow T \subset T_0/\{e'\} \Rightarrow T \subset T_1$

#### Смысл леммы:

Если есть какое-то множество ребер, которое можно дополнить до  $MST$ , то можно просто брать ребро минимального веса, соединяющее две компоненты.

## Алгоритм Прима

Строим граф  $T$  с нуля. Изначально в  $T$  находится одна вершинка.

Для вершинки  $u$  перебираем рёбра. Выбираем такое  $(u, v)$ , что:

- $w((u, v)) = \min$
- $v \notin T$

Делаем так до тех пор, пока в  $T$  не будут включены все вершинки

Работает за  $O(V^2 + E)$  на массиве,  $O(E \log V)$  на бин куче,  $O(V \log V + E)$  на Фиб. куче.

```
1  int main() {
2      vector<vector<int>> g;
3      vector<bool> used(n, false);
4      vector<int> dist(n, INF), parent(n, -1);
5      dist[0] = 0; // стартуем из 0
6      repeat(n) {
7          int v = -1;
8          for (int j = 0; j < n; ++j)
9              if (!used[j] && (v == -1 || dist[j] < dist[v]))
10                 v = j;
11          if (dist[v] == INF) {
12              cout << "No MST!\n";
13              return 0;
14          }
15          used[v] = true;
16          for (int u = 0; u < n; ++u)
17              if (g[v][u] < dist[u]) {
18                  dist[u] = g[v][u]; // в `dist` поддерживаем минимальное
ребро от компоненты до вершинки
19                  parent[u] = v;
20              }
```

```

21     }
22 }

```

## Алгоритм Краскала

Построение MST в связном графе

Строим мст с нуля. Сначала это пустой граф  $T$ . Сортируем рёбра по возрастанию, получаем массив `Edges`.

Добавляем ребро, если оно лежит между вершинкой  $\in T$  и вершинкой  $\notin T$ .

```

1  std::vector<int> parent(n, -1);
2  std::vector<int> rank(n, 0); // ну типа высота дерева
3
4  int find_set(int v) {
5      if (parent[v] == v) {
6          return v;
7      }
8      return parent[v] = find_set(parent[v]);
9  }
10
11 void union_set(int a, int b) {
12     int v_a = find_set(a);
13     int v_b = find_set(b);
14     if (v_a == v_b) {
15         return;
16     }
17     if (rank[v_a] < rank[v_b]) {
18         std::swap(v_a, v_b);
19     }
20     parent[v_b] = v_a;
21     if (rank[v_a] == rank[v_b]) {
22         ++rank[v_a];
23     }
24 }
25
26 int kraskal(std::set<std::pair<int, std::pair<int, int>>> &Edges) {
27     int cost = 0;
28     for (const auto &edge : Edges) {
29         int w = edge.first, u = edge.second.first, v = edge.second.second;
30         if (find_set(u) != find_set(v)) {
31             union_set(u, v);
32             cost += w;
33         }
34     }
35     return cost;
36 }

```

`find_set()` за  $O(\log n)$ , потому что высота дерева (aka rank) не больше  $\log n$ .

`union_set()` можно заставить работать за  $O(1)$ .

## Корректность алгоритма

Рассмотрим разрез графа:  $A \subset V$  &  $B = V/A$ .

Изначально  $T = \{\emptyset\}$ . Будем набирать рёбра в  $T$ .

$e$  -- ребро минимального веса, соединяющее  $a \in A$  и  $b \in B$ . Рёбра меньшего веса находятся либо в  $A$ , либо в  $B \Rightarrow$  их нет смысла брать. Остальные рёбра между  $A$  и  $B$  имеют больший вес. Значит  $e$  -- безопасное ребро, его можно добавить в  $T$ .

Если полученное  $T$  -- не дерево (т.е. не связно), то граф изначально был несвязный, противоречие.

$$\left. \begin{array}{l} T - \text{дерево} \\ T \subseteq MST \end{array} \right\} \Rightarrow T = MST$$

## 22-11-18

---

### DSU

DSU -- Система Непересекающихся Множеств

Умеет:

- `init(n)` -- создать  $n$  множеств из одного элемента
- `get(a)` -- узнать уникальный идентификатор множества, в котором лежит  $a$
- `join(a, b)` -- объединить множества, в которых лежат  $a$  и  $b$

### DSU на списках

```
1 vector<set> sets;
2 vector<int> id;
3
4 void init(int n) { // O(n)
5     for (int i = 1..n) {
6         id[i] = i;
7         sets[i].insert(i);
8     }
9 }
10
11 int get(int a) { // O(1)
```



```

12     return id[a];
13 }
14
15 void join(int a, int b) {    // O(n)
16     if (sets[a].size() < sets[b].size())
17         std::swap(a, b);
18     for (int x : sets[b]) {
19         id[x] = id[a];
20         sets[b].erase(x);
21         sets[a].insert(x);
22     }
23 }

```

### Th. (суммарное время работы join)

Суммарное время работы всех `join` не больше  $n \log n$ .

*Док-во:*

На каждом `join` кол-во множестве уменьшится вдвое  $\Rightarrow$  соединять множества можно не больше  $\log n$  раз  $\Rightarrow$  все `join` работают за  $O(n \log n)$ .

## DSU на деревьях

Базовая реализация:

```

1  vector<int> p;
2
3  void init(int n) {
4      for (int i = 1..n) {
5          p[i] = i;
6      }
7  }
8
9  int get(int a) {
10     return p[a] == a ? a : get(p[a]);
11 }
12
13 int join(int a, int b) {
14     pa = get(a);
15     pb = get(b);
16     p[pb] = pa;
17 }

```

## Оптимизации

### 1. Сжатие путей -- перевешивание вершинок корню

```
1 int get(int a) {
2     if (p[a] == a)
3         return a;
4     return p[a] = get(p[a]);
5 }
```

### 2. Сохранение рангов для перевешивания меньшего дерева к большему.

Ранг вершины – глубина поддеревы, если бы не было сжатия путей.

```
1 vector<int> rank;
2
3 void init(int n) {
4     for (int i = 1..n) {
5         p[i] = i;
6         rank[i] = 0;
7     }
8 }
9
10 join(a, b) { // O(logn)
11     pa = get(a);
12     pb = get(b);
13     if (rank[pa] < rank[pb])
14         std::swap(pa, pb);
15     p[pb] = pa;
16     if (rank[pa] == rank[pb])
17         rank[pa]++;
18 }
```

### Th. (про размер поддеревы)

Если у  $v$  ранг  $k$ , то её поддерево размера  $\geq 2^k$

*Док-во:*

База:  $k = 0$ . Размер дерева = 1.

Переход:

Чтобы получить дерево ранга  $k + 1$ , нужны для дерева ранга  $k$ . Подвешиваем дерево с корнем  $u$  к дереву с корнем  $v$ .

Новый размер дерева с корнем  $v$  =  $\underbrace{\text{старый размер дерева с корнем } v}_{\geq 2^k \text{ (по предп. инд.)}} + \underbrace{\text{размер дерева с корнем } u}_{\geq 2^k \text{ (по предп. инд.)}} \geq 2^{k+1}$

**Th. (время работы join и get)**

`join` и `get` работают за  $O(\log n)$ .

*Док-во:*

Работа `join` зависит от `get`. `get` работает за  $\text{rank}(\text{root}) \leq O(\log n)$ .