Алгоритмы, II семестр

Весна 2015, лектор: Копелиович Сергей

Собрано: 9 июня 2015 г. 13:52

Оглавление

Глава 1

Деревья поиска

Сколько бывает деревьев поиска из элементов $0, 1, \dots, n-1$?

$$K_n = \sum_{i=0}^{n-1} K_i \cdot K_{n-i-1}$$

(узнаём формулу чисел Каталана)

Хранить дерево будем структурой:

```
struct Node {
    Node *L, *R;
    int x;
    DataType Data;
};
```

Замечание 1.0.1. Если бы дерево было не бинарное, то мы бы сделали не `*L` и `*R`, то есть левого и правого сыновей, а `*L` И `*Next` — левого ребёнка и правого соседа.

Замечание 1.0.2. Ещё дерево можно хранить как `n`, `x[n]` и `parent[n]` — то есть как вектор значений и родителей, доступ осуществляется по номеру вершины.

```
void insert(node* v, int x) {
    if(!v)
        return new Node(0, 0, x);
     if(x < v->x)
        return new Node(insert(v->L, x), v->R, x);
    if(x > v->x)
        return new Node(v->L, insert(v->R, x), x);
    return v;
}
```

Def 1.0.1. Персистентная структура данных (persistent data structure) — структура данных, у которой в любой момент доступны все её версии (от начальной и до текущей).

```
Node* & find(Node* &v, int x) {
    if(!v)
        return v;
    if(v -> x == x)
        return v;
    return find(x < v-> x ? v -> L : v-> R, x);
}
```

Замечание 1.0.3. Другой способ сделать `Insert`: `find(Root, x) = new Node(0, 0, x);` Правда, тут стоило бы отдельно рассматривать случай возможного наличия x в дереве.

Def 1.0.2. Поддерево — множество всех вершин, которые мы рекурсивно обошли, выйдя из данной(которая тоже включается).

Теперь мы хотим сохранить дерево в файл, причем так, чтобы его было удобно читать. Пусть узлы нашего дерева описываются структурой

```
struct node* {
    node *L, *R;
    int x;
};
```

Тогда функция сохранения в файл будет выглядеть следующим образом:

```
Save(node* v) {
    if(!v) {
        out(0); //out() "--- записать в файл четверку байт
        return;
    }
    out(v->x); //предположение: x != 0
    Save(v->L);
    Save(v->R);
}
```

Функция чтения также рекурсивна.

Можно хранить (в файле) компактнее — не хранить ничего, кроме ключей. Порядок обхода при хранении делать прямым и понимать, что правое поддерево началось в тот момент, когда мы в первый раз встретили ключ, больший x. Упражнение по этому поводу: сделать это за $O(n)$.

Def 1.0.3. Порядок xLR — прямой порядок обхода дерева.

Порядок LxR — симметричный порядок обхода дерева.

Замечание 1.0.4. Если выписать элементы дерева в порядке симметричного обхода, то мы получим отсортированный массив.

Еще три интересные нам функции — это `Left(node*)`, `Right(node*)` и `Delete(node*&)` (ей мы будем скармливать результат `find`— а). `Left` и `Right` будут пока устроены обычным образом — пойти влево(вправо) один раз, а затем до упора вправо (влево, соответственно). И отдельно случай, когда у узла нет соответствующего поддерева.

```
Delete(node* &v) {
    if(!v->R)
        v = v->L;
    else {
        node* &t = Right(v);
        swap(v->x, t->x);
        t = t->R;
    }
}
```

Также полезной функцией будет `DebugOuput`, который можно реализовать тремя основными способами:

1. Первый рекурсивный способ: мы выводим рекурсивно сначала левое поддерево в скобках, затем сам элемент, затем правое поддерево, также в скобочках и также рекурсивно. Получается примерно так:
$$(L)x(R)$$
2. Второй рекурсивный способ: выводить также в порядке прямого обхода, но вертикально, передавая в качестве параметра при рекурсивном вызове количество пробелов, которое необходимо поставить перед очередным элементом (это количество от уровня к уровню должно как-либо увеличиваться). Псевдокод выглядит так:

```
Out(node* v, int depth) {
    Out(v->L, depth + 1);
    print(' ' * depth, x); //привет, Python? :)
    Out(v->R, depth + 1);
}
```

3. Способ вывести так, как мы дерево обыкновенно рисуем: завести двумерный буфер, из размера левого сына получать размер отступа, в качестве параметра передавать «координаты» точки, в которую надо писать.

Теперь покажем, что наше дерево поиска хорошо работает. Заметим, что

Insert: $O(height)$

Delete: $O(height)$

Def 1.0.4. Случайное дерево поиска — дерево поиска, полученное добавлением случайной перестановки в пустое дерево.

Def 1.0.5. Глубина случайного дерева — максимальная из глубин вершин.

Def 1.0.6. Глубина вершины — расстояние от неё до корня.

Def 1.0.7. Глубина дерева («высота» в терминах AVL-деревьев) — максимальная из всех глубин вершин этого дерева.

Def 1.0.8. Сбалансированное дерево поиска — такое дерево поиска, глубина которого равна $O(\log n)$

Теорема 1.0.1. Средняя глубина вершины случайного дерева равняется $O(\log n)$.

Лемма 1.0.1. Сумма размеров поддеревьев равна сумме глубин по всем вершинам.

▶
$$\sum size_i = \sum_b |\{a\colon b \text{ — предок } a\}| = \sum_a |\{b\colon b \text{ — предок } a\}| = \sum depth_i$$
◀

Лемма 1.0.2. У любого поддерева случайного дерева корень — случайная вершина из этого поддерева.

▶ Корень — это то, что мы добавили первым. Первый элемент случайной перестановки — это случайная величина. ◀

Мы хотим сделать наше дерево сбалансированным (как всегда, для скорости). `Insert` будет работать за $\Omega(\log n)$, потому что мы можем за линию сделать симметричный обход и получить отсортированный массив, и если наш `Insert` работал быстрее, то мы могли бы сортировать любые данные быстрее, чем за $O(n \log n)$. Глубина BST (Balanced Search Tree) — $O(\log n)$. Операции `Find` и `Delete` можно сделать за $O(1)$ — для этого нужно, в первом случае, использовать hash table, а во втором не удалять элемент, а просто помечать удалённым.

Операции `Right` и `Left` можно сделать за амортизированное $O(1)$:

$$T = \frac{\sum_{v=1}^{n} Right(v)}{n} = O(1)$$

Но, когда очень хочется, можно сделать и так, чтобы $O(1)$ было в хужшем случае: для этого достаточно поддерживать ссылки на своих соседей `Right` и `Left` и пересчитывать их после `Insert` и `Delete`.

Глава 2

AVL, Treap, Impicit key, Persistent search tree

2.1. AVL-дерево

2.1.1. Основное

1962, Адельсон— Вельский— Ландис.

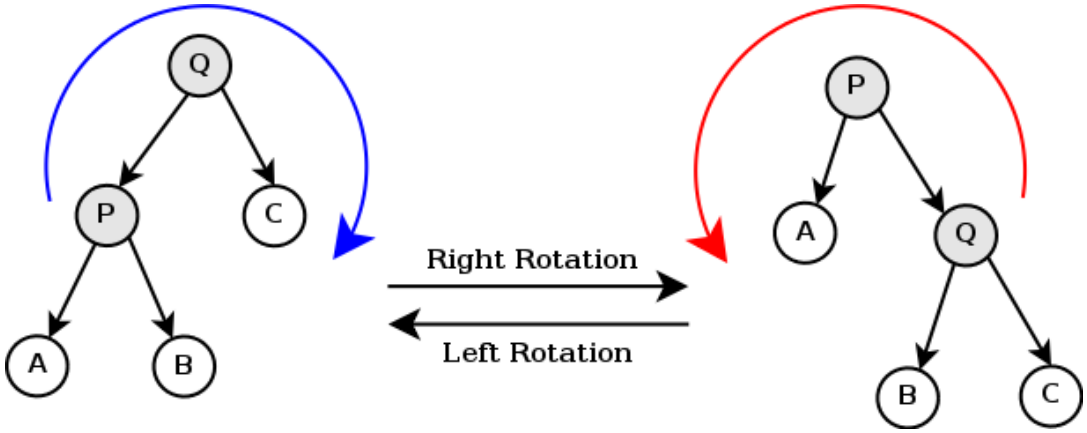
AVL-дерево — это BST, у которого поддерживается инвариант, что разница высот левого и правого поддерева для каждой вершины не превосходит 1. Зная это, можно получить следующие ограничения на размер дерева:

$$size[h] \geqslant size[h-1] + size[h-2]$$
$$2^h - 1 \geqslant size[h] \geqslant F_n$$

Из данного соотношения видно, что $h = O(\log n)$.

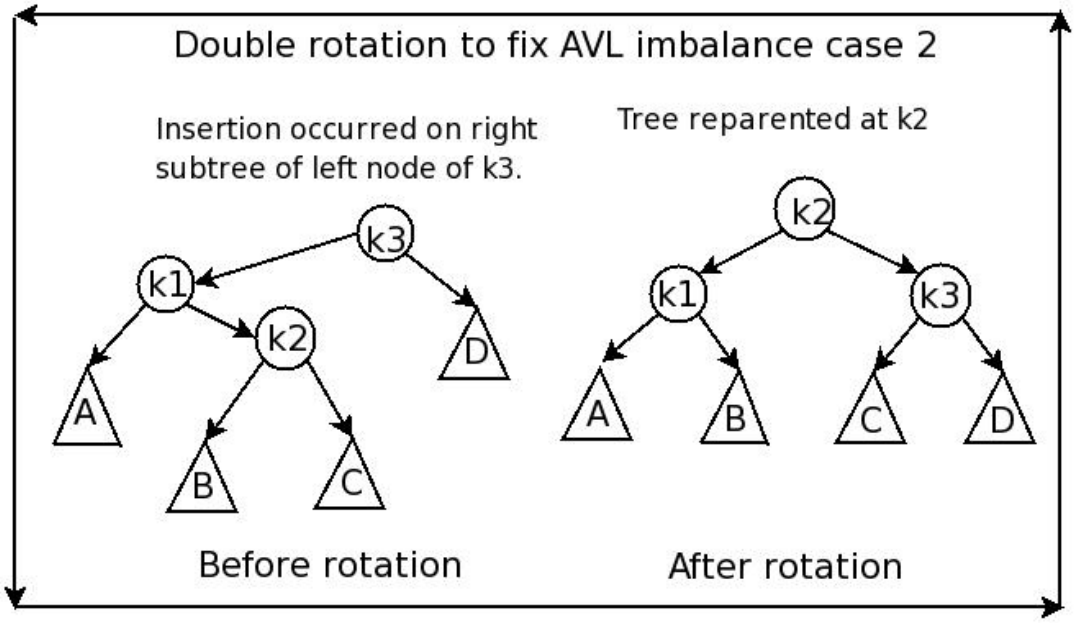
2.1.2. Балансировка

Чтобы балансировать наше дерево, мы будем поступать так: как только мы как-то изменили вершину, смотрим, какие высоты у ее левого и правого сыновей. Пока они различаются более, чем на 1, то совершаем один из четырех поворотов: Right rotation, Left rotation, Big right rotation и Big left rotation.



Из картинки видно, что правый поворот мы совершаем тогда, когда высота левого поддерева больше, чем высота правого. Тем не менее, и этот случай делится на два. Right rotation мы совершаем только в том случае, если $P.h > C.h \wedge A.h \geqslant B.h$ (аналогично для (малого) левого поворота — он происходит, тогда, когда $Q.h > A.h \wedge C.h \geqslant B.h$).

Остались случаи, когда $P.h > C.h \wedge A.h < B.h$ (и симметричный для Big Left Rotation). Здесь мы будем применять Big right rotation.



То есть поворот здесь происходит, если $k_1.h > D.h \wedge k_2.h > A.h$. Аналогично делается Big left rotation. Теперь можем заметить, что Right rotation — это вращение вокруг ребра P—Q. А Big right rotation — вокруг ребер k_1—k_2 (обычный left rotation), а потом k_1—k_3 (обычный right rotation).

2.1.3. Реализация

Код этого дела будет выглядеть примерно так:

```
Node* right_rotation(Node* n) {
    Node* left = n->l;
    n->l = left->r;
    left->r = n;

    upd(left); upd(n);

    return left;
}
```

`upd` — функция, в которой пересчитываются высоты.

```
Node* make_balanced(Node* n) {
    size_t lh = get_h(n->l);
    size_t rh = get_h(n->r);

    while (abs(lh - rh) >= 2) {
        if (lh > rh) {
            if (get_h(n->l->l) >= get_h(n->l->r)) {
                n = right_rotation(n);
            } else { //Это Big right rotation
                n->l = left_rotation(n->l); // поворот вокруг k1-k2
                n = right_rotation(n); //поворот вокруг k1-k3
```

```
            }
        } else {
            if (get_h(n->r->r) >= get_h(n->r->l)) {
                n = left_rotation(n);
            } else { //А это Big left rotation
                n->r = right_rotation(n->r);
                n = left_rotation(n);
            }
        }

        lh = get_h(n->l);
        rh = get_h(n->r);
    }

    upd(n);
    return n;
}
```

2.1.4. Удаление

Как делать добавление понятно — так же, как и в просто в BST, только надо будет следить за высотами поддеревьев. Удаление легче всего делать следующим образом: свопнемся с первым большим себя элементом (Right) из нашего поддерева. Теперь себя можно честно удалить, подвесив своего правого сына (если таковой был) вместо себя к нашему предку.

2.1.5. Ссылки

На Хабре есть хорошая статья про AVL деревья: http://habrahabr.ru/post/150732/

2.2. Неявный ключ

Пусть нам задан некоторый массив чисел и мы хотим по нему построить BST так, чтобы при симметричном обходе (то есть просто если посмотреть на его ключи слева-направо) все его ключи были в том же порядке, что и в изначальном массиве.

По-идее, мы могли бы просто добавлять пары (i, a_i) и интерпретировать i как x, по которому и строим BST, а a_i — просто некоторая информация, хранящаяся в вершине. В принципе да, мы решили поставленную задачу — ключи идут в нужном порядке.

Теперь заметим, что на самом деле хранить сам ключ i в вершине нам совершенно необязательно — нам просто нужно знать для каждой вершины сколько вершин левее нее. Если больше, чем i, то наша вершина должна быть левее текущей. Если меньше, то правее.

Получаем, что мы можем вообще не хранить x для вершин. Храним только поле `left_size`. Собственно, в этом и заключается идея неявного ключа. Когда мы пользуемся «явным» ключом, мы как бы говорим, что нам не важен относительный порядок элементов, а важно лишь кто кого больше. Если мы используем ”неявный”ключ, мы не сможем сделать так, чтобы a_i были расположены в соответствии с тем, как они между собой соотносятся, но зато мы всегда знаем их порядок в массиве. То есть неявный ключ, в отличие от явного, применим тогда, когда нам важен порядок, в котором заданы элементы, и нам абсолютно все равно, как соотносятся $data(a_i)$.

Поиск k-го элемента в этом дереве будет выглядеть примерно так:

```
int find(Node* n, int k) {
    if (n->lsize == k)
```

```
3         return n->data;
4
5         if (n->lsize > k)
6             return find(n->l, k);
7         else
8             return find(n->r, k - n->lsize);
9         }
```

2.3. Персистентность

Чтобы сделать наше дерево персистентным (то есть таким, что мы всегда как-то храним все его предыдущие версии и можем к ним откатиться), надо просто немного изменить добавление (и удаление) вершины. Вместо того, чтобы делать какие-то изменения в текущей вершине, мы будем создавать ее точную копию и изменять именно ее. Возвращать из функции нужно тоже именно эту новую вершину. Дальше нужно просто хранить ссылки на все корни.

Вообще, любое сбалансированное дерево можно сделать персистентным, увеличив время работы в $\log n$ раз.

Кстати, заметим, что мы только что научились делать персистентный массив — строим персистентное BST по неявному ключу на элементах этого массива.

2.4. Декартово дерево

2.4.1. Основное

1989, Cecilia R. Aragon, Raimund Seidel.

Декартово дерево (Treap, Дерамида, Курево) — BST, в котором выполняется следующее свойство — оно является BST по нашим x-ам и в то же время кучей по y-ам. То есть пусть нам дано множество пар:

$$\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots, \langle x_n, y_n \rangle$$

где x_n и y_n — какие-то случайные перестановки. Докажем два факта:

1. Существует ровно один Treap, построенный на этих ключах.

   Это более-менее очевидное утверждение. Просто заметим, что есть какой-то наибольший y_k. Он должен быть в корне Treap'а. Заметим теперь, что x_k поделил нам задачу на две независимых, в которых мы так же единственным образом восстанавливаем корни, которые будут детьми $\langle x_k, y_k \rangle$, и т.д.

2. Если $y_1, y_2, ..., y_n$ — случайная перестановка, то матожидание высоты Treap'а — $O(\log n)$.

   Доказательство примерно такое же, как и для qsort'а. Просто теперь на каждом отрезке у нас случайным образом выбирается корень (т.к. y-ки образуют случайную перестановку) поддерева и задача разделяется на две независимых.

   На практике, конечно, никто не генерирует рандомные перестановки — просто каждый раз, создавая очередную вершину, генерируем ей рандомный y.

2.4.2. `Split` и `Merge`

У декартова дерева есть 2 основных операции — split и merge. Split по заданному ключу делит дерево на два — в первом из них ключи меньше заданного, во втором больше. Merge принимает два

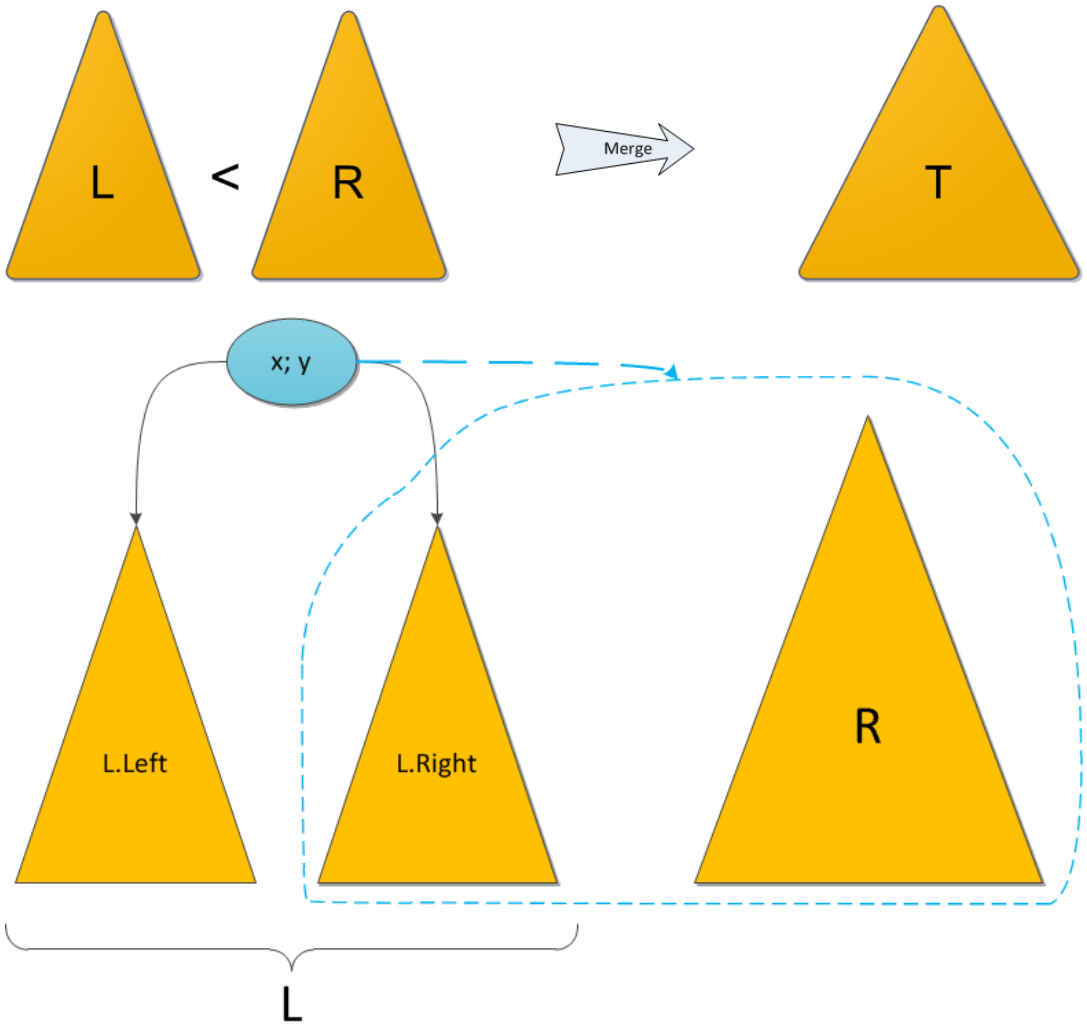
дерева, первое из которых меньше второго (все x из первого меньше, чем x из второго) и делает из них одно.

```
Treap Merge(Node* n1, Node* n2) {
    if (!n1) {
        return n2;
    }
    if (!n2) {
        return n1;
    }

    if (n1->y < n2->y) {
        Treap tmp = Merge(n1->r, n2);
        n1->r = tmp.root;

        return Treap(n1);
    } else {
        Treap tmp = Merge(n1, n2->l);
        n2->l = tmp.root;

        return Treap(n2);
    }
}
```

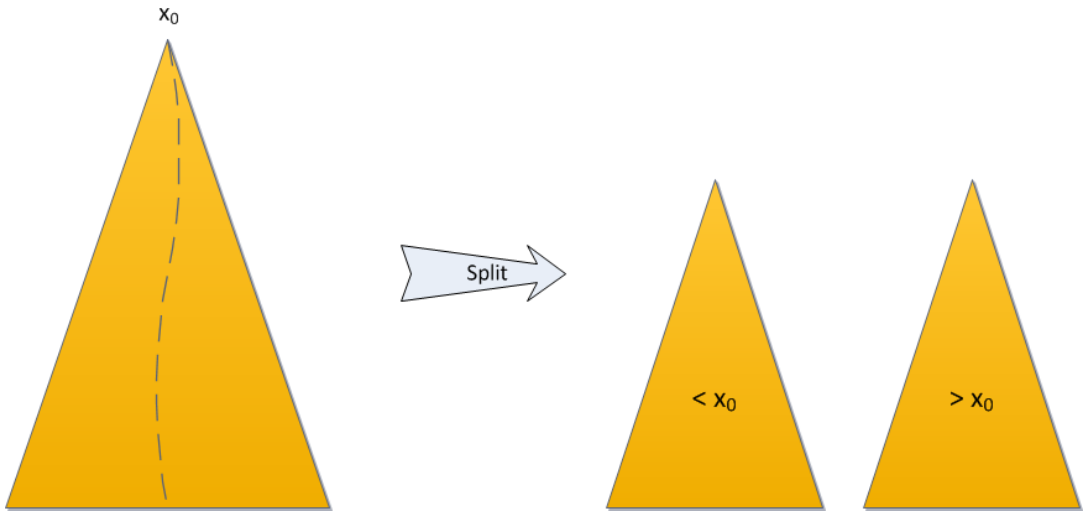


```
TreapPair split(Node* n, int x) {
    if (!n) {
        return mp(Treap(NULL), Treap(NULL));
    }

    if (n->x < x) {
        TreapPair tmp = split(n->r, x);
        n->r = tmp.first.root;

        return mp(Treap(n), tmp.second);
    } else {
        TreapPair tmp = split(n->l, x);
        n->l = tmp.second.root;

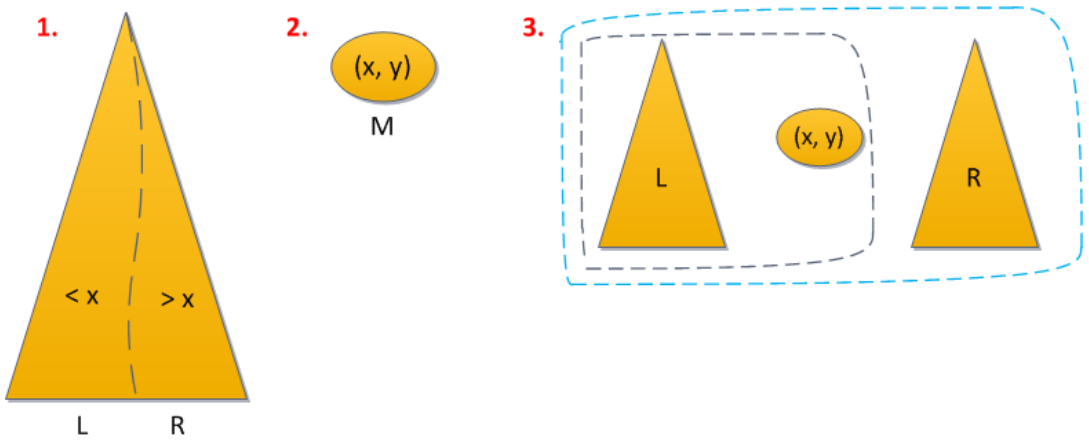
        return mp(tmp.first, Treap(n));
    }
}
```



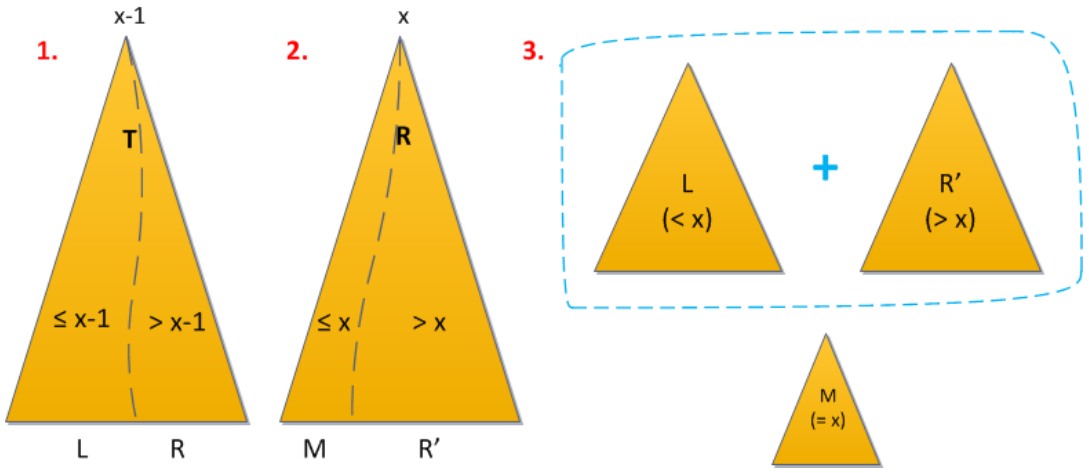
2.4.3. Insert/Delete

Теперь понятно, как должны выглядеть insert и delete. Insert — разделил дерево на 2 — меньшее и большее нашего x-а. Теперь примержили к первой половине наш x, после чего этот результат примержили ко второй половине. Аналогично Delete — дважды разрезаем, чтобы получить нужную вершину, после чего мержим два других дерева без этой вершины. То есть

$$Insert = Split + Merge + Merge$$



$$Delete = Split + Split + Merge$$



Теперь ускорим `Insert` в 2 раза. Сейчас мы делаем два спуска по дереву (один из Merge'й на самом деле работает за $O(1)$). Можно делать только 1: идем по дереву поиска, пока не получим тот момент, `n->y > y` (если считать, что минимум y-ов должен быть в корне). Теперь Split можно запускать уже только с этого места. После чего создаем новую вершину и подвешиваем за нее те два дерева, что вернул Split.

2.4.4. Ссылки

На Хабре есть отличный цикл из трех статей про декартово дерево (эти замечательные картинки взяты оттуда): http://habrahabr.ru/post/101818/

2.5. Случай равных x-ов

1. Можно хранить в вершине не только x, но и $count$ — кол-во этих x-ов.
2. Хранить пару $\langle x, i \rangle$ — не только x, но и его порядковый номер.
3. Говорим, что каждый следующий x чуть больше предыдущего (то есть просто в split'е все равные нашему x-ы кидаем в левое дерево).

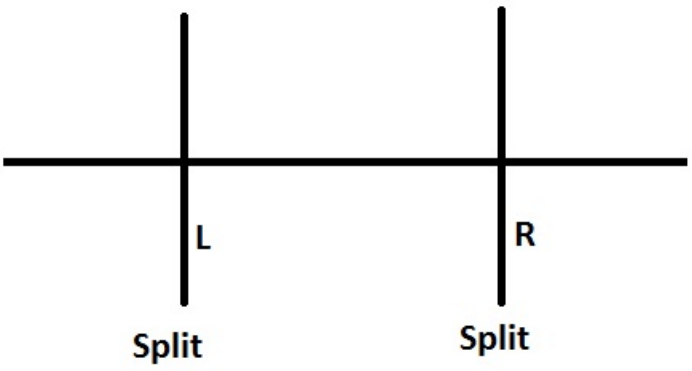
Глава 3

Операции на отрезке, Rope, SkipList, 2-3-Tree, B-Tree, RB-Tree, AA-Tree

3.1. Неявный ключ: операции на отрезке $[L, R]$

1. `Sum(L, R)`
2. `Min/Max(L, R)`
3. `Reverse(l, R)`
4. `Add(L, R, d)`
5. `Set(L, R, val)`

$[L, R]$, $L \leqslant x \leqslant R$
«Выспличиваем отрезок» от L до R.



В вершине храним флаг, присвоено ли что-нибудь, если да, то храним значение, иначе None.
`Root->value = ...`

3.1.1. pushDown

```
pushDown(v)
{
    if (v->value != None)
    {
        v->l->value = v->value;
        v->r->value = v->value;
        v->value = None;
        x = v->value;
        v->value = None;
    }
}
```

Update:

```
    v->sum = v->value != None ? v->value * v.size :
    v->l->sum + v->r->sum + v->x;
```

Замечание 3.1.1. Update не нужен, если перед заходом в сыновей всегда делать `pushDown()`.

3.1.2. Reverse

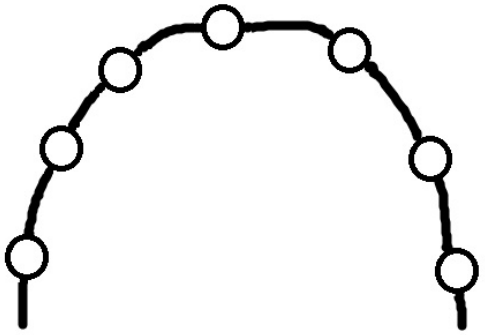
```
if (Rev == 1)
{
    L->Rev ^= 1;
    R->Rev ^= 1;
    swap(L, R);
    Rev = 0;
}
```

3.1.3. Sum

```
getSum(v, vL, vR, L, R) //v, [vL, vR] -- отрезок соответствующий v, [L, R]
{
    if (v == NULL)
        return 0;
    if (vL > R || vR <  L)
        return 0;
    if (vL >= L && vR <= R)
        return v->sum;
    return getSum(v->L, vL, v->x - 1, L, R)
        + getSum(v->R, v->x + 1, vR, L, R)
            + (L <= v->x <= R ? v-> x : 0);
}
```

Асимптотика $O(\log N)$, так как на каждом уровне рассмотрим не более 4 отрезков.

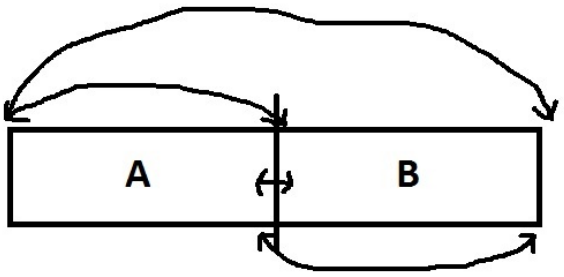
3.2. Rope



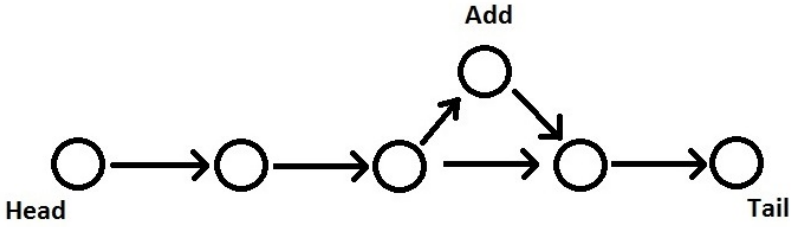
Def 3.2.1. Rope – интерфейс.

1. `Access(i)`
2. `Link`
3. `Cut`

Реализация: treap+ reverse, так как чтобы сделалть правильный Link нужно уметь переворачивать отрезки, потому что можем соединять любой конец одного отрезка с любым концом другого.



3.3. SkipList

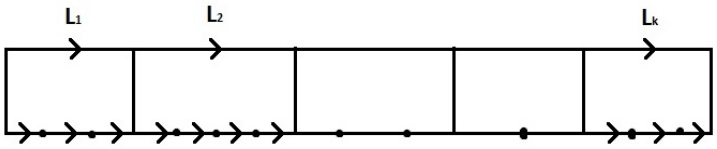


3.3.1. Operations

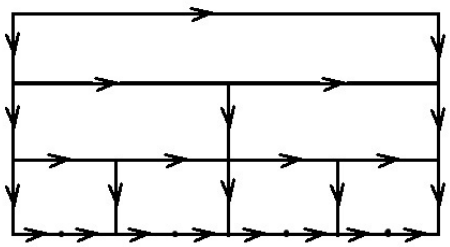
1. `Find = O(k)`
2. `Add = Find + O(1)`
3. `Delete = Find + O(1)`
4. `Split = Find + O(1)`

3.3.2. Find $= O(\sqrt{N})$

Разобьем массив на $\sqrt{N}$ кусочков и будем хранить в первом элементе отрезка ссылку на первый элемент следующего отрезка, также запомним L_i – на сколько элементов мы при этом прыгаем. При добавлении элемента просто изменяем L_i для начала текущего отрезка. Если $L_i \geqslant 2\sqrt{N}$, тогда Rebuild. Теперь Fnd за $O(\sqrt{N})$



3.3.3. Ideal Model



Всего $\log N$ уровней и на каждом уровне не более одного раза мы идем в право, значит Find $= O(\log N)$. Остальные операции не работают.

3.3.4. Skip-List

Храним $\log N$ списков, в первом лежат все элементы, во втором каждый элемент из первого листа присутствует с вероятностью $\frac{1}{2}$, на третьем каждый элемент из второго списка лежит с вероятностью $\frac{1}{2}$ и т.д.

$M_{i,k}$ — Мат. ождиание того, что элемент i окажется на k-ом уровне.$P = \frac{1}{2^k}$, $\boxtimes = 1$, значит $M_{i,k} = P\xi = \frac{1}{2^k}$. Значит, матожидание количества вершин на k-ом уровне равно $\frac{N}{2^k}$, т.к. матожидания складываются.

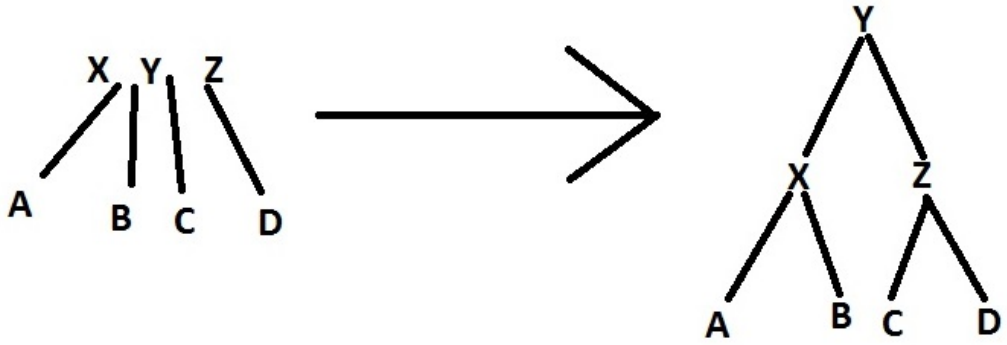
Всего $\log N$ переходов вниз и на каждом уровне Мат. ожидание на ход вправо $O(1)$.

Если мы сделаем k прыжков в право, значит k элементов подряд не смогли прыгнуть на уровень выше, вероятность этого $\frac{1}{2^k}$. Мат. ожиание k прыжков в право равно $\frac{1}{2^k}$.

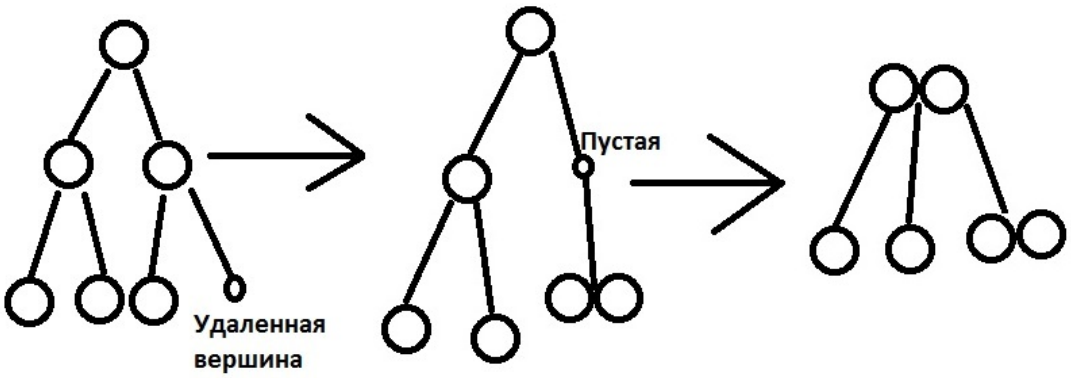
$X = \frac{1}{2} + \frac{1}{4} + \cdots = 1$. Сложили мат. ожидания.

3.4. 2-3-Tree

Все ключи различны. $A < X < B < Y < C$. Все листья имеют одинаковую глубину.
Добавление: спускаемся по дереву до листа и приклеиваем вершину. Могли получить 4-вершину.



Возможно, придется сделать так и выше, возможно изменится корень.
Удаление: избавляемся от пустых вершин.



3.5. B-Tree

K-Tree == B-Tree.

$[2-3] \longrightarrow [k, 2k)$

Пусть $k = 1024$, тогда получим очень маленькую высоту дерева.

$\log_{2k} N \leqslant h \leqslant \log_k N$

$$\frac{\log_k N}{\log_{2k} N} = \frac{N/\log k}{\log N/\log 2k} = \frac{\log k + \log 2}{\log k} = 1 + \log_k 2$$

$k = 1024$, $h \leqslant 3$, $N = 1e9$

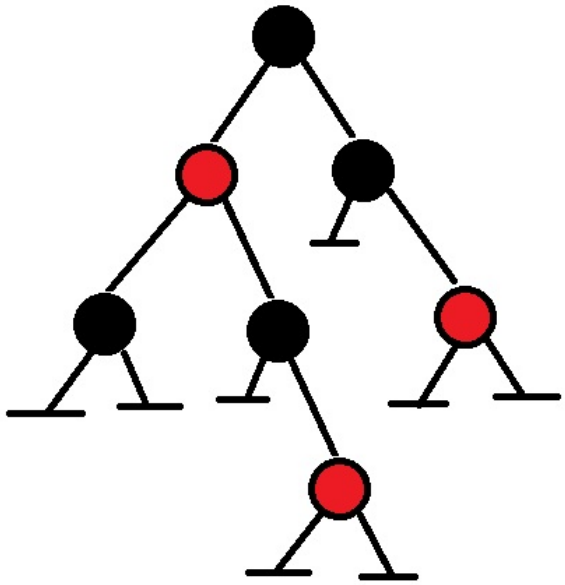
Легче хранить на диске, можно читать вершины с диска.

Диск: $\log_k N$. Время: $k \log_k N \longrightarrow \log_2 k \log_k N$.

3.6. RB-Tree

3.6.1. Описание

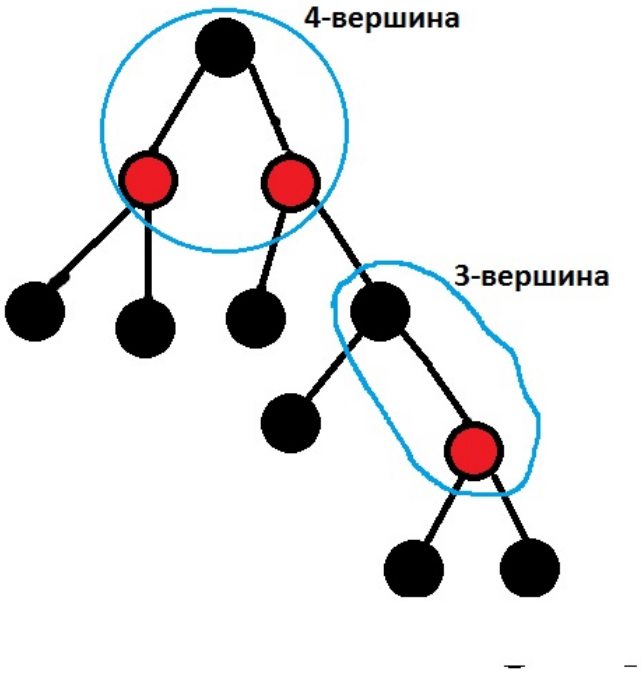
1. Бинарное дерево.
2. Вершины покрашены в красные и черный цвет, причем нет двух красных вершин соединенных ребром.
3. Все фиктивные листья имеют одинаковую черную глубину.
4. $2^h - 1 \leqslant size \leqslant 2^{2h} - 1$, где h — черная глубина дерева.



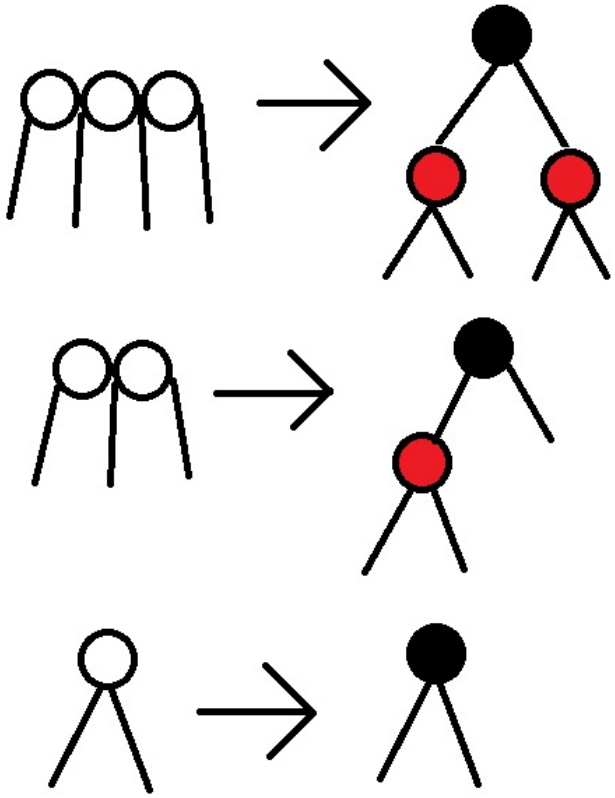
3.6.2. RB-Tree и 2-3-4-Tree

RB-Tree $\longleftrightarrow$ 2-3-4-Tree

Объединим красные вершины с их родителями и получим RB $\longrightarrow$ 2-3-4.

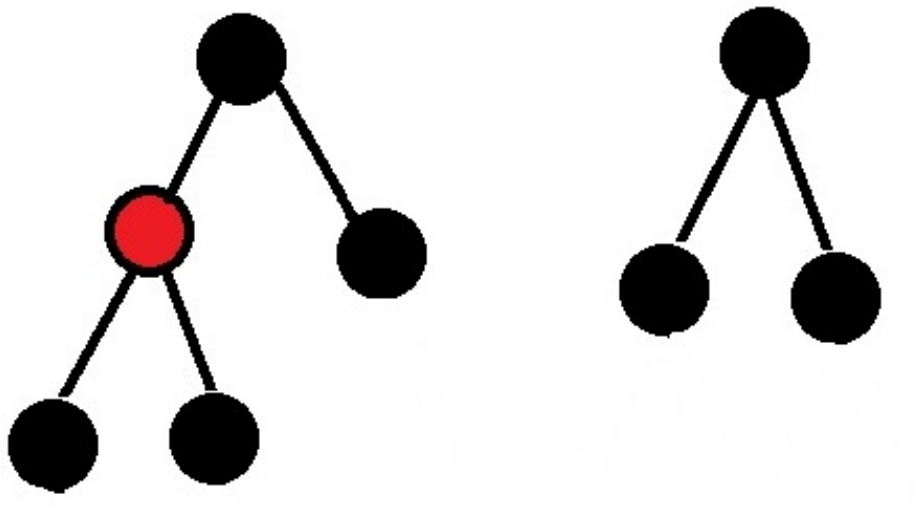


Следующими преобразованиями получим RB ⟵ 2-3-4.



3.6.3. AA-Tree

AA-Tree — вид RB-Tree, так же AA-Tree ⟷ 2-3-Tree, т.к. есть только такие конструкции:



$2^h - 1 \leqslant size \leqslant sqrt3^h - 1$, где h — черная высота дерева.

3.6.4. B^+-Tree

Является 2-3-Tree.
Все ключи хранятся в листьях, а в вершинах храним Max/Min в поддереве.

3.6.5. B^*-Tree

$[2-3] \longrightarrow [k, 1.5k)$

3.7. Персистентность

3.7.1. Offline

Пусть есть какая-то структура данных, мы хотим ее сделать персистентной — научиться уметь отвечать на запросы к конкретной версии. Пусть мы умеем для каждой операции делать обратную к ней. Тогда можно легко превратить эту структуру в персистентную, отвечая на запросы `Offline`. Давайте заметим, что версии образуют, подвешенное дерево, корнем которого является исходная структура данных, а ребра — запросы, создающие новую версию. Давайте обойдем это дерево `dfs`-ом. Тогда спуск в ребенка — применение какой-либо операции, а подъем из него — ее откат. Когда мы находимся в вершине, рассматриваем все запросы к версии, которой соответствует данная вершина, и отвечаем на них.

3.7.2. Garbage Collector

Задача: есть массив элементов, хотим на нем применять функцию `memmove(from, to, len)` — взять отрезок $[from; from + len)$, скопировать его в отрезок $[to, to + len)$. Хотим научиться решать эту задачу за $O(\log N)$. Казалось бы, это делается обычным декартовым деревом по неявному ключу — просто берем, с помощью `split` выделяем отрезки $[begin; from), [from; from+len), [from+len, end]$ и $[begin; to], [to, to + len), [to + len; end)$, после чего склеиваем эти отрезки в порядке 4, 2, 6.

Но обычное декартово дерево с такими запросами справляется плохо, так как при попытке изменить вершину, которая была уже скопирована, будут изменены две вершины сразу. Поэтому

наше дерево должно быть персистентным. А это значит, что каждый `split, merge` будет выделять дополнительные $O(\log N)$ памяти. Но нам нужна каждый раз только одна версия дерева, так что ненужную память надо чистить. Заметим, что отрезки 1, 3, 5 нам не нужны, они использовались только для того, чтобы выделить нужные куски массива. Но заметим, что нельзя просто рекурсивно обойти деревья, соответствующие этим отрезкам, и их полностью удалить, так как часть вершин в этих деревьях действительно новые и ненужные, а часть — это те же вершины, что будут и в дереве, полученном `memmove`. Значит, надо каким-то более умным способом понимать, когда какая-то вершина нам не нужна и ее можно удалить.

Можно заменить указатели на `shared ptr`, тогда утверждается, что все будет корректно. Действительно, если на вершину больше нет никаких ссылок, то это, по сути, означает, что она не является ничьим ребенком и нигде в самой программе мы с ней больше не работаем, а, значит, ее можно спокойно удалить. Несложно понять, что и обратное тоже верно — если вершина не нужна больше, то на нее не должно быть никаких ссылок.

Можно взять и написать то же самое своими руками, чтобы это работало быстрее. Для каждой вершины будем хранить переменную, которая будет говорить нам сколько существует указателей, ссылающихся на данную вершину. Если на вершину ссылок больше нет, то мы ее будем удалять. Удаление вершины будет заключаться в уменьшении количества ссылок у ее детей на 1 и, собственно, в самом освобождении памяти, занимаемой этой вершиной.

```
void decreaseRefCount(Node *v) {
    if (v == Node::null)
        return;
    v->refCount--;
    if (v->refCount == 0) {
        decreaseRefCount(v->l);
        decreaseRefCount(v->r);
        delete v;
    }
}

Node* memmove(Node *array, int from, int to, int len) {
    Node *a1, *a2, *a3, *a4, *a5, *a6, *result;
    a1, a2, a3 = split(array, from, len);
    a4, a5, a6 = split(array, from, len);
    result = merge(a4, a2, a6);
    decreaseRefCount(array);
    decreaseRefCount(a1);
    decreaseRefCount(a2);
    decreaseRefCount(a3);
    decreaseRefCount(a4);
    decreaseRefCount(a5);
    decreaseRefCount(a6);
    return result;
}
```

В приведенной реализации мы предполагаем, что `split, merge` возвращают вершины, у которых значение счетчика равно 1. Заметим, что надо уменьшить количество ссылок и на `array`, так как нас интересует только результат операции, а то, что было до ее применения уже не важно.

3.7.3. Persistent Queue

Можно сразу же получить персистентную очередь, работающую за логарифм, сделав очередь на персистентном массиве. Но можно сделать персистентную очередь, которая будет работать за $O(1)$.

У нас есть персистентный стек, работающий за $O(1)$ (это, на самом деле, просто дерево). Казалось бы, раз мы умеем делать очередь на двух стеках, то можно сделать персистентную очередь на двух персистентных стеках, которая будет работать за $O(1)$. Но это, к сожалениию, не так.

Вспомним, как работала очередь на двух стеках. В первый стек мы клали элементы, из второго доставали. Когда второй стек опустошался, мы начинали доставать элементы из первого стека и добавлять их во второй стек, пока не переложим все элементы. Заметим, что это было не чистое $O(1)$, а амортизированное. А персистентность амортизацию ломает, так как можно взять конкретную операцию, которая будет работать долго, применить ее, после чего откатить структуру, снова повторить эту операцию, и так далее.

Попытаемся исправить эту проблему. Попробуем просто избавиться от амортизации. Как это делается? Будем брать и переворачивать стек не сразу, а постепенно, каждую операцию перекладывая какое-то константное число элементов. Назовем стек, из которого мы делаем `pop`, L, а стек, в который делаем `push`, R. Тогда если $|L| > |R|$, то мы ничего не делаем, а в тот момент, когда $|L| = |R|$, мы начнем переворачивать стек R. Проблема в том, что его нельзя переворачивать, так как мы должны еще иногда обрабатывать операции `push` в него. Тогда давайте создадим стек $R' = \emptyset$, сделаем `swap(R, R')`. Теперь R – пустой стек, в который мы будем делать `push`, а R' – его копия, которую мы будем переворачивать, кладя элементы в R_{rev}. Когда R' опустошится, просто удалим его.

Теперь у нас есть R_{rev} – перевернутая версия R', который теперь стал пустым и больше нам не нужен, есть стек R, в котором уже, возможно, лежат какие-то элементы, но это нас совершенно не интересует, и есть стек L. Если бы L был бы пуст, то мы просто сказали бы, что теперь мы будем делать `pop` из R_{rev}, и все было бы хорошо. Но мы могли не успеть этот стек опустошить, и в нем что-то может лежать. Это значит, что надо взять и поместить содержимое стека L поверх содержимого стека R_{rev}.

К сожалению, это так легко не делается. Но мы можем стек L перевернуть в стек L_{rev}, после чего начать доставать из L_{rev} элементы и класть их в R_{rev}. Поскольку стек L мы не можем менять, так как мы из него делаем операции `pop`, мы создаем стек $L' = L$, который будем переворачивать.

Сейчас алгоритм выглядит так: когда мы понимаем, что хотим перевернуть стек R, мы его меняем с пустым стеком R', создаем L' – копию L, достаем элементы из R', кладя их в R_{rev}, а из L' кладем в L_{rev}, после чего из стека L_{rev} будем доставать элементы и класть их в R_{rev}. Стоит заметить, что нам не важно, сколько побочных операций мы будем делать помимо операций `push, pop`, нам лишь хочется, чтобы это число было какой-то константой, не от чего не зависящей, эту константу мы установим чуть позже.

Но и это, еще не конечная версия алгоритма. Пока мы делали все эти операции, из стека L мы могли какие-то элементы достать, таким образом, в финальной версии R_{rev} у нас будут какие-то лишние элементы. Но заметим, что, элементы, которые мы с вершины стека L, в стеке L_{rev} находятся в самом конце. Тогда можно взять и просто завести переменную *popcount*, в которой будем хранить, сколько элементов из стека L мы успели достать, пока делали все эти операции, и последние *popcount* элементов из L_{rev} просто не будем доставать и класть в R_{rev}.

После того, как мы сконструировали корректный стек R_{rev}, мы его меняем местами со стеком L, который нам больше не нужен, и который можно будет удалить, после чего считаем, что переворачивание закончено.

Теперь надо аккуратно сформулировать, какие инварианты мы хотим поддерживать, и понять, сколько же именно операций надо делать каждый раз кроме `push, pop`.

Во-первых, надо понять, сколько дополнительных операций надо делать каждый раз, чтобы мы все успели перевернуть к моменту опустошения L. Всего нам надо сделать $|L|+|R|+|L|-\#\texttt{pop}$

операций, а, поскольку мы начинаем переворачивать стек в момент, когда $|L| = |R|$, то надо сделать не более $3|L|$ операций, то есть, по 3 дополнительных операции на каждую операцию `push, pop`.

Во-вторых, после того, как мы стек перевернем, должно выполняться $|R_{rev}| \geqslant |R|$, иначе получится, что мы уже опять должны были начать переворачивать стек, а мы этого еще не делали. $|R_{rev}| = |R_{old}| + |L_{old}| - \#\texttt{pop}, |R| = \#\texttt{push}$. Заметим, что $L_{old} = R_{old}$, так как мы начали переворачивать в момент, когда наступило равенство. Откуда $2|L_{old}| \geqslant \#\texttt{push} + \#\texttt{pop}$. Это значит, что количество операций, через которое мы полностью перевернем стек, не должно превышать $2L_{old}$. Но уже было показано, что если мы каждый раз будем делать 3 дополнительных операции, то мы сможем перевернуть стек за $|L_{old}|$ запросов к очереди, так что этот инвариант тоже будет соблюден.

Итого, у нас есть стеки L, R, из первого мы достаем элементы, во второй кладем, есть флажок $isReversing$, который показывает, переворачиваем мы сейчас стек R, или нет. Если мы его переворачиваем, то возникают дополнительные стеки L', R' – копии L, R на момент начала переворота, при этом стек R мы делаем пустым, стеки L_{rev}, R_{rev} – результат переворота L', R', переменная $popCount$, которая будет показывать, сколько раз мы сделали операцию `pop` за время переворота стека, то есть, сколько элементов со дна L_{rev} мы должны проигнорировать.

А вот так выглядит псевдокод работы с персистентной очередью.

```
void startReverse(){
    isReversing = 1;
    L' = new Stack();
    swap(L, L');
    R' = R; //такое присваивание в случае персистентного стека -- ОК
    R_rev = new Stack();
    L_rev = new Stack();
    popCount = 0;
}

void doSth(){
    if (R'->size())
        R_rev->push(R'->pop());
    if (L'->size())
        L_rev->push(L'->pop());
    if (L'->size() == 0 && R'->size() == 0)
        if (L_rev->size() > popCount())
            R_rev->push(L_rev->pop());
        else{
            isReversing = 0;
            swap(L, R_rev);
            delete R_rev;
            delete L_rev;
            delete R';
            delete L';
        }
}

void push(int a){
    R = R->push(a);
    if (L->size() == R->size())
        startReverse();
```

```
    FOR(i, 3)
        if (isReversing)
            doSth();
}

int pop(int a){
    int res = L->pop();
    if (L->size() == R->size())
        startReverse();
    FOR(i, 3)
        if (isReversing)
        {
            popCount++;
            doSth();
        }
}
```

Замечание 3.7.1. Функция `doSth` делает каждый раз либо одну, либо две дополнительных операции, поэтому, чтобы все работало корректно, ее надо вызвать 3 раза. Так же стоит заметить, что приведенная реализация, на самом деле, работает за амортизированное $O(1)$, так как `delete L_rev` будет работать за $O(popCount)$. Этого можно избежать, добавив в `doSth()` еще и простое опустошение стека L_{rev}, от этого может лишь несколько увеличиться константа, но зато очередь гарантированно будет работать за $O(1)$.

3.7.4. Persistent RBST

Хотим сделать персистентный `Treap`. Казалось бы, можно просто брать и каждый раз не менять вершину, а создавать ее копию и работать с ней. Но тогда у нас в какой-то момент может взять и сломаться балансировка. А именно, у нас могут появляться одинаковые y у вершин. Более того, если мы будем делать запросы вида `r=merge(r, r)` кучу раз, то таких вершин будет реально много. Пусть у нас была такая реализация `merge`:

```
Node *merge(Node *a, Node *b){
    if (a->y >= b->y) {
        a->r = merge(a->r, b);
        return a;
    } else {
        b->l = merge(a, b->l);
        return b;
    }
}
```

То после повторения в цикле такой операции: `r=merge(r, r)`, мы получим бамбук, направленный вправо. Пусть мы повторили эту операцию m раз, тогда размер дерева стал 2^m, и его глубина тоже стала 2^m вместо желаемой m.

Решение этой проблемы достаточно простое. Просто не будем хранить y в вершине. Вспомним, зачем нам нужны были случайные y. Мы говорили, что если все y будут случайными, то полученное дерево тоже будет случайным, то есть каждая вершина будет корнем с одинаковой вероятностью. Ну давайте это возьмем и реализуем.

```
Node *merge(Node *a, Node *b){
    if (a->size / (a->size + b->size) >= rand() / RAND_MAX) {
```

```
        a->r = merge(a->r, b);
        return a;
    } else {
        b->l = merge(a, b->l);
        return b;
    }
}
```

Рассмотрим все вершины, входящие в a и b. Мы хотим, чтобы корень был случайной вершиной. Тогда вероятность того, что корень будет лежать в a это как раз и есть $a->size/(a->size+b->size)$, то есть данный код нам как раз дает RBST.

Глава 4

Статистически оптимальное дерево поиска. Дерево отрезков

4.1. Статически оптимальное дерево

$\sum\limits_i k_i dep_i \to min$

$Time = O(n^2)$

Динамика по подотрезкам. Перебираем корень дерева, для префикса и суффикса динамика уже посчитана, получаем $O(n^3)$.

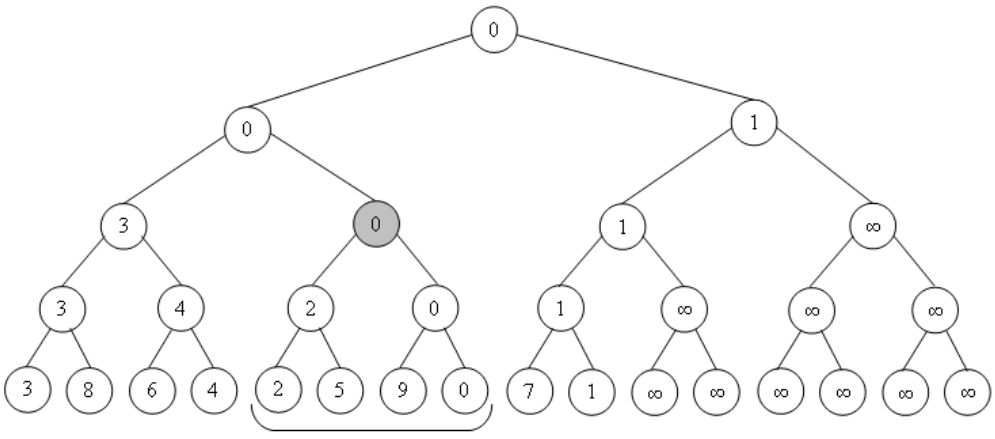
Обозначим оптимальный корень для отрезка $[L, R]$ за $p[L, R]$. Тогда верно следующее утверждение: $p[L, R-1] \leqslant p[L, R] \leqslant p[L+1, R]$. Доказывается примерно так же, как и аналогичное утверждение в конце прошлого семестра (см. задачу про выбор оптимальных точек на вещественной прямой).

$Time = \sum\limits_{L,R} p_{L,R} = \sum\limits_{L,R} (p_{L+1,R} - p_{L,R-1} + 1) = O(n^2) + \sum\limits_{\substack{\text{линейное} \\ \text{число} \\ \text{слагаемых}}} O(n) = O(n^2)$

4.2. Дерево отрезков

4.2.1. Описание и примеры применения

Описание структуры



В каждой вершине мы храним функцию от отрезка, за который эта вершина отвечает. Корень отвечает за отрезок $[0, N)$.

Высота такого дерева $\log_2 N$

$$V : [L, R) \Rightarrow \begin{cases} 2V : & [L, \frac{L+R}{2}) \\ 2V+1 : & [\frac{L+R}{2}, R) \end{cases}$$

Лемма 4.2.1. Заметим, что отрезок $[L, R]$ разбивается на $\leqslant 2\log_2 N$ вершин дерева отрезков.

▶ На одном уровне (которых всего $\log_2 N$) отмечено не более двух вершин. Пусть отмечено хотя бы три, тогда посмотрим на среднюю. Тогда можно было брать не её, а её родителя. Противоречие. ◀

Запросы Get и Update элемента

- Как отвечать на запросы `Update` элемента с помощью дерева отрезков?

  Нам достаточно изменить значения во всех вершинах на пути от корня до нашей вершины. Их количество равно высоте, то есть $O(\log_2 N)$.

- Как отвечать на запросы `Get` с помощью дерева отрезков?

  Пусть запрос `Get` на $[L, R]$. Разобьём отрезок $[L, R]$ на вершины дерева отрезков, в каждой из которых уже посчитан правильный ответ. Теперь осталось просто взять функцию по значениям в выбранных вершинах.

- Какие функции мы можем посчитать с помощью дерева отрезков?

  Несложно понять, что функция должна обладать свойством ассоциативности.

- Примеры функций:
  - $\sum$
  - min
  - gcd
  - Композиция перестановок
  - Произведение матриц

Запросы `Get` и `Update` на отрезке

Существует два вида запроса `Update` на отрезке: прибавление и присвоение. Решаются они одинаково: теперь мы храним в вершине дополнительное значение: сколько нужно прибавить или присвоить на этом отрезке. Когда заходим в вершину в процессе обработки запросов, проталкиваем значения в сыновей и обнуляем у себя.

Применение дерева отрезков в задаче «Художник»

Хотим уметь отвечать на два типа запросов:

1. `Paint` $[L, R]$ *color* — красим отрезок $[L, R]$ в цвет *color*.
2. `Get` — хотим узнать количество белых отрезков.

Будем хранить в вершине три дополнительных поля: *count*, *colorL* и *colorR* — количество белых отрезков, цвет левой клетки и цвет правой клетки соответственно. *colorL* и *colorR* пересчитываются просто, с *count* чуть сложнее:

```
1 count[v] = count[2 * v] + count[2 * v + 1]
2     - (colorR[2 * v] == WHITE && colorL[2 * v + 1] == WHITE)
```

`Get`: просто возвращаем *count*[1].

Динамическое дерево отрезков и сжатие координат

Хотим научиться поддерживать следующие три запроса:

1. `Insert(x)`
2. `Delete(x)`
3. `Count(L, R)` Количество таких x, что $L \leqslant x \leqslant R$.

Построим дерево отрезков по сумме на массиве $count[x]$ — количество чисел x в множестве. Запросы превращаются в прибавление ± 1 и сумму на отрезке. Осталась проблема с большими x.

Два пути решения:

1. Offline: Используем метод сжатия координат.

   $x_i \to position[x_i]$, где $position[j]$ — позиция j в отсортированном массиве x-ов.

   Таким образом, теперь все $x_i \leqslant N$.

2. Динамическое дерево отрезков. Вместо массива теперь пользуемся структурой, как в Treap. Есть указатели на левого и правого сыновей, причём создаём их, только если запрос спускается в соответствующую ветку.

$$Time = O(N \log M)$$
$$Memory = O(N \log \frac{M}{N})$$

   $\log \frac{M}{N}$ появляется из-за того, что во многих запросах будет одинаковое начало — первые $\log N$ уровней, поэтому на них выделять память каждый раз заново мы не будем.

Количество вершин в дереве отрезков

Реализация массивом: $4N$
Реализация структурой: $(2N-1)\text{sizeof}(node)$

4.2.2. k-ая порядковая статистика на отрезке

Нет запросов изменения

Научимся отвечать на вопрос: сколько чисел на $[L, R] \leqslant x$.

В дереве отрезков храним отсортированные кусочки массива. Умеем строить за $O(N \log N)$ — `merge()` сыновей. Пусть нам пришел запрос, разбили отрезок на вершины дерево отрезков, в каждой вершине сделали бинпоиск.

$$Time = O(\log^2 N)$$
$$Memory = O(N \log N)$$

Есть запросы изменения

Вместо кусочков массивов храним дерево отрезков

- BST. Добавление и удаление за $O(\log^2 N)$: на всём пути до корня вставляем/удаляем элемент из очередного BST.

  Запрос: сколько чисел на $[L, R] \leqslant x$ за $O(\log^2 N)$: разбиваем отрезок на вершины дерева отрезков, в каждом BST за $O(\log N)$ отвечаем на запрос, храня количество вершин в поддереве и просто спускаясь по BST.

  Time $= O(\log^2 N)$

- Динамическое дерево отрезков. Храним динамическое дерево отрезков $count[x]$ по сумме — сколько раз встречалось число x.

  Добавление и удаление за $O(\log N \log M)$: на всём пути до корня вставляем/удаляем элемент из очередного динамического дерева отрезков.

  Запрос: сколько чисел на $[L, R] \leqslant x$ за $O(\log N \log M)$: разбиваем отрезок на вершины дерева отрезков, в каждом динамическом дереве отрезков за $O(\log M)$ делаем запрос суммы на $[0, x]$.

  Time $= O(\log N \log M)$

После этого сделаем внешний бинпоиск по ответу.

Нет запросов изменения, $O(\log M \log N)$

Хотим уметь отвечать на запросы вида $L \leqslant i \leqslant R$, $a_i \leqslant x$ за $O(\log N)$.

Заметим, что когда мы спускаемся по дереву отрезков, нам не нужно делать в каждой вершине бинпоиск заново. Для этого будем хранить величину $L[i]$ — сколько среди первых i чисел в вершине попали в левого сына. Тогда если бинпоиск вернул нам позицию pos, то нам достаточно перейти в левого сына с позицией $L[pos]$, а в правого — с позицией $pos - L[pos]$.

Строим сверху вниз: смотрим на индекс, кидаем в нужного сына.

Итого: сделали один бинпоиск в корне дерева, быстро спустились по дереву отрезков с помощью $L[i]$.

$$Time = O(\log N)$$

Достаточно хранить отсортированный массив в корне и ссылки $\Rightarrow$ память увеличилась на N.

Вспомним, что мы хотим уметь считать k-ую порядковую статистику на отрезке.

Пусть $cnt(x)$ — количество таких a_i, что $L \leqslant i \leqslant R$, $a_i \leqslant x$. Заметим, что $cnt(x)$ возрастает. Значит, можем сделать бинпоиск по ответу.

Ответ — $\min x : cnt(x) \geqslant k$.

$$Time = O(\log M \log N)$$

Оптимизация $\log M \Rightarrow \log N$

Если запросов изменения нет, то всего N элементов, можно делать бинпоиск по ним, а не по значениям.

Если запросы изменения есть, то мы можем хранить все элементы в отдельном Treap и спускаться «бинпоиском» по Treap-у, используя опорным элементов в бинпоиске текущую вершину.

Нет запросов изменения, $O(\log^2 N)$

Пусть для первых i ячеек мы знаем дерево отрезков по сумме $count_i[x]$ — сколько раз встречалось число x. Обозначим это дерево за $Tree_i$. Заметим, что $count_i$ от $count_{i-1}$ отличается всего одним изменением (в ячейке a_i). Значит, мы можем сделать наше дерево отрезков персистентным.

Хотим уметь отвечать на запросы вида $L \leqslant i \leqslant R$, $a_i \leqslant x$ за $O(\log N)$.

Ответ — `Tree[R].get(0, x) - Tree[L-1].get(0, x)`.

Внешне всё еще бинпоиск.

$$Time = O(\log^2 N)$$

Нет запросов изменения, $O(\log N)$

Заметим, что мы можем спускаться по деревьям параллельно. Изначально мы стоим в корнях деревьев $Tree_R$ и $Tree_{L-1}$.

Пусть сумма в левом сыне $Tree_R$ равна A, а сумма в левом сыне $Tree_{L-1}$ равна B (вспомним, что мы строили эти деревья именно по сумме). Тогда если $A-B \geqslant k$, то мы идем в левых сыновей, иначе в правых, не забывая сделать `k -= A - B`.

Вершина, в которую придём в итоге, и есть ответ.

$$Time = O(\log N)$$

Есть запросы изменения, $O(\log N \log M)$

Структура: деревья отрезков динамических деревьев отрезков.

Оптимизация аналогична предыдущей: вместо внешнего бинпоиска и спуска по дереву отрезков, мы разбиваем запрос на вершины дерева отрезков, а дальше аналогичный трюк. Суммируем суммы по все левым сыновьям вершин, на которые разбился запрос. Если эта сумма $S \leqslant k$, во всех деревьях спускаемся в левого сына, иначе в правого, не забывая сделать `k -= S`.

$$Time = O(\log N \log M)$$

4.3. Дерево отрезков снизу.

```
for (L+=n, R+=n; L <= Rl L/=2, R/=2)
{
    if (L%2 == 1) sum += get(L++);
    if (R%2 == 0) sum += get(R--);
}
```

За каждую итерацию цикла рассматриваемый отрезок становится $\geqslant$ в 2 раза короче $\Rightarrow$ цикл сделает $O(\log(R-L))$ операций.

На каждой итерации цикла L и R - отрезок вершин, которые нужно обработать.

Переход от запросов сверху к запросам снизу:

1. Одномерное: $\log n \to \log(R-L)$
2. Двумерное: $\log^2 n \to \log(R-L) \cdot BS + \log n \to \log(R-L) \cdot BS + \log(R-L)$

«+» дерева отрезков снизу:

- Нет рекурсии
- $O(\log(R-L))$

Задачи.

Можно сопоставить массиву точки: (i, a_i).

Тогда, чтобы найти решение такой задачи: $\begin{cases} L \leqslant i \leqslant R \\ X \leqslant a_i \leqslant Y \end{cases}$,

достаточно найти точку в прямоугольнике. Для того, чтобы из точек получить массив, надо определить их порядок. Есть несколько способов.

1. $P(p_x, p_y), Q(q_x, q_y)$ Можно задать два отношения:
   - $P <_1 Q$, если $((p_x < q_x) || (p_x == q_x \&\& p_y < q_y))$

- $P <_2 Q$, если $p_y < q_y$

Тогда то, что точка P лежит в прямоугольнике равносильно следующему:

$$\begin{cases} A \leqslant_1 P \leqslant_1 B \\ A \leqslant_2 P \leqslant_2 B \end{cases}$$

2. Sort по x-координате, забивая на одинаковые. Тогда условие $L_x \leqslant x \leqslant R_x$ превратиться в условие на индекс $lower\_bound(L_x) \leqslant i \leqslant upper\_bound(R_x)$.

Для многомерной задачи дерево отрезков нужно хранить в дереве отрезков. $x_1, \dots, x_d$ и условия $L_i \leqslant x_i \leqslant R_i$. Одним деревом отрезков мы понижаем размерность на 1: $d \to d-1$

Sort x_d. Строим на массиве дерево отрезков. В каждой вершине дерева отрезков, соответсвующей $[L..R]$, хранится ещё одно дерево отрезков на точках, хранящихся с индексами с L по R на одну размерность меньше.

Как посчитать ответ: BS(находим L и R индексы в массиве) + Д.О. + Запросы к соответсвующим вершинам.

Д.О. $\to$ Д.О. $\to$ ... $\to$ Д.О. $\to$ массив. Каждый переход осуществляется за $O(\log n)$, но мы умеем последние два перехода делать за $O(\log n)$ при запросах.

Итог:

- $O(n \log^{d-1} n)$ памяти
- $O(\log^d n)$ Change
- $O(\log^{d-1} n)$ Count, Sum, Min

4.4. Дерево Фенвика

```
get(i) { //sum[0, i], $O(\log n)$
        sum = 0;
        for (; i >= 0; i &= (i + 1), i--)
            sum += a[i];
    return sum;
}
```

```
add(i, x){
    for (; i < n; i |= (i + 1))
        a[i] += x;
}
```

«+» Дерева Фенвика:

- Код короткий.
- Нет дополнительной памяти(Inplace).
- Константа меньше.

В ячейке массива $a[i]$ хранится сумма чисел с индексами $[i\&(i+1) \dots i]$

Корректность дерева Фенвика.

- Корректность add.
  1. Для любого i, учатсвующего в for: $i\&(i+1) \leqslant i_0 \leqslant i$, то есть прибавляем только к тому, к чему надо.
  2. Все нужные числа переберём.
- Коррекнтность get.

Ещё один «+» дерева Фенвика: 2D-деревья.

$O(\log^2 n)$

```
get(i_0, j_0) { // sum [0, i_0][0, j_0]
    sum = 0;
    for (i = i_0; i >= 0; i &= (i + 1), i--)
        for (j = j_0; j >= 0; j &= (j + 1), j--)
            sum += a[i][j];
    return sum;
}
```

```
add(i_0, j_0, x) {
    for (i = i_0; i < n; i |= (i + 1))
        for (j = j_0; j < n; j |= (j + 1))
            a[i][j] += x;
    return sum;
}
```

$\sum(L \dots R]$ =get(R) - get(L) одномерный случай.
$\sum(x_1, x_2] \times (y_1, y_2]$ =get(x_2, y_2) - get(x_1, y_2) - get(x_2, y_1) + get(x_1, y_1) двумерный случай
k-мерное: $(\log n)^k 2^k = O((\log n)^k)$. ($2^k$ - количество слагаемых)

4.5. SkanLine

1. n точек, m прямоугольников. Для каждого прямоугольника $\sum$ значений точек/количество точек. За $O((n+m)\log(n+m))$
   (a) Сжатие координат: $0 \leqslant x, y \leqslant n + 2m$
   (b) Переходим к полоскам(стаканам: $[-\inf; x] \times [y_1; y_2]$). Прямоугольник: 1 полоска - 2 полоска.
   (c) ScanLine: идём слева направо по x(Sort Events по x). Дерево отрезков или Фенвик по y.
   События:
   1) Встретили точку с координатой y: Add(y);
   2) sum(y_1, y_2);
   Приоритет: Add < Sum.
2. Последовательность точек, возрастающая по обеим координатам, длиннейшая.
   f[i] - максимальное количество точек у последовательности с концом в этой точке.
   f[i] = $\max\limits_{y_j < y_i} f[j] + 1$; ← Д.О. с max.
   PS: есть решение с динамикой + BS.

4.6. Задачки про ScanLine

1. Условие: Найти площадь объединения прямоугольников.

   Решение: Заведем события двух типов: прямоугольник начался, прямоугольник закончился. Отсортируем события по x координате. Также у нас будет дерево отрезков по y, где в вершинах мы будем хранить минимум в поддереве и количество раз, которое он встречается. Будем обрабатывать события слева на право. Если начался новый прямоугольник, то прибавим к соответствующему отрезку в дереве отрезков 1. Если прямоугольник закончился, то посмотрим чему равен минимум в дереве. Если он равен 0, то прибавим к ответу $count \cdot (x_i - x_{i-1})$, где $count$ — это количество нулей в дереве, а $x_i - x_{i-1}$ — это расстояние между двумя последними обработанными событиями.

2. Условие: Вам даны взвешенные точки, нужно находить сумму/минимум/gcd в стакане.

   Напоминание: стакан задается тремя числами: L, R, M. Точка (x, y) лежит в стакане, если $L \leqslant y \leqslant R$ и $x \leqslant M$.

   Решение: Сжимаем координаты по y. Соритруем запросы по x координате. Заводим дерево отрезков по y. Обрабатываем запросы слева на право. Задачка решена!

Глава 5

Sparse table, Disjoint sparse table, Фарах-Колтон-Бендер

5.1. Sparse table

5.1.1. Основное

Sparse table — структура данных, которая умеет считать идемпотентные функции на отрезке. Для каждой позиции i в массиве будем считать функцию на отрезках вида $[i, 2^k)$.

5.1.2. Реализация

```
// построение за O(n log n)
// запрос за O(1)

void build(int* data, size_t size) {
    for (int i = 0; i < size; ++i)
        table[i][0] = data[i];
    for (int k = 1; k <= LOG_MAX_SIZE; ++k)
        for (int i = 0; i < size; ++i)
            table[i][k] = min(table[i][k - 1], table[i + (1 << (k - 1))][k - 1]);
}

int getMin(int l, int r) {
    int k = Log[r - l + 1]; // Log[x] - максимальная степень 2, которая не превосходит x
    return min(table[l][k], table[r - (1 << k) + 1][k]);
}
```

5.2. Disjoint sparse table

Добьем массив нулям так, что бы его размер стал степенью двойки. Разобьем его на отрезки вида $[i, \frac{n}{2}](i \leqslant \frac{n}{2})$ и $[\frac{n}{2}+1, i](i \geqslant \frac{n}{2}+1)$. Построим туже структуру для левой и правой частей массива. Посчитаем нужную функцию на этих отрезках.

Очевидно, что построение работает за $O(n \log n)$. Также понятно, что мы можем за $O(1)$ отвечать на запрос на отрезке, если мы знаем позицию самого высоко разреза (позиции, относительно которой массив был когда-то разделен на две части), которая попадает на наш отрезок. Чтобы научиться быстро находить такую позицию для соответствующего отрезка $[L, R]$, надо понять, что

её можно задать максимально совпадающим префиксом чисел L и R в двоичной записи. Чтобы его получить, нужно будет предподсчетать старший бит для чисел.

5.3. Модификации Sparse table

1. Построение за $O(n)$, запрос за $O(\log n)$.

   Разобьем массив на отрезки размера $\log n$, посчитаем на них функцию и построим sparse table.

2. Построение за $O(n \log \log n)$, запрос за $O(1)$.

   Разобьем массив на отрезки размера $\log n$, посчитаем на них функцию и построим sparse table. Также построим sparse table на каждом из отрезков.

3. Построение за $O(n)$, запрос за $O(\log \log n)$.

   Разобьем массив на отрезки размера $\log n$ и построим на них sparse table. Также на каждом отрезке построим дерево отрезков.

5.4. RMQ ± 1

Построим новый массив d, где $d[i] = 0$, если $a[i]+1 = a[i-1]$, и $d[i] = 1$, если $a[i]-1 = a[i-1]$ (a — исходный массив). Разобьем этот массив на отрезки размера $k = \frac{\log n}{2}$ и построим на них sparse table. Каждый отрезок начинается либо с 0 либо с 1, а остальных возможных суффиксов всего 2^{k-1}. Поэтому предподсчетаем позицию минимума в каждом суффиксе суммарно за $2^k k^2 = \theta(\sqrt{n} \log^2 n)$.

Теперь мы умеем получать минимум на отрезке за $O(1)$ и построением за $O(n)$.

5.5. Фарах-Колтон-Бендер

Хотим уметь получать минимум на отрезке за $O(1)$ с построением за $O(n)$.

5.5.1. Сведение RMQ к LCA

```
int getMin(int l, int r) {
    int vL = getVertex(l); // vL - вершина в дереве
    int vR = getVertex(r); // vR - вершина в дереве
    int lca = getLca(vL, vR); // получаем LCA двух вершин
    return index[lca]; // позиция элемента в массиве, который соответствует вершине lca
}
```

Заметим, что $l \leqslant index[lca] \leqslant r$, ведь иначе вершины vL и vR лежали бы в одном сыне lca, а также $a[index[lca]]$ — минимальный элемент на отрезке $[l, r]$ (по построению декартового дерева).

5.5.2. Ребята, давайте построим дерево

Построим за $O(n)$ декартово дерево, в котором ключами будут индексы элементов, а ключом — значения.

```
for (int i = 0; i < n; ++i) {
    Node* last = NULL;
    Node* node = new Node(i, a[i], NULL, NULL);
```

```
    while (!path.empty() && path.top()->y > node->y)
        last = path.pop();
    node->left = last;
    path.push(node);
}
```

5.5.3. Сведение LCA к RMQ $\pm$ 1

Построим эйлеров обход для нашего дерева, каждый раз выписывая номер вершины кода в нее входим. Теперь, что бы найти наименьшего общего предка для вершин a и b, нужно найти вершину с минимальной глубиной, которая находиться в обходе между любыми вхождениями вершин a и b.

Легко заметить, что глубины подряд идущих вершин в обходе отличаются на 1. Поэтому мы можем свести задачу к RMQ $\pm$ 1.

```
int getLca(int a, int b) {
    int i = RMQ(h, pos[a], pos[b]); // pos[x] - позиция вершины x в обходе дерева
    return v[i]; // вершина с индексом i  в обходе
}
```

Глава 6

Функции на путях

Примеры функций: сумма, минимум.

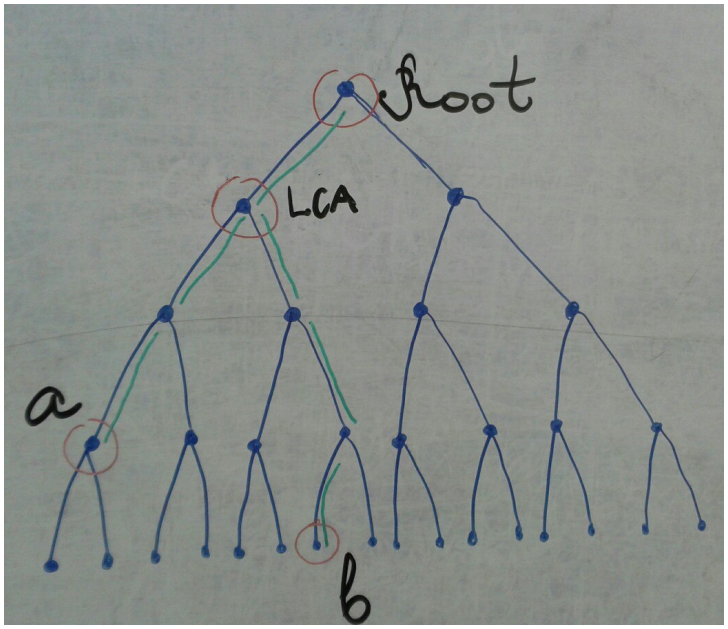
Путь: ребра от a до b, где a и b — вершины дерева.

Варианты: сумма весов вершин или сумма весов ребер. Достаточно научиться считать для ребер и построить биекцию между ребрами и вершинами. Биекция: каждой вершине сопоставляем ребро в отца. Для корня можем сделать фиктивное ребро.

6.1. Сумма на пути, дерево не меняется. $T = O(1)$

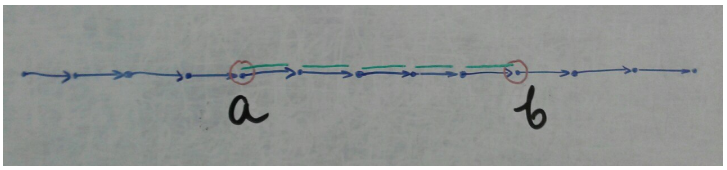
Сумма на пути по вершинам.

1. Ищем LCA за $O(1)$.
2. Используем массив `s` — массив префиксных сумм.
3. Ответ будет таким: `s[a] + s[b] - 2*s[LCA]`



Замечание 6.1.1. Когда мы пытаемся что-то сделать на дереве, сначала надо попытаться провести аналогию с массивом (частным случаем дерева).

Замечание 6.1.2. Любой путь первым делом разбиваем на два вертикальных отрезка.



6.2. Сумма на пути, дерево меняется

Запросы:

1. `get(a, b)` — найти функцию на пути из a в b.
2. `w[e] = x` — изменить вес ребра e на x.

Решение: Используем Эйлеров обход с ориентированными ребрами (каждое в двух экземплярах: $+w$, если спускаемся, $-w$, если поднимаемся).

Тогда суммой на пути будет значение от любого вхождения LCA до любого вхождения a. Все ненужные ребра сократятся, так как если сворачивали с кратчайшего пути, то обязательно возвращались обратно. Однако, обычно берут первые вхождения, тогда и сумма получается с нужным знаком.

Чтобы изменить ребро, необходимо изменить его вес в Эйлеровом обходе.

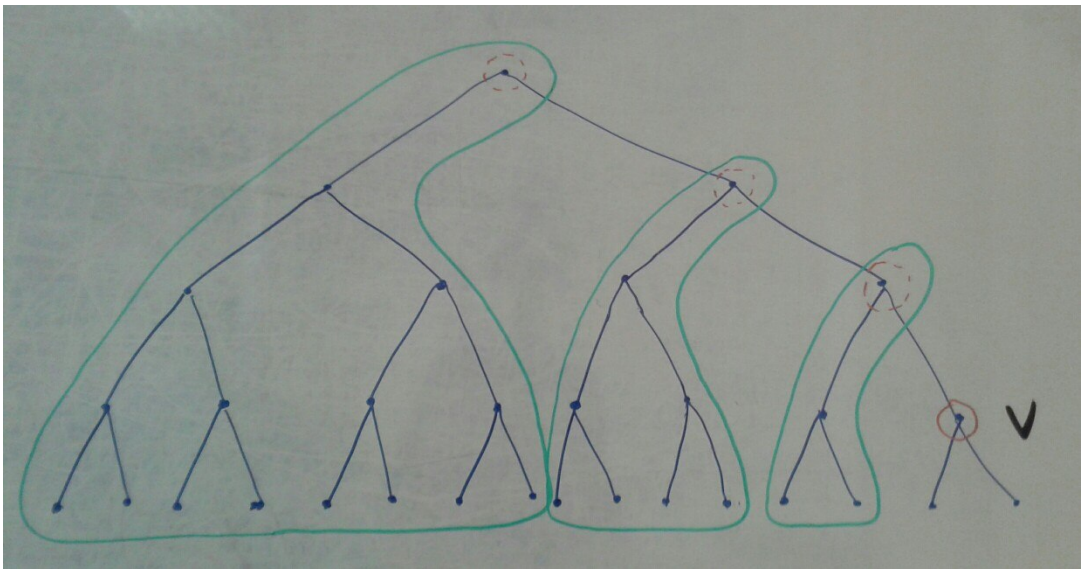
Таким образом, обе операции работают за $O(\log n)$.

Код построения Эйлерова обхода:

```
go(v) {
    pos[v] = n; // сохраняем позицию вершины в обходе (первое вхождение)
    for (auto e: edges[v]) { // по всем ребрам из v, идущим в x
        first[e] = n;
        a[n++] = w(e); // в a храним обход
        go(x);
        second[e] = n;
        a[n++] = -w(e);
    }
}
```

6.3. LCA-Offline. $T = O\left(A^{-1}(m,n)\right) = O^*(1)$

Будем использовать СНМ вершин, которые уже обошли, и множество вершин от корня до текущей. Таким образом, из любой вершины, из которой уже вышли, легко посчитать LCA с вершиной v (стоим в ней сейчас).




```
void go(v) {
    color[v] = 1; // для СНМ
    for (int a: List[v]) // список вопросов, связанных с v
        if (color[a] == 1)
            LCA = get(a); // возвращает представителя
    for (int x: edges[v]) { // для всех соседей $v$
        go(x)
        join(x, v)
    }
}
```

6.4. RMQ-Offline (минимум на отрезке)

Способ 1: Сводим RMQ к LCA, решаем LCA Offline

Способ 2:
1. Перебираем правую границу в порядке возрастания (левая — произвольная)
2. На всем отрезке есть минимальный элемент (с индексом m_i)
3. Ответом будет m_i: $i = min$: $L < m_i$ Пояснение:

$$a[m_1] \leqslant a[m_2] \leqslant \ldots; m_1 = \min_{[} 1; R], m_{i+1} = \min_{[} m_i + 1; R]$$

Если храним отрезки в СНМ, то ответ — get к СНМ. Представитель: самый правый элемент в отрезке.

```
for (i = 0; i < q; ++i) // запросы
    List[R[i]].push_back(L[i]);
for (i = 0; i < n; ++i) // запросы
    p[i] = i; // Init DSU
stack<int> m;
for (int r(0); r < n; ++r)
    while (m.size() > 0 && a[get(m.top())] >= a[R])
        p[get(m.top())] = R;
        m.pop();
    m.push(R);
```

```
    for (int i: List[R])
        min[i] = get(L[i]);
//...
int get(v) {
    return (v == p[v]) ? v : (p[v] = get(p[v]));
}
```

6.5. Рандомизированный алгоритм построения MST за линейное время

Вспомним алгоритм Борувки (он умеет за линию в два раза уменьшать число вершин) и применим три раза. Тогда, из имевшихся (n, m) получим $(\frac{n}{8}, m)$. После этого возьмем случайную половину ребер и запустимся от этого рекурсивно.

Скажем, что ребро (a, b) потенциально хорошее, если вес этого ребра меньше, чем максимальное ребро на пути из a в b в полученном остовном дереве (то есть при замене мы можем что-то улучшить), или если эти вершины лежат в разных компонентах связности.

Теперь, для ребра надо быстро определять является ли оно хорошим. Другими словами, найти $\max\{a, b\}$. Для этого используем максимум на пути за $O(1)$ в Offline.

Докажем линейность времени работы. Когда переходим в рекурсию, у нас остается $\frac{n}{8}$ вершин, $\frac{n}{8}$ из остовного дерева и $\frac{n}{8}$ хороших (об этом будет сказано позже) ребер.

$$T(n, m) = O(m) + T\left(\frac{n}{8}, \frac{m}{2}\right) + T\left(\frac{n}{8}, \frac{n}{4}\right)$$

Последнее слагаемое — вызов от остова + "хорошие ребра"

По индукции докажем, что $T = O(m)$. Для этого заменяем $O(m)$ на cm и выносим c за скобку:

$$T(n, m) = m + 2\left(\frac{n}{8} + \frac{m}{2}\right) + 2\left(\frac{n}{8} + \frac{n}{4}\right) = n + 2m \leqslant 2(n + m)$$

Чтобы оценить количество хороших ребер, докажем следующую лемму (в нашей задаче параметр p будет равен $\frac{1}{2}$):

Лемма 6.5.1. Если H — подграф G такой, что любое ребро взято с вероятностью p, то в G E[количество хороших ребер относительно MST H] $\leqslant (n-1)(\frac{1}{p} - 1)$

▶ Хорошее ребро — такое ребро, которое добавил бы алгоритм Краскала от G, следовательно таких событий (добавлений ребра) $n - 1$. С вероятностью p мы добавим его в H, значит с вероятностью $1 - p$ оно станет хорошим, значит, нужно оценить количество таких событий.

Оценим: $E = 1 + (1 - p)E =$ количество хороших ребер до остовного + остовное ◀

```
sort(edges.begin(), sort.end())
for (edge e: edges) {
    if get(a[e]) == get(b[e]) //это условие выполнится $\frac{n - 1}{p}$ раз
        if (p) // с вероятностью $p$
            Join(a[e], b[e]) // $n - 1$
        else
            GoodEdges.add(e) // $\frac{n - 1}{p} - (n - 1)$
}
```

Замечание 6.5.1. Если алгоритм Краскала не пытался включить e в остов ребер H, то в минимальном остове ребер G этого ребра тоже не будет.

Замечание 6.5.2. Рекурсия обрывается, когда остается одна вершина (сама себе остов).

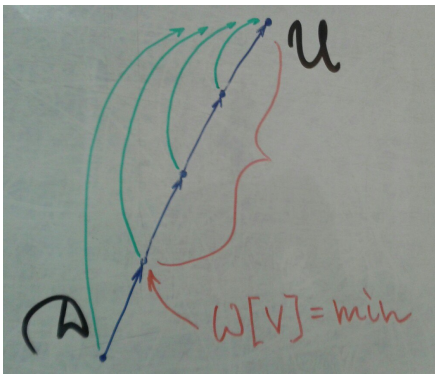
Итоговое время работы:

Среднее: $O(n+m)$

Худшее: $\min\{O(n^2), O(m\log n)\}$ (если осталось $\frac{m}{2}$ хороших ребер, но от кратных ребер избавляемся, значит, на каждом шаге не больше n^2 ребер)

6.6. Min-offline $T = O^*(1)$

Пути вертикальны, значит перебираем в порядке возрастания верхних границ, новый вес вершины — минимальный на соответствующем пути. Применяем эвристику сжатия путей, получаем $O(\log n)$. Преобразования корректны, так как u возрастает.

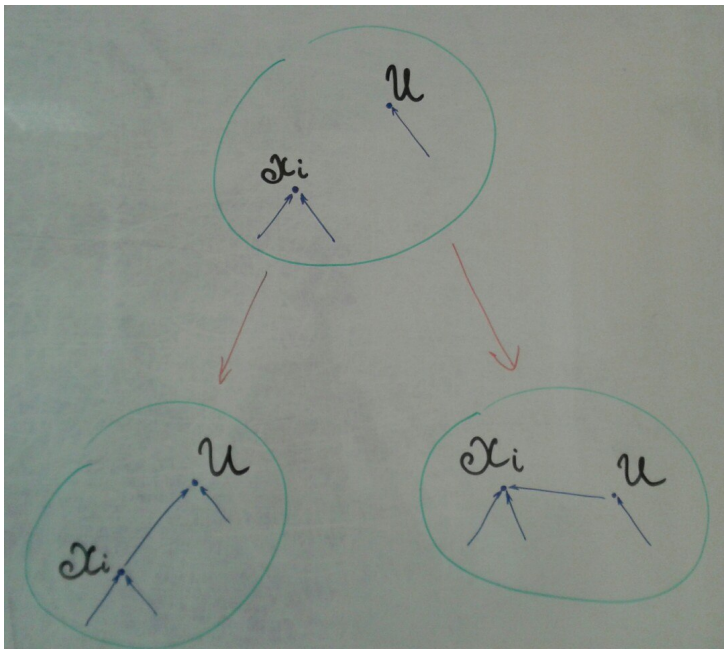



```
for u in ascending // верхняя граница запроса
    for x: p[x] = u
        Join(x, u)
    for d in ascending // нижняя граница запроса
        ans = get(d)
```

Теперь о подвешивании (`Join`): если $w[u] \leqslant w[x]$ может и так, и так подвесить. Чтобы минимумы остались теми же, надо будет `w[x] = w[u]`.

1. $p[x] = u$ — подвешивание оригинальное, не меняет дерева.
2. $p[u] = x$ — альтернативное подвешивание, меняет дерево.

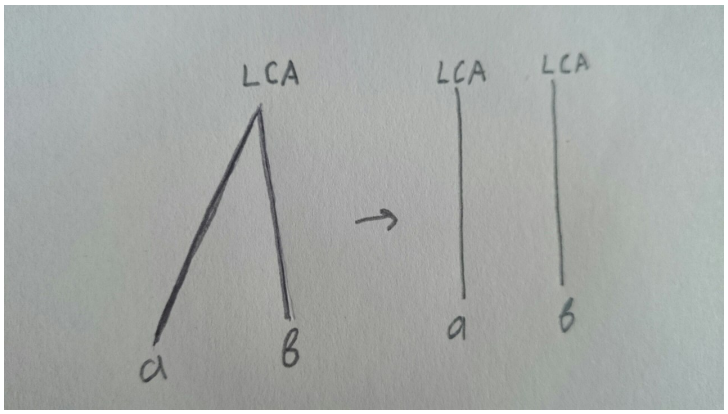
Если же $w[u] > w[x]$, то для поддерева x ответ должен быть меньше и подвешивать надо по-особенному, подробно расскать об этом не успели.



6.7. Максимум на пути за $O^*(1)$

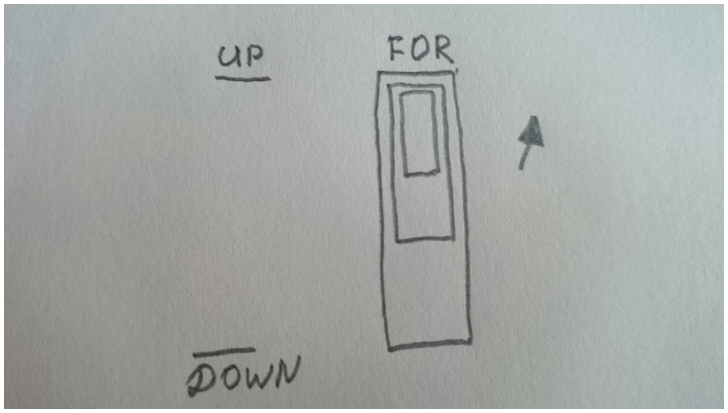
Более подробный рассказ уже описанного алгоритма.

Алгоритм работает offline за $O((n+m)A^{-1}(n,m))$, где n — количество вершин, m — количество запросов. Хотим применить сжатие путей, но перед этим нужно понять, что все пути вертикальны.



Делаем обход из нижней вершины к верхней, таким образом узнаем максимумы на всех путях от вершин до представителя, для этого потребовалось $O((m+n)log(n))$ (из-за эвристики сжатия путей). То есть, когда поступает запрос максимума на пути, перевешиваем все вершины к представителю и записываем на вновь появившемся ребре максимум.

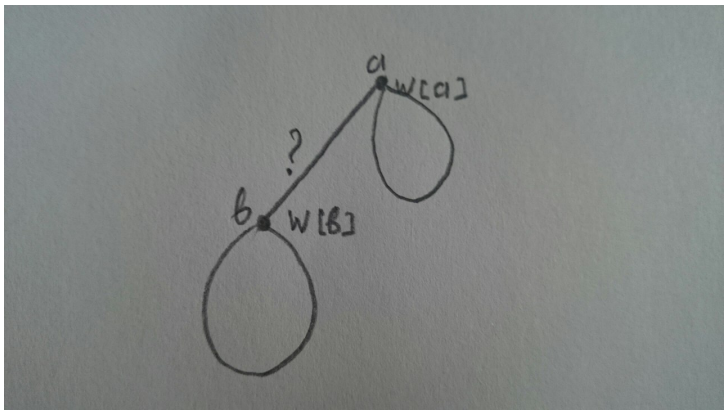
Чтобы улучшить асимптотику, нужно улучшить ранговую эвристику (подвешивания меньшего к большему).




```
for UP
    for queries[up]
        Get[down]
Join(parent[up], up)
```

Тепрь рассмотрим как должен работать Join

`w[a]` — максимум, который достигается из вершины сейчас.



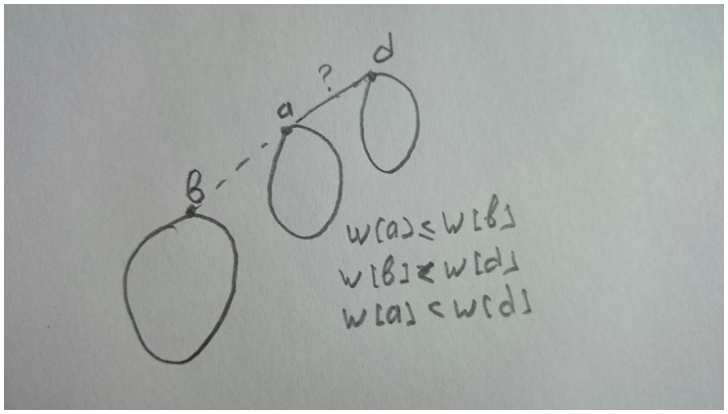
Join(a, b)

1. $w[a] \geqslant w[b]$: Нужно переподвесить. При этом переподвешиваем с учетом ранговой эвристики. В представителя записываем значения $w[a]$.

2. $w[a] < w[b]$: Ничего не надо делать. (*Ну почти ничего...*) Все вершины, которые проходили через b, поднимаются выше, max не изменится.

Рассмотрим более подробно второй случай. Пусть сейчас есть подмножество вершин b и подмножество вершин a. При рассмотрение «нужно ли подвесть b к a» - ничего не меняем.

Но может появится вершина d которая больше, чем a и b. Тогда для обоих подмножеств нудно изменить ответ.

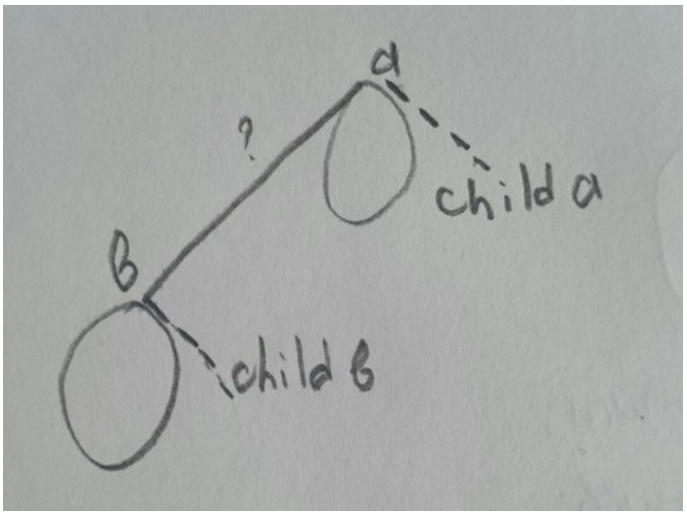


Избежим пересчитывания, введя «сына». Сын у вершины будет только 1. И для сына b вершины a верно, что $w[b] > w[a]$.

Вспомним возникающие случаи.

1. $w[a] \geqslant w[b]$: Переподвешиваем, теперь ответ для вершины b увеличился, стал $w[a]$. Значит теперь могло стать неверным $w[b] < w[child]$. Если $w[child]$ стало больше, то объеденим с родителем. Повотрим, пока не дойдем до конца дерева, или пока условие на ребенка не станет верным.

2. $w[a] < w[b]$: Мы хотим сказать, что вершина b теперь стала ребенком вершины a, но что делать если вершина a уже имела сына?

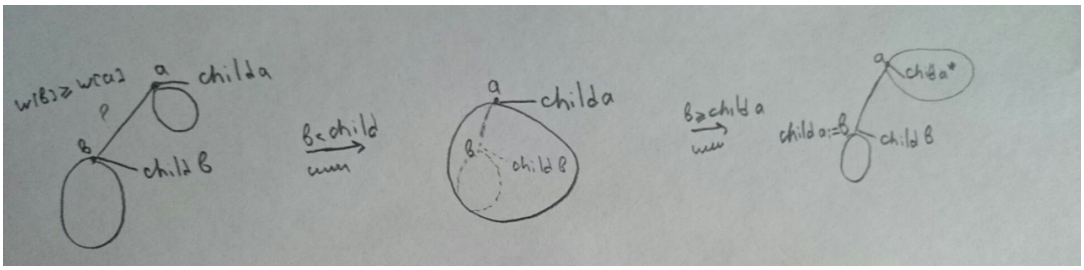


Чтобы улучшить асимптотику — выбираем меньшего из b и ребёнка a, и объединяем с родителем. Для этого храним *size* - настоящий размер, с учетом отложеных *Join*.

```
Merge(a, b) { // допустим b меньше ребенка a.
    DSU.parent[b] = a; //сливаем a и b.
    Merge(a, child[b]); //сливаем a с ребенком b.
```

То есть раньше мы бы сделали три объединения: a со своим ребенком, b со своим, и потом подвесели бы b к a. А сейчас это похоже на ранговую эвристику — берем меньшего.

Замечание 6.7.1. Все это для того, чтобы подниматься и искать максимум. *Child* добавляется когда у нижней вершины вес меньше, чем у верхней. При объединении двух отцов с детьми, объединяем однонго из них с ребенкои, того у которого размер ребенка меньше. И делаем *Join*.



Можно для большего улучшения времени объединения делать следующим образом:

```
while ((B != null) && (C != null))
    let B <= C
    Join(A, B)
    B = child B
```

где C — ребенок A, то есть таким образом мы сможем объеденить мимнмальное число врешин.

Мы верим, что эвристика сжатия путей, работает. Теперь попробуем понять, почему работает ранговая эвристика.

Заметим, что, когда мы разрешаем ребро child, размер дерева хотя бы в два раза больше размера поддерева.

Когда мы проводим ребро просто так, мы это делаем в зависимости от ранга, значит, ранговая эвристика работает.

Глава 7

Euler Tour Trees, Heavy-Light Decompostion

7.1. Euler Tour Trees

7.1.1. Определение

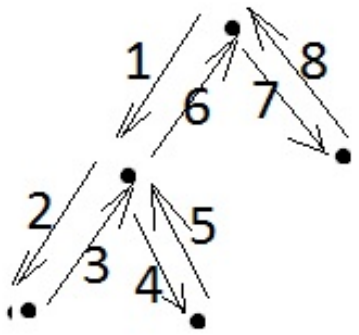
Euler Tour Trees — это структура данных, которая в лесу обрабатывает запросы

1. `Link(a, b)` — провести ребро
2. `Cut(a, b)` — удалить ребро
3. `IsConnected(a, b)` — проверить, что две вершины лежат в одной компоненте

Все операции за $O(\log n)$.

7.1.2. Что нужно хранить

Для реализации этой структуры будем хранить Эйлеров обход по ребрам в декартовом дереве по неявному ключу.



Так же необходимо хранить:

1. Для каждой вершины множество ребер(`set<edge> edges[v]`)
2. Для всех ребер хранить обратное(`rev[e]`, но можно и без них)
3. Для ребра ссылку на узел в декартовом дереве.(`node[e]`)
4. В узле декартова дерева хранить указатель на правого и левого ребенка и на предка

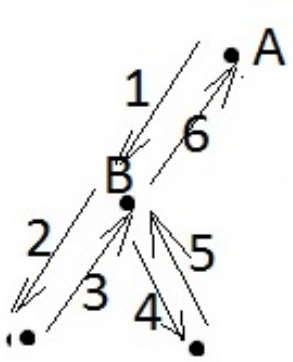
7.1.3. Реализация

`Get(v)` — узнать, в какой компоненте вершина

Вершина → любое выходящее ребро → `node*` → корень дерева.

1. Для вершины хранится `set<edge>`, выбираем любое ребро
2. По ребру переходим в узел в декартовом дерве
3. Поднимаемся по предкам к корню дерева.

`MakeRoot(v)` — сделать корнем вершину



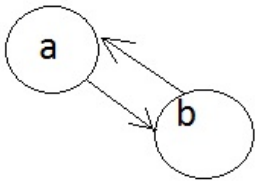
Если корень вершина A, то эйлеров обход выглдяит так: 1 2 3 4 5 6. Если корень вершина B: 4 5 6 1 2 3. Для того, что бы сделать корнем другую вершину, нужно просто сделать циклический сдвиг.

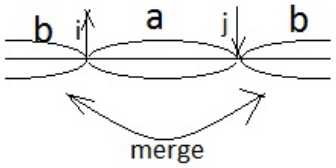
Вершина → любое выходящее ребро → `node*` → позиция в декартовом дереве → циклический сдвиг.

`Link(a, b)` — провести ребро

1. `MakeRoot(a)`
2. `MakeRoot(b)`
3. Выпишем компоненту вершины a
4. Проведем ребро ab
5. Выпишем компоненту вершины b
6. Проведем ребро ba

`Cut(a, b)` — удалить ребро





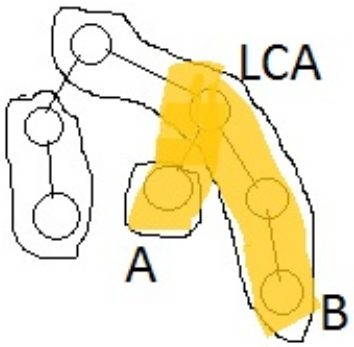
По ab находим позиции ребер (прямого и обратного) в массиве. Разделяем массив на три части и склеиваем первую и последнюю.

7.2. Heavy-Light Decomposition

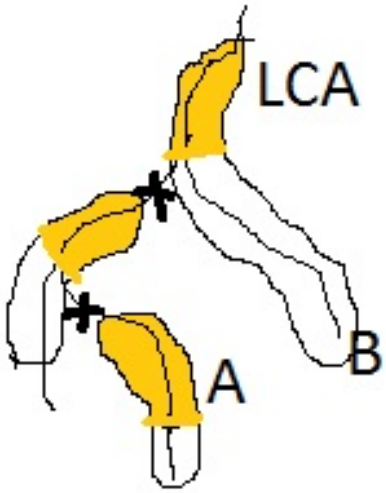
7.2.1. Определение

Хотим покрыть дерево вертикальными путями так, что бы каждая вершина лежала ровно в одном.

Def 7.2.1. Heavy-Light Decomposition — такое разбиение на пути, что на пути от каждой вершины до корня не более $\log n$ ребер, не покрытых путями.



LCA разбивает путь на два вертикальных.
Что бы посчитать функцию, нужно посчитать функцию на двух вертикальных путях.



Def 7.2.2. Назовем прыжком переход от одного пути в разбиении к другому.

7.2.2. Конструкция

Def 7.2.3. Ребро ba назовем тяжелым(b — предок a), если

$$size[a] > \frac{1}{2} size[b]$$

Def 7.2.4. Не тяжелые ребра называются легкими.

У каждой вершины не более одного тяжелого ребра. Пути состоят только из тяжелых ребер. И все тяжелые ребра лежат в путях.

Прыжков не более log, когда мы делаем прыжок, мы переходим по легкому ребру, значит, размер поддерева удваивается хотя бы в два раза.

7.2.3. Реализация

```
// no[v] - номер пути, котором лежит вершина
// pos[v] - номер вершины в этом пути
// p[v] - отец вершины v
build(int v) {
    if (p[v] != -1 && 2 * size[v] > size[p[v]]) {
        no[v] = no[p[v], pos[v] = pos[p[v]] + 1;
    } else {
        top[cnt] = v;
        no[v] = cnt++;
        pos[v] = 0;
    }
    for (int x: childs[v])
        build(x);
}

get(a, LCA) {
    while(no[a] != no[LCA]) {
        Tree[no[a]].get(0, pos[a]);
        a = p[top[no[a]]];
    }
```

```
21      Tree[no[a]].get(pos[LCA], pos[a]);
22  }
```

Время работы: построение $O(n)$, запрос $O(\log^2 n)$, обновление в точке $O(\log n)$.

Замечание 7.2.1. Когда мы решаем задачу, нужно сначала решить задачу на массиве, а потом обобщить.

Замечание 7.2.2. Если нет запросов на обновление, то heavy-light, чаще всего, не нужен.

7.3. Level Ancestor

𝔇𝔢𝔣 7.3.1. $LA(v,k) = parent^{(k)}(v)$. Необходимо найти k-ого предка вершины v.

Перечислим методы решения данной задачи при помощи уже известных нам алгоритмов и структур:

Offline: $O(n)$ Обходим дерево, когда мы находимся на вершине v, знаем для нее всех предков и ответы на все запросы.

Online:
1. $\langle n \log n, \log n \rangle$ — двоичные подъемы.
2. $\langle n^2, 1 \rangle$ — полный предподсчет.
3. $\langle n, \log n \rangle$ — heavy-light (нужно либо прыгнуть в следующий путь за $O(1)$, либо один раз взять соответствующий элемент в массиве на пути).

7.3.1. Решение за $\langle n \log n, 1 \rangle$

Различные способы разбить на пути:

Heavy-Light Продолжаем путь в самое тяжелое поддерво

Замечание 7.3.1. Продолжать путь в самое тяжелое поддерево не хуже, чем брать тяжелые ребра.

Longest-Path Будем продолжать путь в более глубокое поддерево.

Ladder Из любой декомпозиции можно сделать лестницу. Есть путь длины L, продолжим этот путь на L наверх. Теперь пути перекрываются.

$$LA = \text{Двоичные подъемы} + \text{Ladder-Longest-Path}$$

Прыгнем на самый большой возможный прыжок s. $2^s \leqslant k < 2^{s+1}$.

Теперь достаточно сделать один прыжок в Ladder-Longest-Path.

▶ Вниз можем спуститься хотя бы на 2^s, значит, наверх он продлен хотя бы на эту величину. Значит мы с предком на одном пути. ◀

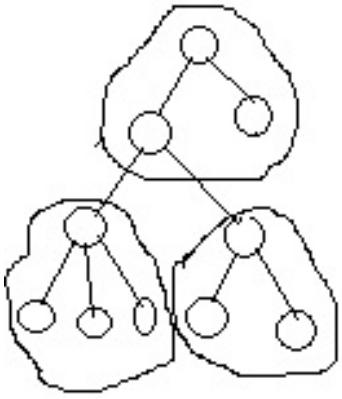
7.3.2. Решение за $\langle n, 1 \rangle$

Задача соптимизировать двоичный подъем.

$\langle M \log n, 1 \rangle$ M — количество листьев. Можем помнить любой лист в поддереве и расстояние до него. Теперь чтобы прыгнуть вверх от v на k, достаточно прыгнуть от leaf[v] на k + d[v].

$\langle M \log n \rangle$ прыжков в Offline. $\langle leaf, 2^k \rangle$. От каждого листа для каждого k посчитать прыжок на 2^k.

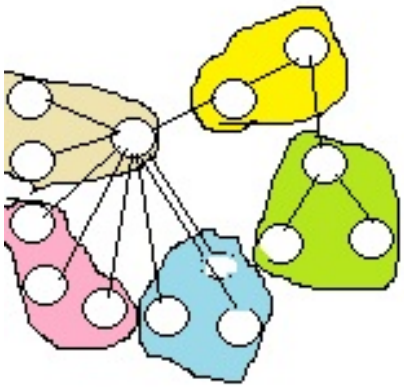
$\langle M \log n + n, 1 \rangle$ Наша цель сделать $M = O(\frac{n}{\log n})$ Сжатие деревьев.



Микро-макро эвристика

Хотим разбить вершины на поддеревья размера не больше k. Просто так это сделать сложно, поэтому разрешим корню поддерева не принадлежать данному куску.

$\frac{k}{2} < size \leqslant k$. Верно для всех кусков кроме корня. Тогда количество кусков $\frac{n}{k/2} + 1$.

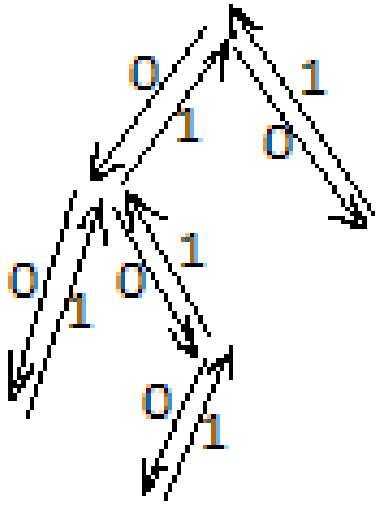


```
Compress(v) {
    for (v -> x) Compress(x)
}
```

$$k \leqslant \frac{\log n}{16}$$

Вершина лежит в куске. Знаем тип куска и позицию в куске. Различных типов мало. Для вершины за $O(1)$ хотим прыгать на k вверх. Предподсчет.

Двоичные подъемы у нас по исходному дереву. Делаем мы их только от корней k-деревье. От листа помним корень k-дерева, позицию в нем, тип k-дерева.



$\leftrightarrow$ 0010011101

Глава 8

Паросочетания

8.1. Паросочетания. Независмые и контроллирующие множества. Алгоритм Куна

Максимальная клика, максимальное независимое множество и минимальное покрывающее множество — одна и ты же задача с точностью замены графа на обратный.

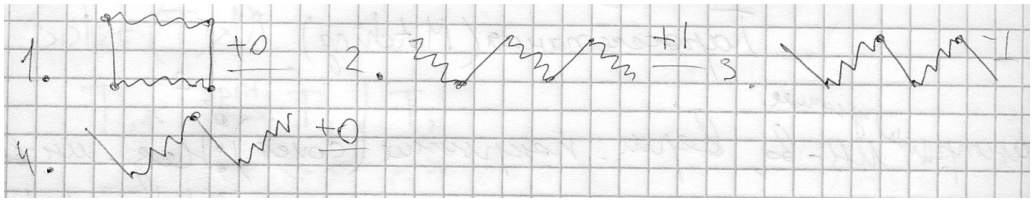
В произвольном графе проверить, что существует клика заданного размера — NP-полная задача.

Def 8.1.1. Паросочетание — набор попарно несмежных ребер.

Максимальное паросочетане — набор наибольшего размера.

Теорема 8.1.1. Если существует дополняющая чередующаяся цепь, то можно увеличить размер паросочетания. Обратное тоже верно.

▶ M — текущее паросочетание, P — какое-то паросочетание. Пусть $\exists |P| > |M|$. Рассмотрим $P \Delta M$. Граф разбился на 4 типа объектов.



т.к. $|P| > |M| \Rightarrow \exists$ объект типа 2, потому что только в нём рёбер из P больше, чем рёбер из M. Его-то мы и возьмём в качестве дополняющей цепи. ◀

Алгоритм:

1. найти цепь
2. применить её

Код для двудольного графа:

```
bool dfs(v) {
    u[v] = 1;
    for (int x: G[v]) {
        if (pair[x] == -1 || (u[pair[x]] == 0 && dfs(pair[x])) {
            pair[x] = v;
            return true;
```

```
        }
    }
    return false;
}
```

Тупой алгоритм

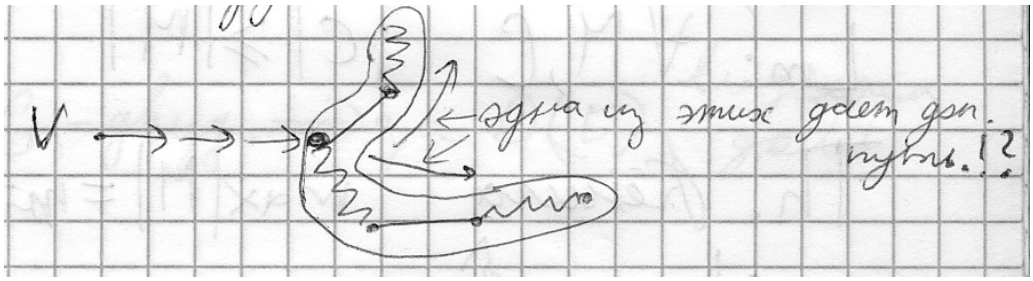
```
while(good) {
    good = false;
    u = {0};
    for v {
        if (u[v] == 0 && is[v] && dfs(v)) {
            good = true;
            break;
        }
    }
}
```

Алгоритм Куна $O(VE)$

```
for v {
    u = 0;
    dfs(v);
}
```

Теорема 8.1.2. Кун корректен

▶ Пусть не существовало дополняющего пути из вершины v. Мы применили некоторый путь и путь из вершины v появился. Посмотрим внимательно на картинку и поймём, что он уже был до этого.



◀

Оптимальный Кун $O(|M|E)$

```
u = {0};
for v {
    if dfs(v) {
        u = {0};
    }
}
```

Жадная инициализация

```
random_shuffle(Edges);
for (e: Edges) {
    if (!w[a[e]] && !w[b[e]]) {
```

```
4           w[a[e]] = w[b[e]] = true;
5           A += {e};
6       }
7   }
```

$|A| \geqslant \frac{Max}{2}$ потому что каждое ребро могло помешать не больше, чем двум рёбрам из ответа. Теперь оптимальный Кун работает хотя бы в 2 раза быстрее.

Пусть есть Max Matching, получим Cover и IS(независимое множество) за O(E)

Теорема 8.1.3. теорема Кёнига $max|M| = min|C|$

В двудольном графе максимальное паросочетание равно числу вершин в минимальном вершинном покрытии.

▶ $|C| \geqslant |M|$ (Пусть вершинное покрытие меньше, тогда какая-та вершина «покрывает» хотя бы два ребра. Противоречие).

Предъявим $|M| = |C|$.

Возьмем вершины, которые не лежат в паросочетание в левой доле.

Запустим из них Куна. Мы дошли из них до какого-то множества вершин.

A^+ — вершины из первой доли, до которых мы дошли

B^+ — вершины из второй доли, до которых дошел dfs

A^- — вершины из первой доли, до которых мы не дошли.

B^- — вершины из второй доли, до которых мы не дошли.

Cover $= A^- \cup B^+$

IS $= A^+ \cup B^-$

из A^+ нет рёбер в B^-

$|A^- \cup B^+| = |M|$ потому что A^- и B^+-концы рёбер из M ◀

8.2. Оптимизации Куна

Пусть есть Алгоритм Куна. За сколько он найдёт паросочетание размера K в двудольном графе на V вершинах и E рёбрах? $O(K(V+E))$.

Пусть все степени не более D. Тогда можно улчлшить Кун до $O(K^2D)$: каждый DFS обойдёт не более $K+1$ вершины первой доли, так как в паросочетании всего K вершин, а тогда из каждой такой вершины мы рассомтрели D рёбер. Итого $O(KD)$ на DFS (если мы умеем быстро обнулять метки) и $O(K)$ итераций.

Рассмотрим ещё такой код:

```
for (bool run = true; sun; ) {
    run = false;
    used = {0, 0, ...};
    for (int i(0); i != n; ++i)      // |
        if (hasNoPair[i] && dfs(i)) // | O(V + E)
            run = true;             // |
}
```

Этот алгоритм будет очень жадно улучшать паросочетания, в том числе на первом шаге оно будет работать как жадная эвристика. Ассимптотика такая же, просто гораздо быстрее.

Глава 9

Потоки

9.1. Потоки

Def 9.1.1. Пусть есть ориентированный взвешанный граф $G = \langle V, E \rangle$, на котором у каждого ребра есть неотрицательная характеристика c — *пропускная способность*, и две выделенных вершины: исток s и сток t.

Замечание 9.1.1. Можно рассматривать c как функцию $c \colon E \to [0, \infty)$.

Потоком называется такая функция $f \colon E \to \mathbb{R}$, такая, что:

1. Поток в каждой вершине, кроме истока и стока, сохраняется:

$$\forall v \notin \{s, t\}, \sum_{e \in E \colon v \to u} f(e) = \sum_{e \in E \colon u \to v} f(e)$$

2. Поток не превосходит пропусной способности.

$$\forall e \in E, f(e) \leqslant c(e)$$

Также мы для работы с потоками введём **обратные рёбра**. Они будут позволять нам понимать, сколько потока можно пустить обратно. У каждого ребра $e \colon v \to u$ будет парное $e' \colon u \to v$ с нулевой пропускной способностью, причём

$$\forall e \in E, f(e') = -f(e)$$

Также, обратным к обратному ребру будет исходное ($e'' = e$).

Тогда второе свойство можно перезаписать, используя обратные рёбра: сумма по всем исходящим рёбрам должна быть ноль:

$$\forall v \notin \{s, t\}, \sum_{e \in \tilde{E} \colon v \to u} f(e) = 0$$

поскольку все входящие рёбра исходного графа перейдут исходящие, обратные к ним, и с другим знаком.

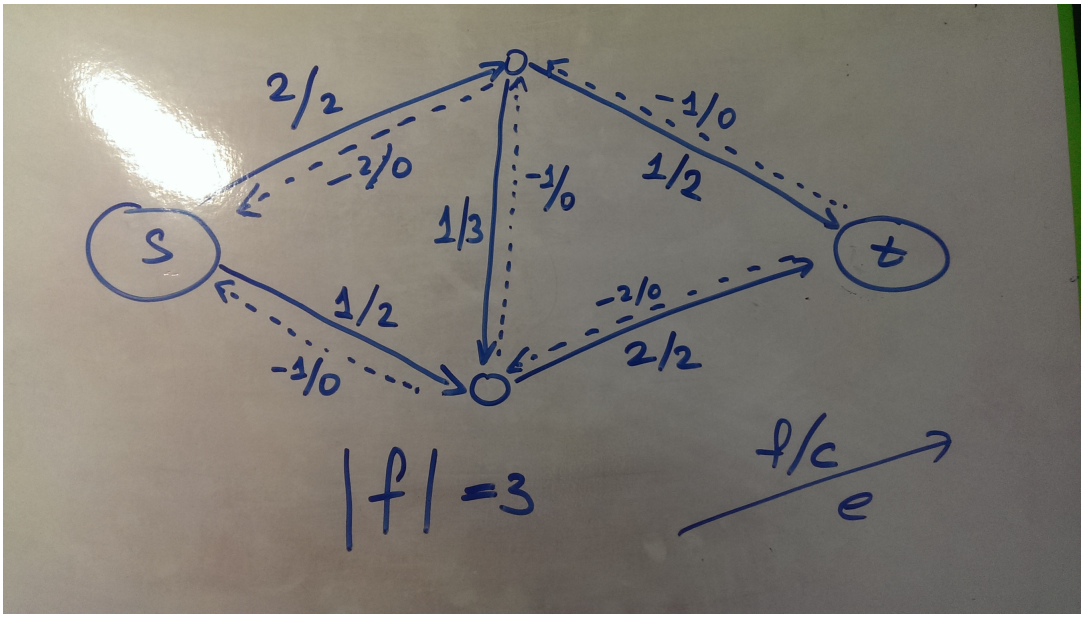
Замечание 9.1.2. Если мы хотим рассмотреть поток в неориентированном графе, достаточно у обратных рёбер сделать пропускную способность такую же, как у исходных рёбер.

Замечание 9.1.3. Начиная с этого места обратные рёбра рассматриваются наравне с обычными, если не оговорено иначе.

Def 9.1.2. Величиной потока называется сумма по всем исходящим из истока рёбрам:

$$|f| = \sum_{e \colon s \to u} f(e) = -\sum_{e \colon u \to t} f(e)$$

Задача максимизации потока: найти поток максимальной величины.



Def 9.1.3. Разрезом называется дизъюнктное разбиение $V = S \sqcup T$, что $s \in S$ и $t \in T$, то есть: мы разбили множество вершин на две группы, при этом исток и сток не лежат в одной группе.

Def 9.1.4. Величиной разреза называется сумма пропускных способностей рёбер между долями

$$c(S,T) = \sum_{\substack{e\colon v \to u \\ v \in S, u \in T}} c(e)$$

Def 9.1.5. Величиной потока через разрез называется суммарный поток между долями

$$f(S,T) = \sum_{\substack{e\colon v \to u \\ v \in S, u \in T}} f(e)$$

Тогда понятно, что

$$f(S,T) \leqslant c(S,T)$$

Лемма 9.1.1. Величина потока через любой разрез равна величине самого потока:

$$\forall f, \forall S, T \, |f| = f(S,T)$$

▶ Посчитаем сумму потока из всех вершин из S. С одной стороны, каждая вершина кроме s даст нулевой вклад, и эта сумма равна сумме рёбер из s, то есть величине потока.

Теперь по рёбрам: внутри S сумма потока ноль: каждое ребро имеет обратное, а остались только рёбра через разрез. Таким образом, величина потока через разрез равна величине потока. ◀

Лемма 9.1.2.

$$\max |f| \leqslant \min_{\langle S,T \rangle} c(S,T)$$

▶

$$\forall f, \forall \langle S,T \rangle \, |f| = f(S,T) \leqslant c(S,T)$$

◀

Def 9.1.6. Циркуляцией называется поток нулевой величины.

Например, поток по циклу или пустой поток.

Def 9.1.7. Остаточной сетью на графе с потоком называется новый граф на этих же вершинах, где из пропускных способностей вычли поток:

$$\langle G, c\rangle_f = \langle G, c - f\rangle$$

В том числе, например, обратные рёбра могут получить положительную пропускную способность, если имели нулевую.

Def 9.1.8. Дополняющий путь — путь из s в t, на котором все рёбра имеют меньший поток, чем пропускная способность.

Заметим, что такой путь позваляет увеличить поток: мы прибавляем что-то ко всем рёбрам пути и вычитаем из всех парных им. Можно руками проверить все три свойства нового потока.

Это определение можно переформулировать как путь из s в t в остаточной сети.

9.2. Теорема и алгоритм Форда— Фалкерсона

Следующий алгоритм находит максимальный поток:

```
while (ThereIsWay()) {    // Пока есть дополняющий путь в остаточной сети...
    for (auto e: way) {   // ...(ищем через dfs(s))...
        f[e] += 1;        // ...прибавить по всем его рёбрам...
        f[pair[e]] -= 1;  // ...единицу
    }
}
// получили максимальный поток
```

Время работы: $O(|f|E)$.

Докажем его:

▶ Поскольку поиск дополняющего пути не удался, то есть разрез: что посетил DFS (множество S) и не посетил (T). Заметим, что тогда все рёбра из S в T насыщены, то есть

$$|f| = f(S,T) = c(S,T)$$

А значит получили поток, равный разрезу. Значит поток максимальный, а разрез минимальный. ◀

Между прочим, научились по максимальному потоку находить миниамальный разрез.

Можно немного улучшить Форда— Фалкерсона: прибавлять максимум из того, что можно пустить по пути.

Теорема 9.2.1 (Теорема Форда— Фалкерсона)**.** Максимальность потока равносильна отсутствию дополняющего пути.

Доказательство само возникло по ходу доказательства алгоритма.

Следствие 9.2.1.1.

$$\max|f| = \min_{\langle S,T\rangle} C(S,T)$$

Замечание 9.2.1. Для целых пропукных способностей есть максимальный целочисленный поток. Посмотрим, как работает Форд— Фалкерсон: он на каждом этапе даёт целочисленный поток, значит и в конце тоже будет целочисленный.

Лемма 9.2.1. Существует максимальный поток.

▶ Заметим, что разрезов конечное число, минимальный точно существует.

Если только целые пропусные способности, то уже всё получилось: запустим наш алгоритм Форда— Фалкерсона. Для вещественного случая можно построить граф, что Форд— Фалкерсон никогда не остановится. Но поможет кое-кто другой... ◀

9.3. Алгоритм Эдмонса— Карпа

```
while (bfs()) {            // Пока есть кратчайший дополняющий путь...
    df = min{c - f}        // ...взять максимум, что можно прибавить на пути...
    for (auto e: way) {
        f[e] += df;        // ...и прибавить это к каждому ребру
        f[pair[e]] -= df;
    }
}
```

Ниже написано то, чего не лекции не было, но явно должно было быть

Время работы алгоритма $O(VE^2)$

▶ Одна итерация поиска пути и проталкивания потока работает за $O(E)$. Осталось доказать, что таких итераций будет $O(VE)$.

Докажем, что расстояние до каждой вершины не убывает после каждой итерации.

▶ Пусть, после итерации существует вершина, расстояние до которой от s уменьшилось. Рассмотрим ближайшую к s такую вершину. Пусть $\rho_2(v)$ — расстояние до вершины после проталкивания потока, $\rho_1(v)$ — до проталкивания.

Тогда рассмотрим минимальный путь к вершине v в расстояниях ρ_2, то есть после итерации. Также возьмём вершину перед v в этом пути. Пусть это вершина u. $\rho_2(u) \geqslant \rho_1(u)$, так как v ближайшая плохая вершина.

Если ребро (u, v) было до итерации, то

$$\rho_1(v) \leqslant \rho_1(u) + 1 \leqslant \rho_2(u) + 1 = \rho_2(v)$$

Противоречие.

Пусть ребро (u, v) появилось после итерации. Значит увеличение потока происходило на пути с ребром $v \to u$. Значит

$$\rho_1(v) = \rho_1(u) - 1 \leqslant \rho_2(u) - 1 = \rho_2(v) - 2$$

Противоречие. ◀

Если докажем, что каждое ребро может становиться насыщеным, не более чем V раз, то победа.

После каждой итерации хотя бы одно ребро становиться насыщенным, ребер E. Если ребро (v, u) насытилось один раз, то, чтобы оно насытилось во второй раз, перед этим нужно пройти по ребру (u, v).

$$\rho_1(v) = \rho_1(u) - 1$$
$$\rho_2(v) = \rho_2(u) + 1$$
$$\rho_2(u) > \rho_1(u)$$

Значит, что бы пройти по ребру еще один раз, нужно, чтобы расстояние до веришины u увеличилось. Максимальное расстояние может быть V, и расстояние только увеличивается, значит, насыщенным ребро будет становиться максимум V раз. Что и требовалось. ◀

Этот алгоритм доказывает существование потока для вещественного случая, так как никак не завязан на целочисленные пропускные способности. **Выше написано то, чего не лекции не было, но явно должно было быть**

9.4. Декомпозиция потока

9.4.1. Не пересекаются по рёбрам

Задача: найти k различных по рёбрам путей между двумя вершинами. Скажем, что в старт ограничен поток величины k, всё рёбра пропускают 1. Тогда найдём максимальный поток. Осталось выделить пути.

Лемма 9.4.1. В потоке ненулевой величины есть путь с положительным потоком.

▶ Пусть нет такого пути: налицо разрез нулевой величины. ◀

Декомпозиция потока раскалдывает его на пути и циркуляцию. Хотим конечное число путей. Найдём путь по рёбрам с неотрицательным потоком от истока до стока. Тогда заберём с этого пути минимум из потоков. Тогда хотя бы одно ребро «пересохнет». Это $O(E^2)$.

Осталось понять, как это к нашей задаче. Всё просто: не могут два пути из декомпозиции пройти по одному ребру, у него всего 1 пропускная способность. Всего $O(kE)$: мы всё за k шагов сделаем, так как хотя бы на 1 меняем поток.

Разгоним декомпозицию.

$$T = k \cdot dfs + E$$

Разгоним DFS до $O(V)$. Пусть он нашёл цикл. Выкинем весь этот цикл (вычитаем из рёбер максимум по потоку). Тогда DFS за $O(V)$ или найдёт цикл, или найдёт путь до t.

Оптимизация: если мы что-то вычли, нет смысла откатываться вообще назад, продолжим работу. Тогда при выходе из вершины уже всё, что могли, из неё уже сделали. Удалим ребро в неё.

9.4.2. Вершины не повторяются

Раздваиваем вершины (вершина переходит в вершина— ребро— вершина, у ребра пропускная способность 1).

9.4.3. Неориентированные графы

Тут два способа: или говорим, что у каждого ребра обратное парное имеет такую же (а не нулевую) пропускную способность, или просто тупо говорим, что ребро распадается на два ориентированных с ещё двумя парными.

Сейчас пользуемся первой идеей. Если искали рёберно непересекающиеся пути, то решение всё ещё работает. Только ответ надо правильно выписывать. Пусть мы прошли по ребру и туда, и обратно. Но тогда поток по этим двум рёбрам на самом деле нулевой, и при декомпозиции мы увидем немного другие пути, без этих рёбер. Всё хорошо.

9.4.4. По вершинам не пересекаются, неориентированный граф

Всё работает же: получили ориентировнный граф, раздвоили вершины.

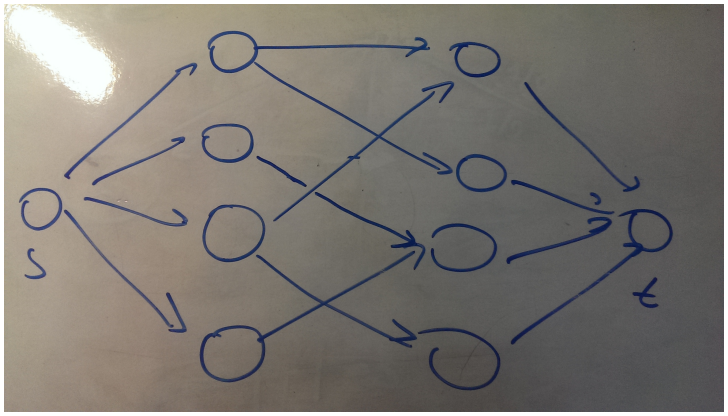
9.4.5. Немножко ещё!

Есть три вершины, нужно из A и из B найти разные пути в C. Тут просто пустить поток величины 2.

А ещё есть четыре вершины, и мы хотим найти из A в B и из C в D. И это решается... Никак. Это NP.

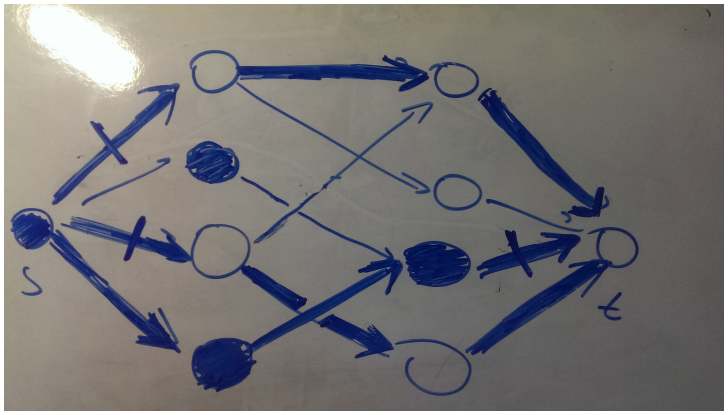
9.4.6. Паросочетание в двудольном графе

Добалвяем исток и сток, рёбра из первой доли во вторую. Чтобы сеть задавала паросочетания, все рёбра из истоки в сток пропускают 1, в середине — бесконечно много.



Это аж $O(VE)$.

Вспомним, что такое контролирующее множество: построим разрез по максимальному потоку. Тогда все рёбра через разрез не в середине: они идут или в первую долю, или из второй (в середине пропукная способность бесконечна). Рассмотрим множество вершин в двух долях, которые являются концами этих рёбер.



Тогда ровно эти вершины будут контроллирующим множеством, причём их будет ровно величина разреза, то есть ровно поток, то есть ровно паросочетание. Так мы нашли минимальное контроллирующее множество. Почему они контроллирующее? Пусть есть ребро паросочетания, не контроллируемое выбранным множеством. Тогда оно идёт из вершины первой доли, принадлежащей S (так как ребро из s в неё не через разрез), в вершину второй доли, принадлежащей T (так как ребро из неё в t не через разрез). Но тогда есть путь из S в T по ненасыщенному ребру (середина не может быть насыщена). Противоречие.

Также увидеть контроллирующее множество можно из алгоритма Куна, так как S — ровно множество помеченным Куном вершин (начали из первой доли вершин не в паросочетании, направо шли по прямым рёбрам, обратно — по паросочетанию, по обратным к насыщенным рёбрам).

Собственно, тут Форд— Фалкерсон работает точь-в-точь, как алгоритм Куна.

9.5. Масштабирование (Scaling)

Улучшим любой поиск потока. Сначала ищем пути по 2^k. Если не нашли — уменьшили k.

Время: цикл по k за $O(\log U)$ итераций. Внутри каждая за $O(E)$ DFS-ов: каждый путь пропускает не более, чем 2^{k+1} (иначе прошлая итерация ещё не закончилась). Тогда есть разрез величины не более $2^{k+1}E$. Тогда мы за $2E$ путей его высушим.

Получили $O(\log UE^2)$.
На вещественных вполне себе тоже норм: только спускаемся до погрешности. Время

$$O\left(\log\left(\frac{U}{\varepsilon}+1\right)E^2\right)$$

А если толкать сколько найдём, то в реальности оно за $O(E^2)$ работает, что очень быстро.

9.6. Заметки о реализации

Форда— Фалкерсона пишем, как Куна: прямо по дороге пропускаем поток:

```
bool dfs(int v) { // возвращает, нашёл ли путь
    if (was[v]) return false;
    was[v] = true;

    if (v == t) return true; // нашли путь
    for (auto& e: edges[v]) { // по рёбрам из v, в том числе обратным
        if (e.f == e.c) continue; // ребро насыщенно, не идём
        if (dfs(e.u)) { // если по ребру пошли и нашли путь до t
            e.f += 1; // пустили поток по ребру
            e.back->f -= 1; // вычли поток из обратного
            return true;
        }
    }
    return false; // ничего не нашли
}
```

9.7. Паросочетания

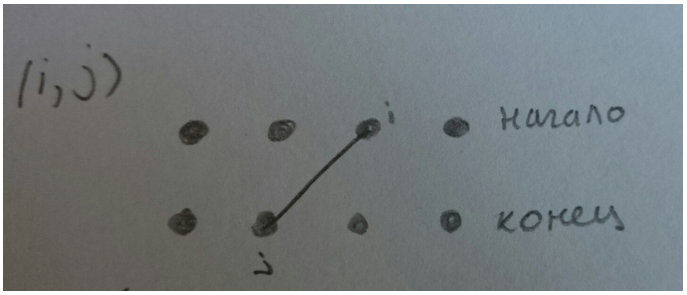
Что уже есть на двудольном графе:
$O(EV^{\frac{1}{2}})$ - Хопкофт-Карп
$O(VE)$ - Кун
$O(E\log V)$ - Регулярный двудольный граф
Произвольный граф:
$O(v^3), O^*(VE)$ - сжатие соцветий
$O(EV^{\frac{1}{2}})$ - Вазирани
$O(V^3)$ - Матрица Татта

Теорема 9.7.1. Теорема Дилворта
max Антицепь = min покрытию

Лемма 9.7.1. Есть ацикличный орентированный граф
(a) - либо транзитивно замкнутый
(b) - либо пути непересекаются
Будем работать в (a)
(a) $\Rightarrow$ (b):

1 Любой покрытие путями больше антицепи, так как на любое звено антицепи нужен путь.

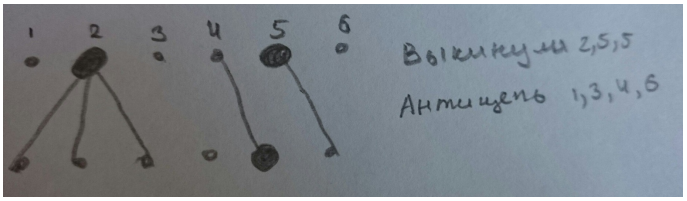
2 Разложим каждую вершину на две: начало и конец всех ребер. Таким образом из max matching можем востановить min path cover



Таким образом path = n - |Matching|. Значит, если $matching -> max$. то $path -> min$

Теперь Найдем антицепь такого же размера как Mathing

Антицепь = $V \setminus CoverSet$



|Cover Set| = |Matching|

Антицепь $\geqslant$ n - |Cover Set| ($\geqslant$ - так как вершины раздвоены, и могли выкинуть вершинуначального графа 2 раза)

Итак для любого path cover $\geqslant$ антицепи, и мы нашли антицепь $\geqslant$ path Cover. Значит они равны.

9.8. Раскраски

1 Двудольный граф: красим ребра так, чтобы у вершины все ребра были разного цвета.

Тогда:

Colors $\geqslant$ max Deg Так

1' Если граф регулярный (двудольный), то по лемме Холла существеут совершенное паросочетание.

Выберем одно паросочетание, в нем ребра не имеют общих вершин, покрасим их в один цвет. Таким образом k-регулярный граф $\to$ k-цветов. За $O(kMathing)$

Решение основных задач.

1 Добавим лишнии ребра так, чтобы сделать D-регулярный граф, где D = max Deg.

2 Добавим вершины так, чтобы количество вершин в левой и правой долях совпало.

$|V_1| = |V_2|$

3 Не D-регулярный значит существует u,v: $u \in V_1, v \in V_2, \deg u < D, \deg v < D$ - тогда соеденим их, при этом кратные ребра ничего не портят.

Быстрая покраска: $O(KMatching) \to O(\log kMatching)$

Замечание 9.8.1. T $\geqslant$ Matching, так что $\log kMatching$ - это быстро

1 Если k делиться на 2:

Выделим Эйлеров цикл, и разделим задачу на две:

четные и нечетные ребра цикла.

2 k - не делиться на 2 Выделим Matching. За $O(Mathing + E)$

$k \to k - 1$.

Время работы как у быстрого возведения в степень - $T(k) = T(k/2) + O(Mathing + E)$

9.9. Количество совершенных паросочетаний

Введем понятие Перманент: A=$\sum_\sigma \prod a_{i,\sigma_i}$, где A - матрица смежности.
Тогда совершенное паросочетание - перестановка если $\prod a_{i,\sigma_i} = 1$.
Итак совершенное паросочетание = perm A
(perm A) mod 2 = (det A) mod 2, так как -1 = 1 по модулю 2.

Замечание 9.9.1. det можно посчитать за $O(n^3)$

9.10. Возвращаемся к покраскам

Хотим покрасить все вершины так, чтобы две одинакового цвета не были соединины ребром.
D=max deg. В D+1 цвет умеем красить.
Эвристика: удаляем degmin, красим, возвращаем и красим min цвет.

Теорема 9.10.1. Теорема Брукса Если граф не является циклом нечетной длины и не полный - то его можно покрасить в D цветов, где D = max deg. (Красим вершины)

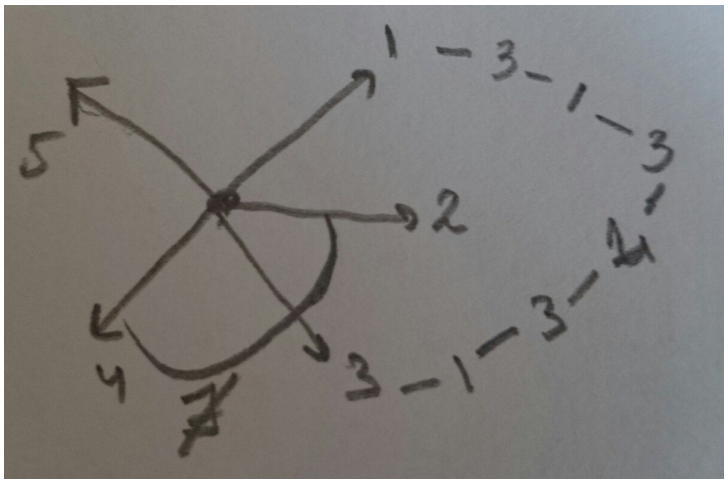
Теорема 9.10.2. Теорема Визина Любой граф можно покрасить в D или D+1 цвет, но это NP-полная задача. (Красим ребра)

Теорема 9.10.3. о 4 красках Любой планарный граф можно раскрасить в 4 цвета.

9.11. Покраска планарного графа

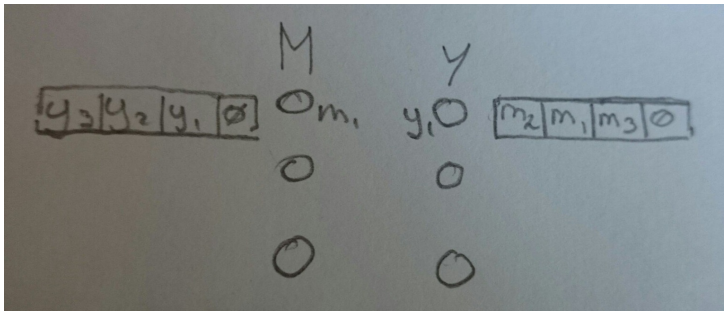
Пусть уже есть укладка графа. Значит существет вершина, со степенью $\leqslant 5$.
Красим вершины.
Можем покрасить в 6 цветов: выбираем вершину, со степенью $\leqslant 5$, "удаляем"вершину, красим все что осталось, возвращаем вершину и красим.

Теорема 9.11.1. Можно покрасить в 5 цветов



▶ Пусть у вершины все пять соседей разного цвета. Рассмотрим вершины 1 и 3: мы не можем поменять цвет вершины 1 на 3 только в том случае, если вершины 1 и 3 соеденины цепочкой $1-3-1-\cdots-1-3$. Пусть соедины, тогда рассмотрим вершины 2-4, они не могут быть соедины, так как граф планарный! Значит можем поменять цвет 2 на 4, и покрасить среднюю вершину в 2.
Существует алгоритм покраски в 4 цвета за $O(v^2)$ ◀

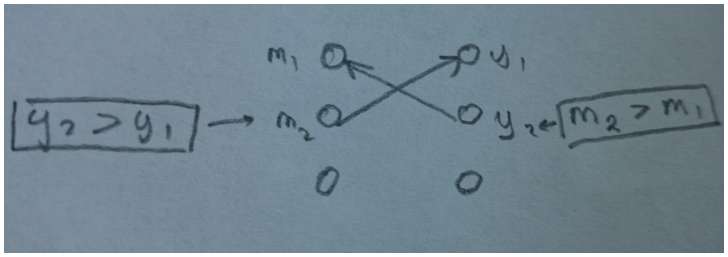
9.12. Stble Marriage



$\forall M \to list[y]$ - упорядоченный список Y

$\forall Y \to list[m]$ - упорядоченный список M

При этом, остаться без пары - хуже всего.



Def 9.12.1. Нестабильностью называется ситуация если приоритет $m_2 > m_1$ в списке y_1, и при этом приоритет $y_2 > y_1$ в списке m_1

Замечание 9.12.1. Stable$\not\Rightarrow$ Max

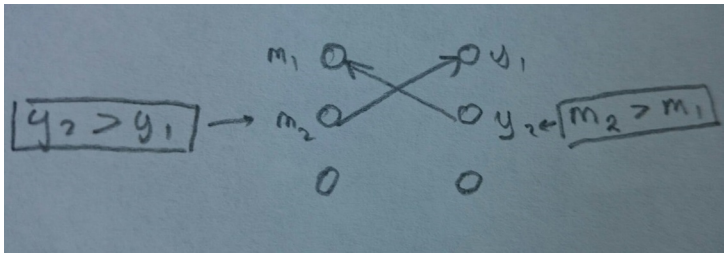
1. Находим не стабильность - исправляем. Может длиться бесконечно.
2. (Мальчики прдлагают - девочки отказываются)

   $\forall$ мальчик предлагает первой девочке в списке. $\forall$ девочка получив предложение говорит "может быть"(если пока лучший из тех, кто предложил) или "нет"(если уже предложил кто-то лучше).

   Пока у всех мальчиков список не будет из 1 элемента.

Замечание 9.12.2. Алгоритм корректен $\Rightarrow$ существует Stable Marriage.

Теорема 9.12.1. Алгоритм корректен



Как могло получиться, что y_2 с m_1, хотя хочет больше быть с m_2. Значит m_2 ей не предлогал, или раньше предложил y_1, но он больше хочеь быть с y_2. Противоречие.

9.13. Алгортим Диница

Алгоритм Диница позволяет найти максимальный поток за $O(VE + VK)$, где K - количество найденных путей из истока в сток. Если добавить scaling, будет $O(VE \log U)$.

Сеть кратчайших путей

Все вершины графа можно разбить части(слои) так, что длина минимального пути от стока до всех вершин одного слоя с одинакова.

Теперь из ребер исходного графа выберем те, которые соединяют вершины из "соседних"слоев. Эта констркуция и называется сетью кратчайших путей, а любой путь из истока в сток по ребрам сети кратчайших путей является кратчайшим.

9.13.1. Описание алгоритма

- Получаем сеть кратчайших путей, найдя расстояние от истока до каждой из вершин.
- Жадно находим дополняющие пути и пускаем по ним потоки. Если ни одн путь не нашелся, алгоритм завершается.
- Строим остаточную сеть и возвращаемся к первому пункту.

Псевдокод

```
while (bfs(s, t)) {
  while dfs(s)
    UpdateFlow();

dfs(v) {
  if(v == t) return 1;
  while exists (v, u) в сети кратчайших путей {
      if (dfs(u))
          return 1;
      else
          удаляем ребро из v в u
  return 0;
}
```

Лемма 9.13.1. После каждого шага алгоритма минимальное расстояние от истока до вершины увеличивается. (см. Эдмондс Карп)

Докажем, что итоговая сложность $O(V(VK+E))$.
$\Sigma(Vk_i + E) \leq VE$ — для одной фазы. E берется из dfs и от того, что каждое ребро мы рассмотрели не более одного раза: либо мы нашли путь, в котором оно участвует, либо удалили соответствующую вершину. Каждую вершину мы рассмотрели не больше раз, чем количество найденных путей. V — фаз всего.

Теорема 9.13.1. Первая теорема Карзанова
Если $P = \sum \min(c_{in}, c_{out})$, то Диниц работает за $O(V\sqrt{P})$

Теорема 9.13.2. Вторая теорема Карзанова
Если нет кратных ребер, количество фаз – $V^{\frac{2}{3}}U^{\frac{1}{3}}$.

▶ $U \cdot min|V_i||V_{i+1}| \leq (\frac{V}{L}^2$
$Flow \leq Cut \leq min|V_i||V_{i+1}| \Rightarrow$
$V^{\frac{2}{3}}U^{1}3 \leq L \Rightarrow$ количество фаз $\leq V^{\frac{2}{3}}U^{\frac{1}{3}}$ ч.т.д. ◀

9.13.2. LR-поток

я не уверена в том, что написано в этом разделе. Пусть кто-нибудь проверит

1. решим задачу, когда есть много истоков и много стоков. И хотим пустить максимальный поток из истоков в стоки.

   ▶ Создадим вспомогательный исток S и вспомогательный сток T.
   Соединим S со всеми основными стоками ребрами пропускной способности ∞ и все стоки со стоком T пропускная способность опять ∞.
   Пустим поток из S в T. ◀

2. Немного усложним задачу. Теперь у истоков есть избытки s_i и у всех стоков недостатки t_j. Известно, что $\sum s_i = \sum t_j$. Нужно "переправить"все избытьки в недостатки .

   ▶ Такая же конструкция, как и в прошлый раз, но пропускная способность ребер не ∞, а s_i и t_j соответственно. ◀

3. Теперь хотим найти LR циркуляцию. У каждого ребра пропускная способнасть ограничена с двух сторо $L \leqslant f_e \leqslant R$ Хотим пустить какую-то циркуляцию, которая удовлетворяет этим условиям.

   ▶ По всем ребрам пустим поток L_e. В результате чего в каких-то вершинах окажутся избытки, в каких-то недостатки, а пропускная способность ребер будет $R - L$.
   Теперь нужно переноправить избытки в недостатки. Свели задачу к предыдущей. ◀

4. LR — поток.

   ▶ Проводим ребро TS с пропускной способнотью ∞ и ищем циркуляцию. ◀

5. $maxLR$ — поток.

   ▶ $f_{max} - f$ — поток в G_f. То есть найдем какой-то $L - R$ поток. То есть теперь у вершине не будет избытков. Они все уже будут перенаправлены и в таком новом графе найдем максимальный поток. ◀

9.14. Mincost Flow, Circulation, Hungrian Method

9.14.1. Mincost Flow

Лемма 9.14.1.

$$\forall |f_1| = |f_2| \colon f = f_2 - f_1 - \text{циркуляция в } G_{f_1}$$

▶ Проверяем 3 утверждения:

1. $f \leqslant c$ (В G_{f_1} пропускная способность $c(e) - f_1(e) \Rightarrow f_2(e) - f_1(e) \leqslant c(e) - f_1(e)$)
2. $\overline{f} = -f$
3. $\sum$ равна нулю

◀

Отсюда получаем следствие: $\forall$ Max поток $f = f^* + circ$, где f^* – это какой-то max-поток.

9.14.2. Mincost k-flow, Mincost max-flow, Mincost Flow, Mincost Circulation и их сведение друг к другу

1. Mincost k-flow – поток минимальной стоимости, где $|f| = k$.
2. Mincost max-flow – из всех потоков, где $|f| = max$ выбираем поток с минимальной стоимостью.
3. Mincost Flow – мы стремимся минимизировать стоимость потока $cost(f) \to min$.
4. Mincost Circulation – $|f| = 0$ – частный случай k-flow.

Уточнение: пусть по ребру течет поток f, а его стоимость равна c. Добавим в граф обратное ребро с потоком $-f$ и стоимостью $-c$, чтобы иметь возможность "отменять"поток когда мы проходим по такому обратному ребру.

Тогда $cost(f) = \frac{1}{2} \sum_{\text{по всем ребрам}} f_e \cdot c_e$ ($\frac{1}{2}$ – если считаем с учетом обратных ребер: $fc + (-f)(-c)$)

Лемма 9.14.2. Сведение.Если мы хотим из потока определенного размера получить поток такого же размера, но минимальной стоимости, нам нужно прибавить к нему циркуляцию минимального веса.

▶ Напрямую следует из первой леммы. ◀

Комментарии к следующей лемме:

Напоминание про потоки:
Величина потока: $|f| = \sum_{i \in V} f_{si}$
Стоимость потока: $cost = \sum_{i,j \in V} C_{ij} f_{ij}$, где C_{ij} – стоимость единицы потока вдоль данного ребра

В следующей лемме фигурирует понятие остаточной сети. Концепция остаточной сети G_f следующая: пусть по нашему ребру e течет поток f_e, а его пропускная способность u_e. Тогда теоретически мы можем по этому ребру пустить еще $u_e - f_e$ потока. Назовем эту величину пропускной способностью остаточной сети. Стоимость же таких дополнительных единиц потока оставим такой же.

Также применим в остаточной сети идею разделения одного ребра на 2 ориентированных, чтобы иметь возможность "отменять"поток. Ее мы обсудили раньше, заметим только, что величина пропускной способности остаточной сети у прямого и обратного ребра будет одинаковой. Теперь мы готовы сформулировать лемму.

Лемма 9.14.3.

$$|f| = k \colon cost(f) = min \Leftrightarrow \nexists \text{ отрицательных циклов в остаточной сети } G_f.$$

▶ 1. Необходимость.
Предположим, что остаточная сеть содержит цикл отрицательного веса. Тогда посмотрим на пропускные спобобности остаточной сети для всех ребер этого цикла и выберем среди них минимум. Заметим, что мы можем увеличить на эту величину поток вдоль всех ребров нашего цикла и уменьшить тем самым стоимость потока (она уменьшится на стоимость цикла, умноженную на величину этого самого минимума). А это противоречит оптимальности f.

2. Достаточность.
Мы уже умеем делать декомпозицию потока, т.е. раскладывать его на пути и циркуляцию. Также мы можем показать, что для двух потоков f и g одинакового размера верно, что g можно представить как $f+$ несколько циклов в остаточной сети G_f. Рассмотрим разность $g - f$ – это почленное вычитание (вычитание потоков вдоль каждого ребра). Исходя из условия, у нас получится некоторый поток нулевой величины (это циркуляция по определению). Произведем декомпозицию этой циркуляции на пути и циклы. Но! Заметим, что если бы в разложении присутствовали пути из истока в сток с положительным потоком, это означало бы, что величина потока в сети тоже была бы положительна. А мы имеем дело с циркуляцией. Значит в результате декомпозиции она разложится на несколько циклов. Значит разность $g - f$ можно представить как сумму циклов в сети G. Но в остаточной сети это также будут циклы, т.к. $g_e - f_e \leqslant u_e - f_e = r_e^f$, а это и есть пропускная способность остаточной сети.

Теперь мы готовы доказать достаточность: рассмотрим поток f, в остаточной сети которого нет циклов отрицательной стоимости. Пусть f^* – поток того же размера, но минимальной стоимости. Т.к. размеры потоков совпадают, то f^* можно представить как $f+$ несколько циклов. Т.к. стоимости всех циклов неотрицательны, то стоимость f^* явно не меньше, чем f. Но в силу оптимальности выбора f^* его стоимость не больше стоимости f, а значит они совпадают. А это значит, что наш поток f минимален по стоимости.

◀

Алгоритм

```
while (exists cost(cycle) < 0) // Пока есть отрицательный цикл, добавляем его к ответу
    f += cycle * min(u_e - f_e) // cycle -- длина цикла, min по пропускной способности
```

Время работы

Если мы обозначим через C наибольшее из стоимостей рёбер, через U — наибольшую из пропускных способностей, то максимальное значение стоимости потока не превосходит mCU (просто по определению стоимости). Если все стоимости и пропускные способности — целые числа, то каждая итерация алгоритма уменьшает стоимость потока как минимум на единицу, следовательно, алгоритм сойдется за линейное время и совершит (mCU) итераций. Тогда итоговая асимптотика составит:
$O(mUC)\cdot$ поиск цикла
Поиск цикла отрицательной стоимости в графе с n вершинами и m рёбрами может производится за $(m\sqrt{n}\log C)$ алгоритмом Гольдберга.

Алгоритм 2. Mincost Circulation → Mincost k-flow Поиск mincost k-flow в графе без отрицательных циклов или уже после нахождения минимальной циркуляции.

```
for k
    f += ShortestPath in G_f
```

Доказательство корректности по индукции:

▶ **База:** f_0 – mincost
Индукционный переход: f_i – mincost $\Rightarrow$ f_{i+1} – mincost

f^*_{i+1} — поток минимальной стоймости размера $i+1$.
Рассмотрим $f^*_{i+1} - f_i$ в остаточной сети G_{f_i}. Разложим $f^*_{i+1} - f_i$ на путь и циркуляцию ($\geqslant 0$ по предположению о том, что поток f_i минимальной стоимости, иначе бы пустили бы поток в доль циркуляции). f^*_{i+1} получаем из f_i добавлением минимального по стоимости пути $\Rightarrow$ получаем объект не худшей стоимости.

◀

Улучшаем алгоритм:
Shortest Path → FB
Shortest Path → Dijkstra

Но проблемой являются отрицательные веса. Чтобы от них избавиться, перед запуском Дейкстры, запустим Форда-Беллмана:

1. FB
   $p[v] = distance[v]$
   $w'_e = p[a] + w_e - p[b] \geqslant 0$
2. Dijkstra

Проблема: тут мы подразумеваем, что веса не меняются, а по факту могли "открыться"обратные отрицательные ребра.
Решение проблемы: $\forall v : p[v] += distance[v]$ – хотим сказать, что если теперь ребро в кратчайшем пути, то оно будет веса 0 $\Rightarrow$ и обратное будет веса 0.
Функция пересчета весов:

```
    Apply(w, p) {
        for w_e = ...    // Пересчитываем вес ребра с учетом потенциалов
    Apply(w, distance)   // Наш потенциал - расстояние
```

Лемма 9.14.4. Расстояния увеличиваются.

▶ Без доказательства, аналогичная уже доказана. ◀

```
    while |f| < k
        f += ShortestPath * min(u_e - f_e), k - |f|
```

Хотим найти поток размера k, ограничиваем этим размер
Сильно уменьшили k – улучшили асимптотику (но не знаем, во сколько)

Диниц:
Все пути, длина которых совпадает с кратчайшим - сеть кратчайших путей.
Алгоритм: Мы или идем по слоям (если можем), а если не можем – ищем максимальный поток.

```
    f = 0;
    while FB
        df = Maxflow в сети кратчайших путей
    f += df;
```

Хотим добавить поток (мы знаем, что поток представим в виде нескольких путей и циркуляции).
Добавили кратчайший путь, по лемме нового кратчайшего возникнуть не могло, значит след. будет кратчайшим ($w2$).
NB!
FB с очередью в mincost потоках работает не за $O(VE)$, а за $E + Maxflow$, поэтому мы не паримся с улучшением поиска кратчайших путей.

9.14.3. Назначения, Транспортная задача

Назначения
Имеется некоторое число работ и некоторое число исполнителей. Любой исполнитель может быть назначен на выполнение любой (но только одной) работы, но с неодинаковыми затратами. Нужно распределить работы так, чтобы выполнить работы с минимальными затратами, то есть $\sum_i C_i p_i \to \min$.
Решение:
$O(E) + n \cdot Dijkstra$ // $O(E)$ – динамическое программирование, на практике это все получается за $O(n^3)$

Транспортная задача
Задача об оптимальном плане перевозок некоторого однотипного продукта из пунктов производства в пункты потребления по некоторой дорожной сети на однотипных транспортных средствах.
Решение задачи
Объединим всех производителей в исток, а потребителей в сток и рассмотрим получившийся двудольный граф. Соединим попарно производителей и потребителейя рёбрами бесконечной пропускной способности и цены за единицу потока. К производителям присоединим фиктивный исток. Пропускная способность рёбер из истока в каждого производителя равна количеству продукта в этом пункте. Цена за единицу потока у этих рёбер равна 0. Аналогично к потребителям присоединяем сток. Пропускная способность рёбер из каждого потребителя в сток равна потребности

в продукте у данного потребителя. Цена за единицу потока у этих рёбер тоже равна 0. Дальше решается задача нахождения максимального потока минимальной стоимости (mincost maxflow).

9.14.4. Mincost Flow to Mincost Circulation

```
f_0 = min_circ
while ShortestPath < 0
    f += ShortestPath*...
```

9.14.5. Быстрый Mincost Flow

Мы уже умеем сводить к Mincost Circulation.

1. Mean Cycle Canceling – вычеркиваем цикл минимального среднего веса.
   $\log(nC) \cdot mn$

   mn – находим цикл алгоритмом Карпа, $\log n$ – крутые структуры данных

2. Capacity Scaling
   $fork = 30..0$
   $u' \leftarrow \frac{u}{2^k}$
   $Algo(u')$

3. Удалим ребро, добавим ребро
   Возьмем любое ребро, удалим его, найдем Mincost Circulation, добавим обратно.
   Mincost Circulation $= m \cdot MincostFlow$ // Когда добавляем ребро – ищем min поток из s в t на графе, где точно нет отриц. циклов.

4. Cost Scaling
   $(-c; +\inf) \to (-c/2; +\inf)$
   Описание одной фазы: пустили поток, применили потенциалы:

```
    f += df
    Apply(w, p)
```

Лемма 9.14.5. Таких фаз не больше, чем $\log(nC)$

9.14.6. Hungrian Method

Постановка задачи Задача, которую решает данный алгоритм - задача о назначениях в ее классической постановке.
Переформулируем ее в матричном виде. Дана неотрицательная матрица размера $n \times n$, где элемент в i-й строке и j-ом столбце соответствует стоимости выполнения j-ого вида работ i-м работником. Нужно найти такое соответствие работ работникам, чтобы расходы на оплату труда были наименьшими. Если цель состоит в нахождении назначения с наибольшей стоимостью, то стоимости C заменяются соответственно на разность между максимальной стоимостью и C, а суть решения не меняется.

Решение задачи У нас есть матрица, клеточки которой показывают соответствие работы и исполнителя.

1. Уменьшаем элементы построчно. Находим наименьший из элементов первой строки и вычитаем его из всех элементов первой строки. При этом хотя бы один из элементов первой строки обнулится. То же самое выполняем и для всех остальных строк. Теперь в каждой строке матрицы есть хотя бы один ноль. Иногда нулей уже достаточно, чтобы найти назначение.

2. Если нулей много, мы можем воспользоваться алгоритмом Куна нахождения наибольшего паросочетания в двудольном графе (если $ij = 0$, значит в нашем графе есть ребро $i \to j$). А также мы проверяем, возможно ли решение в принципе. Так, если в каком-то из столбцов не будет нулей, то задача не имеет решения.

3. Если мы не нашли назначения, мы повторяем те же действия, но теперь со столбцами (ищем минимум и вычитаем) и проверяем на возможность.

4. Если после этого шага не возникло противоречий с разрешимостью задачи, но ответ тем не менее не найдем, то мы будем делать следующее. Сначала используем алгоритм Куна для поиска максимального паросочетания, чтобы отдать как можно больше подходящих работ людям, которые их выгоднее всего сделают. Это как раз те ячейки, значение в которых сейчас 0. Дальше мы помечаем все строки без назначений, столбцы, в которых на пересечении с такими строками стоят нули и строки, в которых на пересечении с такими столбцами стоят работы, которые уже назначены. И выделяем все помеченные столбцы и непомеченные строки. Это все мы сделали для того, чтобы провести наименьшее количество линий, которые покроют все нули.

5. Из непокрытых линиями элементов матрицы находим наименьший, вычитаем его из всех неотмеченных строк и прибавляем к пересечениям выделенных строк и столбцов.

6. Итак, теперь у нас все непокрытые жлементы не меньше нуля, плюс существует новый ноль (мы выбрали какой-то элемент из неотмеченных, при вычитании соответствующая ему клеточка занулилась).

7. Повторяем, пока не нашлось решение.

9.15. Preflow-push поток

Конспект лекции можно увидеть тут.

9.16. Глобальный реберный разрез

Научимся выбирать минимальный реберный разрез для всех позиций s и t. Алгоритм Штор-Вагнера:

Алгоритм состоит из $n - 1$ фазы.

На каждой фазе в множество A начинаем по одной добавлять вершину, такую что $\omega(w) = \sum_{u \in A} c(v, u) = max$.

Две последние добавленные вершины называем истоком s и стоком t, а в качестве размера минимального разреза между ними берем значение $w(t)$.

Обновляем ответ и объединяем вершины s и t.

Алгоритм Каргера-Штейна:

```
while (n > 2) {
    e = Random Edge;
    Join(e.begin, e.end);
}
```

Утверждение 9.16.1. Вероятность того, что ребра, оставшиеся после алгоритма, будут ребрами минимального разреза, равна $\Theta(\frac{1}{n^2})$.

▶ 1. Обозначим размер ответа за ans, минимальную степень вершины за k. Заметим, что $ans < k + 1$.
2. На i-м шаге у нас $n - i$ вершин, вероятность попасть в ребро не из разреза - $1 - \frac{k}{E}$.
3. Тогда, так как $E > \frac{(n-i)k}{2}$, вероятность ни разу не попасть в ребро из разреза:

$$p_n > \prod_{i=0}^{n-2}(1 - \frac{2}{n-i}) = \frac{2}{n(n-1)}$$

.

◀

Эту вероятность можно улучшить до $\Theta(\frac{1}{n^2})$ следующим образом:

Пусть в начале у нас было n вершин. Будем объединять вершины, пока их не станет $\frac{n}{2}$

Теперь выберем не одно, а два случайных ребра, и перейдем к шагу один для графа со сжатым первым ребром, аналогично для второго.

Глава 10

Строки

10.1. Строки. Базовые алгоритмы.

10.1.1. Общие понятия

Пусть дана строка s длины n. Тогда подстрокой строки s называется строка $s[i \dots j]$, где $0 \leqslant i \leqslant j < n$. Префиксом называются подстроки, где $i = 0$, а суффиксом подстроки, где $j = n - 1$.

Период строки это целое положительное число t, такое что $\forall i \in [0 \dots n - t) \colon s[i] = s[i + t]$. Целым периодом называется такой период, что он делит длину строки нацело. Существует так же минимальный период, это минимум из всех периодов.

10.1.2. Префикс-функция

Пусть дана строка s. Префикс-функция $p[i]$ для строки s, длина наибольшего префикса строки $s[0 \dots i)$(не совпадающий со всей строкой), который одновременно является её суффиксом. Более формально $p[i] = \max n < i \colon s[0 \dots n) = s[i - n \dots i)$.

Научимся префикс-функцию для всей строки s.

```
strings s;
int n = (int) s.length();
vector<int> p (n);
int k = 0;

for (int i = 1; i < n; ++i) {
  k = pi[i];
  while (k > 0 && s[i] != s[k])
    k = pi[k];
  if (s[i] == s[k]) ++k;
  pi[i + 1] = k;
}
```

Поймем как и почему это работает. Ждем продолжения...

Почему работает за $O(n)$. Заметим что увеличение префикс функции происходит всего лишь в одной строке `if (s[i] == s[k]) ++k;`, всего таких увеличений не больше n, а количество уменьшений строка `k = pi[k];` не может первосходить количество увеличений, следовательно общее время работы это $O(n)$.

Как ускорить префикс-функцию. Можно убрать строку $k = pi[i]$, так как k уже равно $p[i]$, если не верите посмотрите на последнюю строку цикла.

10.1.3. КМП

Поиск подстроки в строке. Алгоритм Кнута-Морриса-Пратта. Даны две строки s и t. s — образец, а t — текст. Нужно найти все вхождения образца в текст.

Будем решать с помощью префикс-функции. Создадим новую строку, являющейя конкатенацией трех строк `str = s + symb + t`, где `symb` — строка состоящая из одного символа, не входящего ни в s, ни в t (стоп-символ). На полученной строке посчитаем префикс-функию. В местах где префикс функция равна длине строки s, там заканчивается очередное вхождение образца в текст.

Для того чтобы обойтись без применения стоп-символа надо добавить в префикс функцию строку `if (k > len) k = p[k]`, где `len` — длина образца.

10.1.4. z-функция

z-функция от строки s — это массив z, такое что $z[i]$ равно наидлиннейшему префиксу подстроки, начинающейся с позиции i в строке s, который одновременно является и префиксом всей строки s.

$$z[i] = LCP[0, i]$$

где LCP — это наибольший общий префикс (longest common prefix).

Посчитаем LCP.

```
while (s[k] == t[k])
  ++k;
```

Давайте сравнивать `int` так ведь быстрее.

```
si = (int*) s;
ti = (int*) t;
while (si[k] == ti[k])
  ++k;

k *= 4;
while (s[k] == t[k])
  ++k;
```

Отличный результат. Но почему бы не сравнивать еще большие куски памяти, реализуем.

```
const int CMP_LEN = 256;
sn = (int*) s;
tn = (int*) t;

while (memcmp (s, t, CMP_LEN))
  sn += CMP_LEN,
  tn += CMP_LEN;

while (*sn == *tn)
  ++sn,
  ++tn;
```

Наконец-то добрались до самой z-функциии.

```
int n = (int) s.length();
vector<int> z (n);

for (int i = 1, l = 0, r = 0; i < n; ++i) {
  if (z[i - l] < r - i) {
    z[i] = z[i - l];
  } else {
    z[i] = max (r - i, 0);
    while (s[z[i]] == s[i + z[i]])
      ++z[i];
    l = i;
    r = i + z[i];
  }
}
```

10.1.5. Боуер-Мур

Алгоритм Боуера-Мура — алгоритм поиска образца в тексте.

Опишем несколько идей:

Для начала допустим мы сравнивали образец и какую-то подстроку в тексте и получилось так что рассматриваемого символа из текста вообще нет в образце, тогда можно сдвинуть со следующего символа можно заново начинать сравнивать строки.

Эвристика стоп символа. Сделаем предподсчет $next[i, char]$. Ждем продолжения....

10.1.6. Полиномиальный хеш

Пишем хеши как Серёжа: `http://codeforces.com/blog/entry/17507`

10.1.7. Анти-хеш тест

1. $\forall < p, M > \exists Hash(s_1) = Hash(s_2)$ — анти-хеш тест.
2. $< p = rand, M = rand >$ — неизвестно как построить анти-хеш тест.
3. $< p = rand, M >$ — строка Туе-Морса является анти-хеш тестом.

10.1.8. Рабин-Карп

Алгоритм Рабина-Карпа — алгоритм поиска образца в тексте с помощью хешей. Суть алгоритма — перебор всех подстрок, длина которых равна длине образца, и сравнение их хешей с хешом образца. Существует одна проблема — это колизия хешей. Чтобы избавиться от этого, если хеши совпали просто проверим строки на равенство за линейное время. Так как вероятность повторения таких событий довольна маленькая, то суммарное время работы будет линейным $O(n + m)$.

10.1.9. LCP(i, j)

1. $< O(n^2), O(1) >$. Если $s[i] == s[j]$, то $LCP(i, j) = LCP(i + 1, j + 1)$, иначе $LCP(i, j) = 0$.
2. $< O(n), O(\log n) >$. Предподсчет хешами. Бинпоиск по ответу при каждом запросе.

Суффиксный массив за $O(n \log^2 n)$. Просто отсортируем строки стандартным `sort`, изменив компаратор на $LCP(i, j)$ за $O(\log n)$, получаем нужную асимптотику.

Замечание 10.1.1. Наибольшая общая подстрока = бинпоиск + хэши + хэш-таблица. Рещение за $O(n \log n)$. Для k строк за $O(kn \log n)$

10.2. Палиндромы

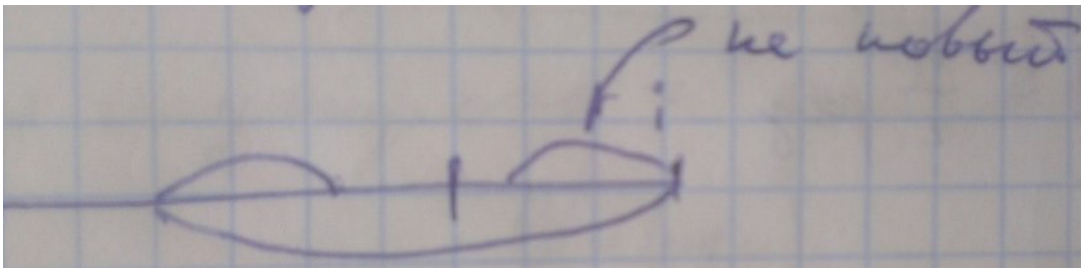
Задачи про палиндромы чаще всего решаются для палиндромов чётной и нечётной длинны независимо, и, чтобы не копировать код, можно писать:

```
for (int t = 0; t < 2; ++t) {
    ...
}
```

Задачу про наибольший общий подпалиндром решите сами =)
Максимальны чётный палиндром за $O(n)$

```
for (i, n) {
    while [i - (m + 1), i) == [i, i + (m + 1)) {
        m++; \\ не больше n / 2 раз
    }
}
```

Алгоритм Манакера для поиска количесва палиндромов в центре с заданной точкой за $O(n)$ (в данном примере только для чётных)
Пусть мы хотим посчитать ответ для позиции i (R[i]), а текущий самый правый палиндром - $a, a+1, ..., b$. Если i между a и b, то рассмотрим позицию $j = (a + b - i)$. Она очень похожа на i, потому что они самметричны относительно центра палиндрома и $R[i] >= \min(R[j], b - i)$. Если $R[j] >= b - i$ нужно попробовать увеличить ответ для $R[i]$ за границы палиндрома $a, a+1, ..., b$ и обновить палиндром $a, a+1, ..., b$, т.к. у нас мог получиться палиндром длиннее.

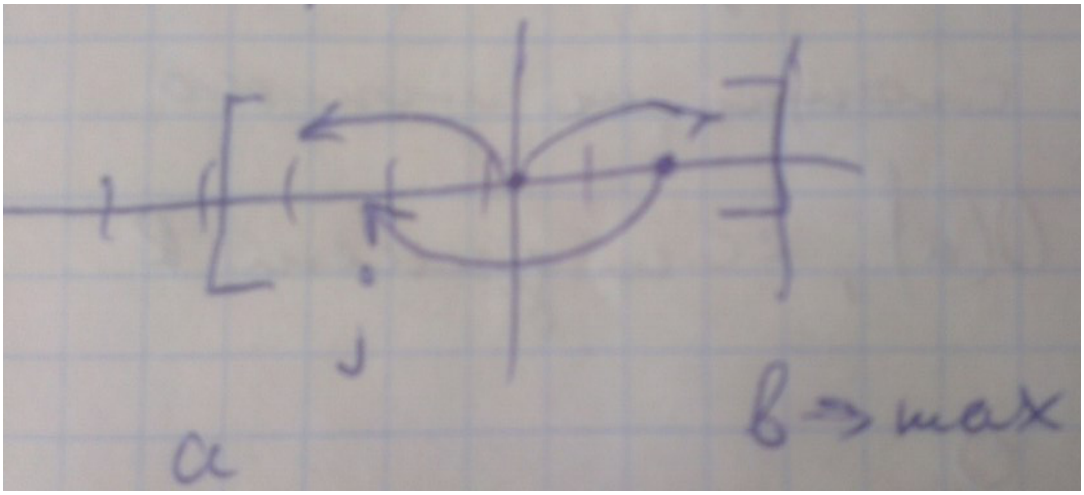



```
a = b = 0;
for (i, n) {
    R[i] = i > b ? 0 : min(b - i, R[j]);
    while можем увеличить палиндром {
        R[i]++;
    }
    if (i + R[i] > b) {
        a = i - R[i];
        b = i + r[i];
    }
}
```

работает за $O(n)$ потому что каждый R[i]++ по сути увеличивает b на 1, а $b <= n$

Теорема 10.2.1. Различных палиндромов не больше n

▶ В позиции i заканчивается только один новый палиндром. Пусть не так, тогда меньший из двух уже встречался в начале большего. ◀

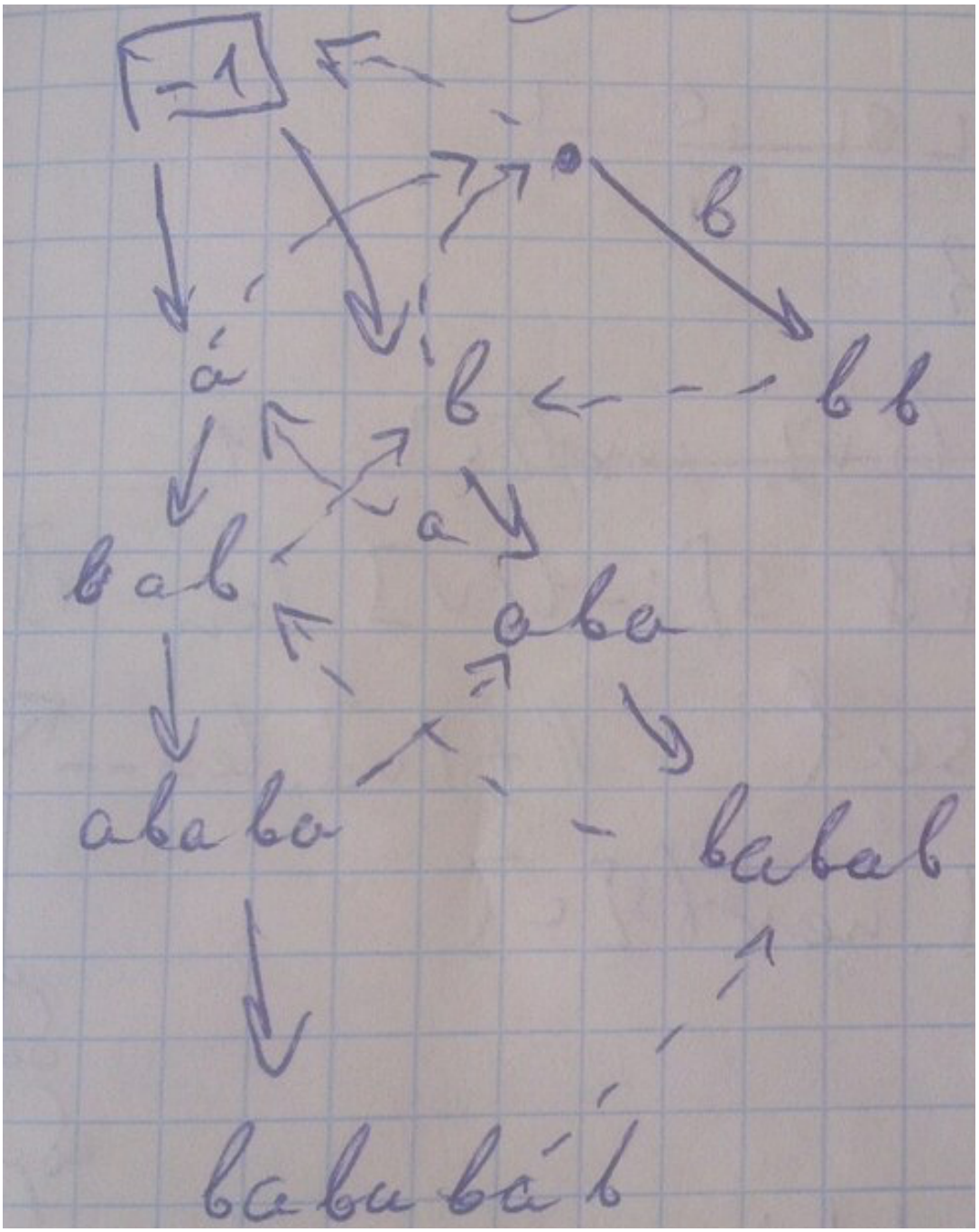


Хотим посчитать число различных палиндромов, используя R.

```
a = b = 0;
for (i, n) {
    for (k = R[i], 0) {
        if ((i - k, ..., i + k) - новый) {
            добавим
        }
        else {
            break; //меньшие уже добавили
        }
    }
}
```

работает за $O(n)$ т.к. различных палиндромов $O(n)$

Хотим построить дерево палиндромов. Вершины - палиндромы. Из палиндрома A есть ребро b в палиндром C, если $bAb = C$. Из палиндрома A есть суффиксная ссылка в палиндром C, если $A = BC$ и C максимально. Есть вершины . и -1.




```
Vertex{
suf, len, next[26];
count;
};
Vertex t[n + 2];

add(s, i, c) {
    while (v $\neq$ 0 && s[i - t[v]].len - 1] $\neq$ c) {
        v = t[v].suf;        //t[v].len--;
    }
    int  &x = t[v].next[c];
    if (x == -1) {
        x = vn++;
        t[x].len = t[v].len + 2;
```

```
        if (v == 0) {
            t[x].suf = 1;
        }
        else {
            v = t[v].suf;
            while (--||--) {
                --||--;     \\такой же самый whilie
            }
            t[x].suf = t[v].next[c]; //+2
        }
    }
    v = x; //+2
    // строчки ниже для подсчёта числа различных палиндромов
    t[v].count++;
    for (v, t[v].len) {
        t[t[v].suf].count += t[v].count
    }
}
```

Время работы $O(n)$, т.к. t[v].len уменьшается не больше, чем увиличивается, а увеличивается по $+2$ $O(n)$ раз.

```
getNext(v, c) {
    int & r = next[v, c];
    if (r == -1) {
        r = getNext(getSuf(v), c);
    }
    return r;
}

getSuf(v, c) {
    int &r = suf[v];
    if (r == -1) {
        r = getNext(getSuf(p[v]), c[v]);
    }
    return r;
}
```

10.3. Бор, Ахо-Корасик

10.3.1. Бор

Def 10.3.1. Бор - структура для хранения множества строк. Дерево, на каждом ребре которого написана буква. Каждой вершине дерева соответствует строчка - последовательность символов на пути до нее от корня. Вершины, соответствующие строкам из множества помечаются терминальными.

Добавление строки:

```
struct Node {
    static const int ALPHABET = 26;
```

```
    int go[ALPHABET];
    bool is_terminal;

    Node() : is_terminal(false) {
        memset(go, -1, sizeof(go));
    }
};

Node nodes[MAX_SIZE];
int size = 1;

void add(const string &s) {
    int v = 0;
    for (int i = 0; i < s.size(); ++i) {
        int &ref = nodes[v].go[s[i] - 'a'];
        if (ref == -1)
            ref = size++;
        v = ref;
    }
    nodes[v].is_terminal = true;
}
```

Аналогично выглядит проверка строки на принадлежность множеству - смотрим пришили бы мы в терминальную вершину, если бы добавляли эту строку.

Хранение бора:

Есть несколько вариантов хранения ребер (V - число вершин, Σ - размер алфавита, LEN - длина обрабатываемой строки):

	Mem	Time	Примечания
`Array`	$\Sigma \cdot V$	LEN	
`map<int, int>`	V	$LEN \log \Sigma$	
`unordered_map<int, int>`	$V \cdot O(1)$	$LEN \cdot O(1)$	констаты могут быть заметными
`list< pair<int, int> >`	V	$LEN \cdot \Sigma$	ничем не лучше map
`SplayTree`	V	$LEN + \log \Sigma$	что это?

Def 10.3.2. Сжатый бор это когда на ребрах написано не по одному символу, а какие-то подстроки данных строк.

В реализации можно например хранить все добавленные строки и задавать надпись на ребре тремя числами - номер строки из которой взята подстрока, позиция первого символа подстроки и ее длина.

Добавление новой строки создает в сжатом боре максимум две новые вершины - терминальную и еще одну в том месте, где произошло первое несовпадение (если там не было вершины).

Использование бора для сортировок и set, map:

Бором можно сортировать строчки - добавим их все в бор и обойдем его в глубину (идем сначала по ребрам с меньшим символом). Тот порядок в котором мы встретим терминальные вершины - лексикографический порядок строк, им соответствующих.

Бор также можно использовать вместо `unordered_map<string, T>`, если в терминальных вершинах хранить значения типа T. Вроде получится даже быстрее.

10.3.2. Суффиксное дерево

Def 10.3.3. Суффиксное дерево некоторой строки - это бор построенный на всех ее суффиксах. Для строки длины n размер несжатого суффиксного дерева $O(n^2)$, сжатого $O(n)$. Мы уже умеем строить его за $O(n^2)$ просто добавив все суффиксы.

Любая подстрока - это префикс некоторого суффикса строки, значит в суффиксном дереве ей соответсвует какая-то вершина. Поэтому построив суффиксное дерево, можем понять количество различных подстрок (число вершин в дереве) и для каждой подстроки получить `count[v]` - число вхождений подстроки - это число терминальных вершин в поддереве v, L[v], R[v] - самое левое и правое вхождения.

$SA + LCP \leftrightarrow ST$

LCP двух суффиксов - это LCA их вершин в суффиксном дереве, умеем считать в offline за $O(1)$, значит LCP всех соседних суффиксов можем получить за $O(n)$. Обойдя суффиксное дерево мы получим и отсортированный массив суффиксов за $O(n)$.

Можно сделать и обратное преобразование - зная суффиксный массив p_i и LCP соседних суффиксов $lcp(p_i, p_{i+1})$ построить сжатое суффиксное дерево за $O(n)$.

Алгоритм: откладываем терминальные вершины суффиксов в лексикографическом порядке. Сначала откадываем от корня ребро в первую терминальную вершину и пишем на нем суффикс p_1. В процессе алгоритма храним стек вершин на пути от корня до очередной терминальной вершины. Чтобы добавить очередной суффикс p_i (когда все меньшие уже добавлены) мы должны поднятся на пути от корня до p_{i-1} до глубины $lcp(p_{i-1}, p_i)$, вынимая из стека все более глубокие вершины, и, если это середина ребра, создать там новую вершину, от котрой отложить терминальную, соответствующую p_i.

10.3.3. Поиск словарных слов в тексте

Задача: дан текст T и набор словарных слов $S_1, S_2, \dots, S_n$ (W - максимальная длина, L - суммарная длина). Нужно найти в T хоть одно слово из S.

Замечание: подразумевается, что текст - это непрерывный набор символов в котором мы ищем подстроку совпадающую со словарной. Если под текстом понимается набор слов, то уже умеем решать задачу хеш-таблицей за $O(|T| + L)$.

Алгоритм 1. Решение хешами.

Можно проверять вхождениее каждого слова отдельно алгоритмом Рабина-Карпа за $O(n|T|)$.

Можно за один проход проверить на вхождение все слова одной длины. Для этого создаем хеш-таблицу с хешами всех таких слов и так же как в Рабине-Карпе идем по тексту "окном"проверяя соотвестсвует ли текущий хеш какому-нибудь из таблицы. Запускаем цикл по всем длинам за $O(W|T|)$. Если быть точнее, это работает за $O(D|T|)$, где D - число различных длин среди словарных слов. $D < \sqrt{2L}$, так как даже если все длины различны и минимальны $L = \sum_i |S_i| = 1 + 2 + \dots + D = \frac{D(D+1)}{2} > \frac{D^2}{2}$.

Алгоритм 2. Решение бором.

Строим бор на всех словах словаря.

FOR $i = 0 \dots |T| - 1$

Встаем в корень бора. Идем по тексту начиная с i-й позиции и одновременно спускаемся в боре по соответствующему символу. Пришли в терминальную вешину - нашли вхождение. Если соответсвующего ребра нет, переходим к следующему i.

Time $= O(W|T|)$

Алгоритм 3. Ахо-Корасик

Сделаем из нашего бора детерменированный автомат. У нас уже есть вершины-состояния и переходы по некоторым символам. Мы будем идти по тексту и переходить по соответствующему

символу в автомате. Мы хотим, чтобы автомат был устроен так, что текущее состояние автомата (после того как ему передали некоторый префикс) соответствовало строке - максимальному суффиксу этого префикса, который встречается в боре (то есть максимальному суффиксу этого префикса, который является префиксом некоторого слова словаря). Иначе говоря для словаря, состоящего из одного слова, мы хотим чтобы состояние в автомате, после того как мы передадим $[T_0, T_i)$ соответствовало префикс-функции строки $S_0\#[T_0, T_i)$, а если в словаре больше слов, то это должен быть аналог этой префикс-функции для случая, когда слева от символа '#' стоит многообразие всех слов словаря.

Итак, те переходы, которые уже есть в боре вполне законны (в некоторой строке после максимального совпадающего префикса идет нужный символ), осталось добавить переходы по всем оставшимся символам из каждого состояния. Чтобы сделать это, введем понятие суффиксной ссылки.

Суффиксная ссылка от вершины бора v будет указывать на такую вершину бора u, которая соответствует собственному суффиксу v-строки, а среди таких выбирается суффикс максимальной длины (то есть самая глубокая вершина u удовлетворяющая условию). Такая ссылка определена для всех некорневых вершин (потому что есть пустой суффикс - корень), для корня ее можно пустить в сам корень.

Теперь если из данноого состояния нет перехода по соответствующему символу, мы должны перейти по суффиксной ссылке и попытаться совершить переход из нее. Проходя по суффиксным ссылкам вверх, мы либо найдем нужное ребро, либо рано или поздно попадем в корень. Для корня разумно добавить ребра по недостающим символам, ведущие в себя (в этом случае больше чем пустой суффикс получить не удастся). Итак, если мы сможем посчитать суффиксные ссылки $suff[v]$ для всех вершин бора, то мы сможем делать переходы по любым символам:

```
while (nodes[v].go[c] == -1)
    v = suff[v];
v = nodes[v].go[c];
```

Исходя из структуры автомата, мы можем вычислить суффиксную ссылку для вершины, если знаем суффиксные ссылки для всех вершин меньшей глубины. А именно возьмем у нашего родителя суффиксную ссылку и перейдем от этой вершины по символу на ребре из нашего родителя в нас.

```
x = suff[parent[v]]
while (nodes[x].go[c[v]] == -1)
    x = suff[x];
suff[v] = nodes[x].go[c[v]];
```

Таким образом, чтобы построить автомат Ахо-Корасик на данном словаре сначала построим на нем бор за $O(L)$, установим `suff[root] = root`, а затем обойдем бор bfs-ом и для каждой вершины вычислим ее суффиксную ссылку кодом выше. Заметим, что подсчет суффиксных ссылок всех вершин на любом пути от корня до какой-то вершины v работает за $O(depth(v))$ - после того как поднимемся на какую-то глубину d для подсчета ссылки очередной вершины, подсчет ссылки следующей на пути начнем с этой глубины d, то есть мы не теряем прогресс и подъемов делаем не больше, чем спусков, и всего $O(depth(v))$ действий. Если возьмем пути до всех терминальных вершин, то покроем весь бор, а значит таким образом мы посчитаем ссылки также за $O(L)$.

Теперь можем решить исходную задачу. Конечными состояниями автомата логично принять те, которые являются терминальными вершинами, или те, из которых до какой-то терминальной вершины можно добраться по суффиксным ссылкам (то есть это слово из словаря - один из суффиксов текущего префикса). Конечность вершинам можно так же расставить во время `bfs` вместе с вычислением суффиксных ссылок. Теперь будем последовательно переходить в автомате по символам из текста и, придя в конечное состояние получим вхождение словарного слова. Не смотря

на то, что некоторые переходы могут занять у нас не $O(1)$ можно так же заметить, что подъемов мы сделаем не больше чем спусков и весь алгоритм работает за $O(L + |T|)$.

С помощью этого автомата можно решать некоторые модификации данной задачи. Например, для каждого словарного слова узнать сколько раз оно встречается в тексте. Снова пройдемся в автомате по всем символам текста и, заходя в очередное состояние v, будем делать `count[v]++`. Теперь поймем, что мы встречали вхождение некоторого слова, соответсвующего терминальной вершине t, тогда и только тогда, когда из текущего состояния v можно было попасть в t по суффиксным ссылкам: в эти моменты слово является суффиксом строки, соответствующей v. Поэтому чтобы получить ответ на задачу мы должны протолкнуть count вверх - идя снизу для каждой вершины сделать `count[suff[v]] += count[v]` и вернуть значения count в терминальных вершинах. Если вместо `count[v]` поддерживать первый/последний момент времени, когда мы побывали в этом состоянии, а в конце так же проталкивать эти значения по суффиксным ссылкам выбирая минимум/максимум, то получим для всех слов первое/последнее вхождение в текст соответственно.

Можно также не вычислять все суффиксные ссылки заранее, а воспользоваться принципом ленивой динамики и изначально считать все ссылки (кроме корневой) неопределенными и вычислять ссылку рекурсивно в первый раз, когда она понадобилась.

10.4. Построение суффиксного дерева за $O(n)$

Алгоритм Укконена умеет строить суффиксное дерево за $O(n)$
КАРТИНКА СУФФИКСНОЕ ДЕРЕВО ДЛЯ ABAC#

Замечание 10.4.1. Количество различных подстрок $= \sum_e len(e)$

Будем строить суффиксное дерево последовательно слева направо, каждый раз добавляя очередной символ, достраивая все суфффиксы в дереве

$$S \to addLetter(c) \to Sc$$

Алгоритм

1. $S \to S\#$, добавим фиктивный символ в конец, чтобы каждый суффикс стал в итоге листом
2. Суффиксы растут при $S \to Sc$
   КАРТИНКА ПЕРЕХОД ИЗ ABA К ABAA
   - Поддерживаем $p[]$ — концы суффиксов в дереве
   - Создаем новую вершину только на разветвлении, поэтому $O(n)$ памяти
   - Текущее время: $O(n^2)$

   Цикл жизни любого суффикса:

   (a) Спуск по дереву
   (b) Развилка
   (c) Лист

Лемма 10.4.1. $S \to Sc$, Лист $\to$ Лист

3. Листья растут сами

$$Edge \begin{cases} [i, i+len) — \text{подотрезрок строки} \\ char = s[i] — \text{первый символ на ребре} \\ to — \text{номер вершины на конце ребра} \end{cases}$$

Добавление символа к листам в дереве есть простое увеличение длинны подотрезка. Скажем, что $len = \infty$ (1-ый способ) или что $len = [n - i]$ (2-ой способ), тогда листья будут расти сами Рассмотрим три фазы жизни любого суффикса. Мы уже сказали, что не спускаемся по дереву (есть $p[]$), также мы не пробегаемся по листьям, осталось учесть суффиксы, которые будут разветвляться

Лемма 10.4.2. i-ый суффикс разветвился $\Rightarrow \forall j < i$ разветвился

▶ Если суффикс не разветвился (и не лист) $\Leftrightarrow$ он встречался раньше как подстрока. Пусть какой-то j-ый суффикс не разветвился, значит он встречается ранее как подстрока, т.е $s[j : n] = s[x : x + (n - j)]$, знаем, что $j < i$, поэтому $s[i : n] = s[x + (i - j) : x + (n - j)]$, значит и i-ый суффикс не разветвился. Противоречие с начальным условием
КАРТИНКА ДОКАЗАТЕЛЬСТВА ЛЕММЫ ◀

Таким образом, надо поддерживать самый длинный неразретвившийся суффикс, будем хранить его в переменной pos, как комбинацию вершины дерева, символа по которому надо пойти и сдвига на ребре (pos может быть на середине ребра), i — позиция начала суффикса

4. $S \to Sc$, что происходит с pos?
$\forall j < i$ — уже листья
При добавлении нового символа может быть 2 случая:

   a) все $j \geqslant i$ спускаются

   b) i-ый разветвился

```
while(не могу спуститься вниз){
    Разветвись
    i++;
    pos = suf[pos]
}
Спустись из pos по символу c
```

Итак, нужно уметь взять суффиксную ссылку от pos

5. Суффиксные ссылки
$\forall$ вершины храним $suf[v]$ В процессе построения дереве уже имеющиеся суффиксные ссылки никак не меняются, как построить суфф. ссылки для уже имеющихся вершин? Суффиксную ссылку для созданной вершины будем проставлять при продление следующего суффикса.

Лемма 10.4.3. $suf[v]$ — вершина, а не середина ребра
КАРТИНКА СУФФИКСНЫЕ ССЫЛКИ

Вот как мы берем суффиксную ссылку от $pos =< v, char, shift >$

```
v = suf[v];
while (shift > 0){
    e = edge[v][s[k - shift]];
```

```
4           shift -= e.len
5           v = e.to
6           //последний прыжок неправильный, но понятно, как переделать
7       }
```

Теорема 10.4.1. Построение суфф. дерева работает за $O(n)$

▶ Введем потенциал $\varphi =$ количество вершин на пути от корня к v (v это начало ребра, на котором находится pos). При спуске потенциал увеличивается на 1, при взятии суффиксной ссылки уменьшается на 1, также знаем, что $\varphi \leqslant n$ (самый длинный суффикс), поэтому $cnt(++) \leqslant n + cnt(--)$, откуда получаем оценку $O(n)$ ◀

10.5. Суффиксный автомат (нет в билетах)

Def 10.5.1. Детирминированный конечный автомат = орграф с выделенной стартовой вершиной и множеством терминальных вершин
$S \in V$ — стартовая вершина
$T \subseteq V$ — множество терминальных вершин
ИЗОБРАЖЕНИЕ ДКА
ДКА принимает строку и двигается по ребрам, соотвутствующим очередному символу

Def 10.5.2. Суфф. автомат — автомат, который принимает только суффиксы

Алгоритмы построения

1. Посжимаем одинаковые вершины в одну
   ИЗОБРАЖЕНИЕ СЖАТИЯ (10 шакалов из 10)

2. $S \to Sc$

   Def 10.5.3. $R_s(v) = \{w \mid vw$ — суфф. $s\}$ (множество правых контекстов строки)

   Пример 10.5.1. $s = ababa$
   $R_s(a) = \{\varepsilon, ba, baba\}$
   $R_s(aba) = \{\varepsilon, ba\}$

   Лемма 10.5.1. v — суфф. $u \Rightarrow R_s(u) \subseteq R_s(v)$

   Лемма 10.5.2. $R_s(u) = R_s(v), |u| \geqslant |v| \Rightarrow v$ — суфф. u

   Лемма 10.5.3. $R_s(v) \cap R_s(u) \Rightarrow (R_s(v) \subset R_s(u), u$ — суфф. $v)$ ИЛИ $(R_s(u) \subset R_s(v), v$ — суфф. $u)$

$$\text{Суфф. автомат} \begin{cases} next[v, c] \\ suf[v] \\ len[v] \\ last \to T \end{cases}$$

   Пусть $last$ — вершина, которой соответствует s, тогда множество терминальных вершин это $T = \{last, suf[last], suf[suf[last]], \dots\}$

   Лемма 10.5.4. В суффиксном автомате $\leqslant 2n$ вершин (без доказательства)

10.6. Построение суффиксного массива за $O(n)$

10.6.1. `Get` — нахождение подстроки

Решение, использующее суффиксное дерево

Напоминание: суффиксное дерево — сжатый бор всех суффиксов строки (умеем строить за $O(n)$).

Решим вначале задачу поиска подстроки:

Online:

1. Строим суффиксное дерево
2. Откладываем от корня строчку
3. Если остались в дереве - ok

   Если нужно посчитать число вхождений: $count = size[v]$, где $size[v]$ — размер поддерева вершины v

Решение, использующее суффиксный массив

Напоминание суффиксный массив (suffix array далее SA) — это отсортированный массив суффиксов.

Binary Search: Выполним бинарный поиск по SA, как результат мы получим отрезок ($L = lower\_bound, R = upper\_bound$), в котором все суффиксы начинаются с нужной нам строчки. За сколько это работает?

- $O(\underbrace{|s_i|}_{\text{сравнение}} \underbrace{\log |text|}_{\text{BS}})$ — если сравнивать строчки просто по символьно
- $O(\underbrace{|s_i|}_{\text{предподсчёт хешей}} + \underbrace{\log |s_i| \log |text|}_{\text{сравнение + BS}})$ — сравниваем, используя хеши

SA + LCP: Теперь решим эту задачу, используя SA и LCP.

Обозначения:

- $LCP(i, j) = \min_{k \in [i, j+1)} LCP(k, k+1)$
- $LCP(L, S) = SL$
- L — левая граница бинарного поиска
- R — правая граница бинарного поиска
- M — середина

Будем хранить SA используя массив p (перестановка индексов) типа `int`, где значение ячейки — начало суффикса.

LCP — только соседних суффиксов.

Нужно оптимизировать бинарный поиск.

Заметим, что:

$$SM \geq \max \begin{cases} \min(SL, LM) \\ \min(SR, RM) \end{cases}$$

Тогда можно записать следующее:

```
SM = max(min(SL, LM), min(SR, RM));
while (s[SM] == m[SM]) {   // пока максимум увеличивается - пусть увеличивается
    SM++;
    if (s[SM] > m[SM])
        L = M, SL = SM;
    else
        R = M, SR = SM;
}
```

Нужно за O(1) находить LM и MR. Умеем, используя Фарах-Колтон-Бендера, но будем проще — используем дерево отрезков. Поймём, что отрезок BS — это некоторая вершина ДО, тогда построим ДО на суффиксном массиве. Минимум на отрезке хранится в нужной вершине ДО.

Время работы алгоритма: $O(|s_i| + \log |text|)$

10.6.2. Построение за SA за $O(n \log n)$

Все индексы брать по модулю

Сейчас мы научились искать подстроку используя SA, научимся же строить SA.

Сделаем все суффиксы одинаковой длины — допишем в конец нули. Заметим, что:

1. циклические сдвиги строки s — это подстроки строки ss.
2. если их отсортировать и отбросить всё, что будет стоять после дописанных нулей, то мы получим SA

Исходя из этого мы можем сделать следующее: приписать в конец символ #, который будет самым 'лёгким' и будем рассмотрим Digital Sort циклических сдвигов строки этой строки.

Digital Sort работает за $O(n(n + \Sigma))$:

```
for (int i = n - 1; i >= 0; ++i)
    countSort(i) // цифровая сортировка по i-ому символу
```

Сейчас наша задача ускорить сортировку. На данный момент сортировка совершает переход: $k \to k + 1$, научимся делать $k \to 2k$.

Рассмотрим диапазон $[i, i + k)$: занумеруем его элементы следующим образом:

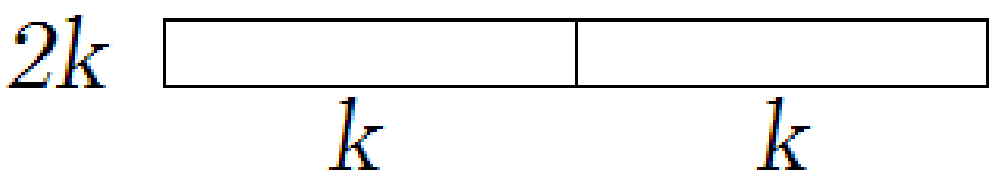
$$color[i] < color[j] \Leftrightarrow strcmp(s + i, s + j, k) < 0$$

База:

1. $O(n + \Sigma)$ — Count Sort
2. Занумеровали

Переход:

Строка длины $2k$, состоит из 2х строк длины k, сравнить между собой мы их можем за $O(1)$, зная значения $color$ и p



Тогда, чтобы отсортировать все блоки длины $2k$, их нужно отсортировать по парам значений $\langle color[i], color[i + k] \rangle$

Когда алгоритм останавливается? Когда $k \geq n$ — тогда можем сказать, что циклические сдвиги длины n уже отсортированы, что нам и было нужно.

Реализация:

```
// в конце строки s - терминальный нуль
// n - длина строки с учётм терминального
// t - временный массив, то для цветов, то для нового суфф-массива
// p - суфф-массив
// cnt - счётчик count-sort'a
// color - массив цветов
// C - число различных цветов

for (int i = 0; i < n; ++i)
        ++cnt[s[i] + 1];
for (int i = 1; i < 127; ++i)
        cnt[i] += cnt[i - 1];
for (int i = 0; i < n; ++i)
        p[cnt[s[i]]++] = i;
for (int i = 1; i < n; ++i)
        color[p[i]] = color[p[i - 1]] + (s[p[i - 1]] != s[p[i]]);

int C = color[p[n - 1]] + 1;
for (int k = 1; k < n; k *= 2) {
        memset(cnt, 0, sizeof(cnt));
    for (int i = 0; i < n; ++i)
            ++cnt[color[i] + 1];
    for (int i = 1; i < C; ++i)
            cnt[i] += cnt[i - 1];
    for (int j, i = 0; i < n; ++i) {
            j = (p[i] - k + n) % n;
            t[cnt[color[j]]++] = j;
    }

    memcpy(p, t, sizeof(t));

    t[p[0]] = 0;
    for (int i = 1; i < n; ++i) // пересчёт цветов
        t[p[i]] = t[p[i - 1]] + (color[p[i]] != color[p[i - 1]] || color[(p[i] + k) % n] !=
    C = t[p[n - 1]] + 1;

    memcpy(c, t, sizeof(t));
}
```

Приведённая выше сортировка нестабильна, чтобы сделать сортировку стабильной нужно вконце отсортировать по этой паре $\langle color[i], i\rangle$ за $O(n)$ (перебираем строки по i и сортируем подсчётом по $color[i]$).

Optimize 1:

```
    if (color[p[n-1]] == n - 1) break;
        // если все строки различные, т.е. и по первым k символам
        // мы их правильно отсортировали
```

Optimize 2:

```
if (period < n && period / n)
    sort(period);
```

10.6.3. Алгоритм Касаи — LCP за $O(n)$

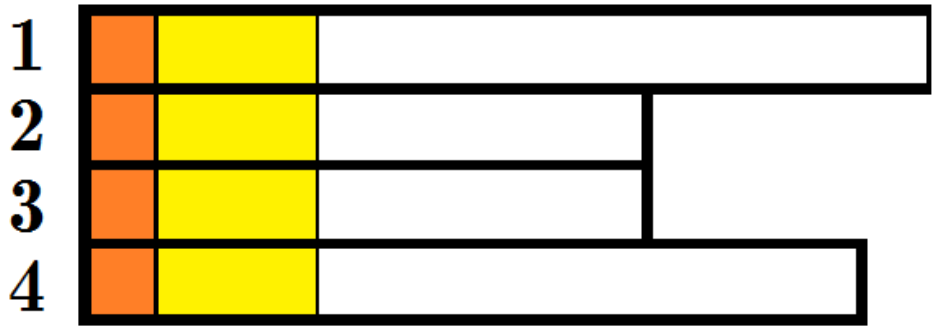
Алгоритм Касаи позволяет найти LCP для соседних элементов SA.

SA — массив p, тогда пусть $\tilde{p}[p[i]] = i$ — т.е. позиция в суффиксном массиве другими словами обратная перестановка.

Заметим, что:

1. $LCP[p[i], p[j]] = \min_{k \in [i,j)} LCP(p[k], p[k+1])$
2. если LCP между парой суффиксов больше 1, то когда мы удалим первый символ — порядок суффиксов не изменится:

$$LCP(p[\tilde{p}[i]], p[\tilde{p}[i]+1]) = LCP(i, p[\tilde{p}[i]+1]) > 1 \Rightarrow s[i+1:] < s[p[\tilde{p}[i]+1]+1:]$$



3. у этих же суффиксов без первого символа LCP на 1 меньше:

$$LCP(p[\tilde{p}[i]], p[\tilde{p}[i]+1]) = LCP(i, p[\tilde{p}[i]+1]) > 1 \Rightarrow LCP(i, p[\tilde{p}[i]+1]) = LCP(i+1, p[\tilde{p}[i]+1]+1)+1$$

Теперь, будем вычислять LCP в порядке уменьшения длины суффикса.

Пусть мы знаем $LCP(p[\tilde{p}[k]], p[\tilde{p}[k]+1]) = LCP(k, p[\tilde{p}[k]+1])$, как нам выразить из того что записано выше $LCP(k+1, p[\tilde{p}[k+1]+1])$?

$$LCP(k, p[\tilde{p}[k]+1]) - 1 = LCP(k+1, p[\tilde{p}[k]+1]+1) =$$
$$= \min(LCP(k+1, p[\tilde{p}[k+1]+1]), ...) \le LCP(k+1, p[\tilde{p}[k+1]+1])$$

Из неравенства следует, очередной раз считать LCP нкжно не с первого символа, а с $LCP(k, p[\tilde{p}[k]+1]) - 1$. Таким образом, идея следующая: инициализируем начальное значение LCP (в коде ниже обозначена, за `k`) нулём и увеличиваем, пока можем.

(в коде ниже $\tilde{p} \equiv$ `p_`)

```
for (int i = 0; i < n; ++i)
        p_[p[i]] = i;       // получаем обратную перестановку

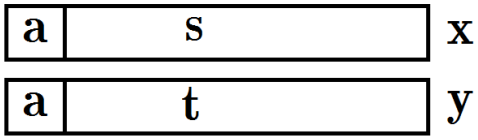
for (int j, k = 0, i = 0; i < n; ++i) {
        j = p_[i];
```

```
        if (k > 0) --k;
        while (j + 1 < n && s[i + k] == s[(p[j + 1] + k)])
            ++k;
        LCP[j] = k;
    }
```

Комментарии для циклических сдвигов: *мне кажется что здесь лажа, проосъба: если вам лень исправлять, то хотя бы скажите, как правильно*

Из-за того, что используем, что после удаления символа строки идут в том же порядке, алгоритм не работает на циклических сдвигах.

Рассмотрим строчки s и t, если это суффиксы, то всегда выполняется $s < t$. Если это циклические сдвиги, то может $s = t$. Пусть сортировка стабильна.



x – позиция цикл сдвиг as, y – если удалим at, то перейдём

$$\langle as, x\rangle, \langle at, y\rangle, as = at \Rightarrow x < y$$

$i + 1 \to s$
$j + 1 \to t$
$y + 1 > x + 1$
Не всегда выполняется, т.к. индексы берутся по модулю.
Нужно добавить в код перед `while`:

```
    if (p[j] == n - 1 || p[j + 1] == n - 1)
        k = 0
```

Время работы: Для циклических сдвигов: (число увеличений `k`) $\leq n$+число уменьшений `k` $\leq 2n \Rightarrow O(n)$

Для циклических сдвигов: (число увеличений `k`) $\leq n +$ число уменьшений `k` $\leq 4n \Rightarrow O(n)$

10.6.4. Карккайнен-Сандерс. Построение SA за O(n)

Время работы: $T(n) = O(n) + T(2/3n) + O(n) \Rightarrow T(n) = O(n)$
$|s| = n$
$0 \leq s_i \leq 2n$

Алгоритм (сортировка суффиксов):

1. Занумеруем символы за $O(n)$ — перешли к числовому алфавиту. Допишем в конец строкеи 3 нуля.

2. Разобьём на тройки символов, пусть тройка символов — новый сивол, т.е. мы получили строку в новом алфавите.

$$\forall i\colon s_i s_{i+1} s_{i+2} = w_i$$
$$0 \le w_i < n \Rightarrow O(n)$$

   Строчку можно представить 3 типами:

   (a) $w_0 w_3 w_6 \ldots$ — просто вся строка

   (b) $w_1 w_4 w_7 \ldots$ — без первого символа

   (c) $w_2 w_5 w_8 \ldots$ — без первых двух символов

   Суффиксом n-ого типа, будем называть суффикс n-ого представления строки, можно заметить что в обычном представлении индексы этих суффиксов дают остатки по модулю 3 равные номеру их типа.

3. Пусть

$$A = \underbrace{\underbrace{w_0 w_3 w_6 \cdots}_{s} \# \underbrace{w_1 w_4 w_7 \cdots}_{s[1:]}}_{\frac{2}{3}n}$$

4. Делаем рекурсивный вызов от A, так мы отсортировали суффиксы 0-ого и 1-ого типов. На выходе мы получили суффиксный массив для этой строки.

5. Заметим, что суффикс строки типа 2 — это символ + суффикс типа 0 $\Rightarrow$ т.к. положение суффиксов типа 1 известно суффиксы типа 2 можно сортирвоать как пару DigitalSort:

$$\text{тип 2} = \langle \text{символ}, \text{суффикс A, точнее положение в SA} \rangle \Rightarrow \text{Digital Sort} \Rightarrow O(n)$$

6. Теперь нужно просто слить 2 отсортированных массива: Merge(Sorted(тип 0-1), Sorted(тип 2)) $\Rightarrow O(n)$

   Какой как сравнивать? Мы можем использовать преобразование между суффиксами разных типов.

   Если 0 и 2 типы:

$$\text{их сравнивать умеем: } \begin{cases} 0 \to 1 \\ 2 \to 0 \end{cases}$$

   Если 1 и 2:

$$\text{их сравнивать умеем: } \begin{cases} 1 \to 2 \to 0 \\ 2 \to 0 \to 1 \end{cases}$$

10.7. Хеши

10.7.1. Строка Туэ-Морса

Def 10.7.1. Два определения строки Туэ-Морса:

1. s[i] = bitcount(i) % 2
2. $s_{k+1} = s_k$ + not s_k

Обозначим <P,M> : $\sum s_i \cdot p^i \ mod \ M$
Строки Туэ-Морса можно использовать для антихеш-тест для $M = 2^m$ (для любого P)
$H_1 = Hash(s[0, 2^k])$, $H_2 = Hash(s[2^k, 2^{k+1}])$
$H_1 = H_2$ для $k \geqslant ?$
Проверим для какой длины случится коллизия.

$$H_1 - H_2 = 0 \mod 2^m$$

$$s[2^k, 2^{k+1}] = \not{s}[0, 2^k] \Rightarrow \sum p^i \cdot (\pm 1) = 0 \mod 2^m$$

$$\sum p^i \cdot (\pm 1) = -p^0 + p^1 + p^2 - p^3 + ... = -(1-p)(1-p^2)(1-p^4)...(1-p^{2^{k-1}})$$

$1-p^4 = (1-p^2)(1+p^2) = (1+p^2)(1+p)(1-p)$, каждая из скобок четная $\Rightarrow$ получаем $1+2+3+..k = \frac{k(k+1)}{2} \geqslant \frac{k^2}{2} \Rightarrow$ при $Len = 2^k$ хеши равны по модулю $2^{\frac{k^2}{2}}$

$m = 2^{k^2/2} \Rightarrow k = \sqrt{2m}$
$m = 64 \Rightarrow k = \sqrt{128} \approx 12$, $2^k = 4096$
Нужна строка длины 2^{k+1}, значит 8192 хватит.

Замечание 10.7.1. $p = 2k$: $p^m \equiv 0 \mod 2^m$

10.7.2. Алгоритм построения коллизии для <P,M> хеша

Хотим найти стороки s и t, такие что $Hash(s_1, \cdots, s_k) = Hash(t_1, .., t_k)$
Алфавит $\Sigma = \{a, b\}$
Если мы сможем подобрать такие коэффициенты(не все 0), то сможем найти исходные строчки(как, покажем потом):

$$\sum p^i \cdot (-1, 0, 1) = 0 \mod M$$

$A_0 = \{p^0, p^1, \cdot, p^{k-1}\}$ //все по модулю M
Если среди элементов A_i есть 0, то все хорошо.)
Получение A_{i+1} :

1. Sort(A_i)
2. $A_{i+1} = \{A_i[1] - A_i[0], A_i[3] - A_i[2], \cdots\}$

Размер множетсва уменьшился в два раза $|A_{i+1}| = \frac{|A_i|}{2}$
Диапазон:

A_0**:** $0 \cdots M-1$

A_1**:** k случайных чисел, $E(x_i) = \frac{M \cdot i}{k+1}$, диапазон $A_1 \sim \frac{M}{k+1}$

A_2**:** $|A_1| = \frac{k}{2} \Rightarrow$ диапазон $A_2 \sim \frac{M}{(k+1)(k/2+1)}$

A_i**:** диапазон делится на $\frac{k}{2^{i-1}}$ $+1$

$$M \leqslant \frac{k}{1} \cdot \frac{k}{2} \cdot \frac{k}{4} \cdots \approx \frac{k^{logk}}{2^{\frac{\log^2 k}{2}}} = 2^{\frac{log^2 k}{2}} \Rightarrow M = 2^{\frac{(log^2 k)}{2}}, \log M = \frac{\log^2 k}{2}$$

$$\log k = \sqrt{2 \log M}$$

$$k = 2^{\sqrt{2 \log M}}$$

$M = 2^{64}$: $k = 2^{\sqrt{2 \cdot 64}} \approx 2^{12} = 4096$

Пример 10.7.1. Элементы A_i можно расписать как многочлен.
$1 - p + p^2 = 0, 1 + p^2 = p$: $bab = aba$ (b обозначает 1, a — 0)

10.7.3. Леммы про вероятность коллизии при выборе многочлена, или точки, или обоих случайными

P - точка, M - модуль
S - многочлен(строка) с коэффициентами 0..M-1
k - степень многочлена(длина строки)

Лемма 10.7.1. <P,M> - fixed, s - random, тогда Pr[s(P) = 0] = $\frac{1}{M}$, Hash(s) = 0
s(x) = (x-p)t(x) + a_0 ← равномерно распределены от 0 до M-1, значит верно

Лемма 10.7.2. <s,M> - fixed, P - random, тогда Pr[s(P) = 0] $\leqslant \frac{k}{M}$
Pr[s(P) = 0] = $\frac{rootnumber}{M}$, M - простое ⇒ корней $\leqslant k$

Замечание 10.7.2. M = 2^{32}, $x^{32} = 0 \mod M$ имеет 2^{31} корней

Лемма 10.7.3. M - fixed, <s,P> - random. Pr[s(P) = 0] = $\frac{1}{M}$ ⇔ у random s(x) ∼ 1 корень
$\sum_{P=0}^{M-1} \frac{1}{M} \cdot \frac{1}{M} = \frac{1}{M}$ через Lm1

Выводы:

1. M – простое $\sim 10^9 + 7$
2. P_1 - random
3. P_2 - random

10.7.4. Хеш-таблица

m списков, всего n чисел: длина одного списка $\sim \frac{n}{m}$
Пусть $m = n$

Лемма 10.7.4. E($Len_i = 1$) (бессмысленна, т.к все равно могут быть все элементы в одном списке)

Лемма 10.7.5. E(max Len_i) = $O(\log n)$

Hash = Random

$$Pr\{len_i = k\} = \frac{C_k^n \cdot (n-1)^{n-k}}{n^n} \leqslant \frac{n^k}{k!} \cdot \frac{n^{n-k}}{n^n} = \frac{1}{k!}$$

Неправильное док-во:

$$E(Len_i \geqslant k) \leqslant n(\frac{1}{k!} + \frac{1}{(k+1)!} + \cdots) \leqslant \frac{2n}{k!}$$

$$E(Len_i \geqslant k) = 1 \colon 2n = k!$$

$$2n = (\frac{k}{e})^k \cdot \frac{1}{\sqrt{2\pi k}}, k^k = 2^{k \log k}, 2n = 2^{k \log k}$$

$$\frac{\log(2n)}{\log\log(2n)} \leqslant k \leqslant \log(2n)$$

Получаем 1 + o(1/n)

10.7.5. Двойное хеширование

$$i_1 = Hash_1(x), i_2 = Hash_2(x)$$

```
void Add(x):
    if (|List[i[i]| <= |List[i[2]|) {
        List[i[1]].pb(x)
    } else {
        List[i[2]].pb(x)
    }
```

Лемма 10.7.6.

$$E(maxLen_i) = \Theta(\log\log n)$$

Без доказательства

10.7.6. Двойное хеширование для открытой адресации

```
i = h1(x)
while(...) i++  // в обычном

while(...) i = (i + h2(x)) mod n // в двойном
```

10.7.7. Совершенное хеширование

Даны $x_1, .., x_n$. Найти $h \colon x_i \to [0..m)$ такое, что $\forall i \neq j \colon h(x_i) \neq h(x_j)$

Алгоритм 1 : Time = O(n), $m = n^2$ Pr{коллизии} $\leqslant \frac{\frac{n(n-1)}{2}}{n^2} \leqslant \frac{1}{2}$

```
while(1) {
    h = hashFunc
    if (good(h)) {
        break;
    }
}
```

Алгоритм 2 :

Space = O(n)

Time = O(n)

1. Распределили n по случайным образом по n ячейкам .

2. Пусть в i-й ячейке a[i] элементов, используем первый алгоритм, получаем $a[i]^2$ дополнительных ячеек.

n_i — количство элементов в i-ом списке.

$$E[\sum n_i^2] = E[\text{количество коллизий}] = \sum (h[i] = h[j]) = n + \frac{n(n-1)}{n} \leqslant 2n$$

Лемма 10.7.7. Маркова

$$E[X] \leqslant A \Rightarrow Pr[A \geqslant 2X] \leqslant \frac{1}{2}$$

$$\Rightarrow P[\sum n_i^2 \geqslant 4n] \leqslant \frac{1}{2}$$

10.7.8. Фильтр Блюма

Занимает m бит памяти.

Есть k хешфункций. Число добавлено, если во всех битиках, которые выдают хешфункции стоит 1.

При добавление числа ставим 1 во всех соответсвующих клетках.

$$h_1, .., h_k$$

$$h_i \colon x_j \to cell_{ij}$$

```
void init() {
    bit = {0};
}

void add(j) {
    for(int i = 1; i <= k; ++i) {
        bit[cell[i][j]] = 1;
    }
}

bool isAdded(j) {
    for(int i = 1; i <= k; ++i) {
        if (bit[cell[i][j]] = 0) {
            return 0;
        }
     }
     return 1;
}
```

Получился алгоритм с односторонней ошибкой.

Pr[Error] = Pr[isAdded(x[i]) = 1, но x[i] нет] //! с оценкой этого на лекции были проблемы

Глава 11

Ретроанализ и функция Гранди

11.1. Множества. bitset

n элементов, целые $1 \dots c$

1. sorted array $O(n+m)$

2. bitset<c> b $O(c/w)$

w – размер машинного слова.
$unsigned\, long\, \underbrace{long}_{w=64}$

Объединение множеств:

```
for (int i = 0; i < c / w; i++) {
    a1[i] |= a2[i];
}
```

bitset умеет:

1. or, and, xor, not
2. <<, >>
3. .count() (__buildin_popcount $O(\log w \cdot \frac{c}{w})$)
4. ```
   x = b[i];
   b[i] = x;
   ```

Так же его можно кастить:

```
unsigned long long *a = (unsigned long long*)&b;
```

Умножение матриц над $\mathbb{F}_2$:

```
forn(i, n) {
    forn(j, n) {
        forn(k, n) {
            c[i][j] ^= a[i][k] & b[k][j];
        }
    }
}
```
```

Заметим, что этот код эквивалентен следующему:

```
forn(i, n) {
    forn(k, n) {
        if (a[i][k]) {
            forn(j, n) {
                c[i][j] ^= b[k][j];
            }
        }
    }
}
```

Соптимизируем, используя битсеты, до $O(\frac{n^3}{w})$, вместо $O(n^3)$:

```
forn(i, n) {
    forn(j, n) {
        if (a[i][j]) {
            c[i] ^= b[j];
        }
    }
}
```

11.2. Игры

11.2.1. DAG

Пусть есть симметричная игра на ацикличном ориентированном графе:
Есть начальная вершина, игроки ходят по очереди. Проигрывает тот, кто не может сделать ход.

Решение:
Понятно, что вершины с исходящей степенью 0 изначально проигрышны(L). Дальше просто запускаем `dfs` из начальной вершины. Условия выигрышности/проигрышности вершины:
$\exists$ ход в $L \Rightarrow W$
$\forall$ ход в $W \Rightarrow L$
Т.е. $f[v] = not(and f[x_i])$

Работает, очевидно, за $O(V+E)$

11.2.2. Функция Гранди

Def 11.2.1.

$$f[v] = mex\{f[x_i]\}$$

$$mex A = min\ x : x \in (\mathbb{N} \cup 0) \wedge x \notin A$$

Замечание 11.2.1.
1. $f[v] = 0 \Leftrightarrow Lose$
2. $mex\ \emptyset = 0$
3. $0 \in A \Rightarrow mex\ A \neq 0$

11.2.3. Прямая сумма игр

$V = \langle V_1, V_2 \rangle$

$\langle v_1, v_2 \rangle \to \langle x_1, v_2 \rangle$

$\langle v_1, v_2 \rangle \to \langle v_1, x_2 \rangle$

$f[\langle v_1, v_2 \rangle] = f[v_1] xor f[v_2]$

11.2.4. Игра ним

Есть кучка из n камней. Игроки по очереди берут из нее любое количество камней. Проигрывает тот, кто не может сделать ход.

Понятно, что для одной кучки верно, что

1. $n = 0 \Rightarrow lose$
2. $n > 0 \Rightarrow win$

Сумма: n_1 и n_2

$$lose \Leftrightarrow f[n_1] \bigoplus f[n_2] = 0$$

Теорема 11.2.1.

$$f[nim(n1), nim(n2)] = n_1 \bigoplus n_2$$

▶ $f[\langle v_1, v_2 \rangle] = mex\{a_i \bigoplus f[v_2], f[v_1] \bigoplus b_i\}$
$a_i \neq f[v_i] \Rightarrow a_i \bigoplus f[v_2] f[v_1] \bigoplus f[v_2]$
Аналогично $b_i \neq f[v_2]$ ◀

Лемма 11.2.1.

$$f[nim(n1), nim(n2)] = n_1 \bigoplus n_2$$

▶ Переход $2^k \to 2^{k+1}$

	$A + 2^n$	A
2^n	A	$A + 2^n$
	2^n	

◀

Следствие из теоремы – ним на k кучках:

$$f[\langle x_1, \dots, x_k \rangle] = x_1 \bigoplus \dots \bigoplus x_n$$

11.2.5. Ретроанализ

Теперь симмитричная игра на ориентированном графе с циклами.

$w \overset{\exists}{\to} l$
$l \overset{\forall}{\to} w$

пока $\exists$ непомеченная, для которой можем сказать – ставим.

Лемма 11.2.2. Непомеченные – ничейные.

▶ Из непомеченных всегда есть цикл (если бы не было, была бы конечная вершина, для которой можно было бы определить состояние). ◀

Алгоритм быстрого ретроанализа:

1. Добавим все вершины, которые изначально проигрышные (их степень равна нулю. Здесь и дальше под степенью вершины подразумиваем кол-во исходящих из нее ребер).
2. Запустим `bfs`.
3. Достаем очередную вершину из очереди. Если она L, то помечаем все вершины, из которых есть переход в данную как W и добавляем их в очередь. Если же она была W, то удаляем из графа (даже не удаляем, а просто уменьшаем счетчик степень у вершин, которые имели ребро в эту) все ребра, что в нее входили. Если у какой-то вершины после этого стала степень 0, то помечаем ее как L и добавляем в очередь.
4. Те вершины, которые мы не смогли пометить, являются ничейными.

Время работы, как и алгоритма `bfs` составляет $O(V+E)$

11.2.6. BinaryTree

Slow: $O(TreeSize)$

Fast:

```
go(v) {
    if (rand() % 2) {
        swap(v.left, v.right);
    }
    if (go(v.left) == LOSE)
        return WIN;
    if (go(v.right) == LOSE)
        return WIN;
    return LOSE;
}
```

$W(h) = L(h-1) + \frac{1}{2} \cdot W(h-1)$

$L(h) = 2 \cdot W(h-1)$

$W(h) = \frac{1}{2} \cdot W(h-1) + 2 \cdot W(h-2)$

Это примерно равно $O(1.687^h)$ (вместо 2^h).

11.3. Битовое сжатие

11.3.1. Оптимизируем рюкзак и алгоритм Флойда

Вспомним практику:

- Рюкзак:

```
bitset<S + 1> f;
f[0] = 1;
forn(i, n)
  f |= f << a[i]; // O(S/word)
```

- Флойд:

```
forn(k, n) {
  forn(i, n) {
    if (c[i][k])
      c[i] |= c[k];
  }
}
```

Флойдом мы нашли транзитивное замыкание графа (т.е. матрицу достижимости). Однако эту задачу можно решить быстрее:

1. Сожмем компоненты сильной связности за $O(n+m)$.
2. Динамика на ацикличном графе: Пусть из вершины v достижимы компоненты $x_1, \dots, x_k$, тогда $f[v] = OR(f[x_i])$. Ассимптотика: $O(\frac{nm}{word})$.

11.3.2. Умножение, деление и Гаусс

Над $\mathbb{F}_2[x]$ мы умеем быстро умножать и делить.

- Умножение (обычная реализация):

```
forn(i, n) {
  forn(j, n)
    c[i + j] = a[i] & b[j];
}
```

Быстрая реализация:

```
forn(i, n) {
  if (a[i])
    c |= b << i;
}
```

- Быстрое деление:

```
for (i = deg(a); i >= deg(b); i--)
  if (a[i])
    a ^= (b << (i - deg(b)));
```

- Быстрый Гаусс: сложения и вычитания строк - это ксор. Меняем местами строки тоже быстро. Получился быстрый Гаусс.

11.4. Крестики-нолики

Играем на поле 30 на 30. Нужо собрать отрезок из пяти крестикиов (или ноликов). Поэтапно придумываем стратегии:

1. Жадно не будем давать собрать палку - рисовать крестик в конец.
2. Разобъем поле на объекты. Каждый объект имеет ценность. Будем минимизировать сумму ценностей имеющихся объектов.
3. Оставим k самых удачных ходов и уйдем в рекурсию с отсечением по времени.
4. Сделаем бинарный поиск по "правда ли, что через какое-то количество ходов суммарная стоимость будет меньше F". Получили 01 дерево. Используя технику рандома с предыдущей пары уйдем в рекурсию. В результате примерная глубина: 6.2

Бывает AB отсечение, это примерно последний пункт, только хуже.

11.5. Метод 4-х русских

11.5.1. Перемножение бинарных матриц

Мы хотим умножить две матрицы над $\mathbb{F}_2$.

Разрежем первую матрицу (A) на k вертикальных полосок, а вторую (B) на k горизонтальных. Заметим, что $AB = \sum a_i b_i$. Пока умножение все еще раотает за $O(\frac{n}{k}(nkn) = n^3)$. Рассмотрим поподробнее, что происходит, когда мы умножаем две матрицы $n * k$. Каждая строка l результата - это сумма каких-то строк из b_i, причем, очередная строка суммируется, если в матрице a_i в строке l в соответсвующей позиции стоит единичка и не учитывается, если там стоит ноль. Пусть $k = \log n$. Тогда за $O(n2^k) = O(n^2)$ мы можем предподсчитать все такие линейные комбинации строк из b_i. Теперь, сразу зная ответ для очередной строки, умножение матриц $n * k$ работает за $O(N^2)$. Таким образом итоговая асимптотика - $O(\frac{n^3}{\log n})$.

Если везде использовать битовое сжатие, то можно все сделать еще в $word$ раз быстрее.

11.5.2. Построение булевой схемы

У нас есть функция $f : \{0,1\}^n \to \{0,1\}$. Хотим для нее построить схему.

1. Приведя выражение в КНФ или ДНФ легко построить схему, размером $n2^n$.
2. Будем разделять функцию на две, пока не кончатся аргументы: $f_0 = f|_{x_n=0} \cup f|_{x_n=1}$

```
if (x[n] == 0)
  f_0(x[1], x[2], ... , x[n - 1]);
else
  f_1(x[1], x[2], ... , x[n - 1]);
```

   Если такую схему строить по принципу: $(f_0 \, and \, !\, x_n) \, or \, (f_1 \, and \, x_n)$, что для каждого аргумента дает константу дополнительных блоков, то ее размер будет $O(2^n)$.

3. Давайте предыдущим алгоритмом избавимся от всех аргументов, кроме первых k: $n \to n-1 \to n-2 \to ... \to k$, где $k = \log n - 1$. Всего различных функций из k аргументов 2^{2^k}. Заранее предподсчитаем все такие функции. Это потребует $2^{2^k} + 2^{2^{k-1}} + \cdots = 2^{n/2} + 2^{n/4} + \cdots = O(2^{\frac{n}{2}})$. Теперь будем просто использовать предподсчитаные функции. Размер итоговой схемы: $2^{n-k+1} = 2^{n-\log n+2} = \frac{4}{n}2^n$.

11.5.3. Наибольшая общая подстроку

Найдем наибольшую общую подстроку за $\Theta(\frac{n^2}{\log n^2})$.

1. Вспомнили старую динамику за квадрат.
2. Заметим, что $dp[i][j+1] - dp[i][j] \in \{0, 1\}$ и $dp[i+1][j] - dp[i][j] \in \{0, 1\}$.
3. Представим матрицу динамики в виде кучи квадратиков, со сторонами длиной k. Для того, чтоб посчитать динамику в каждом квадратике, нужно только знать значения динамики на его верхней и левой границе, причем, так как на этих границах значения неубывают и отличаются не больше, чем на единицу, их (границы) можно задать битовой маской. Теперь нам ничто не мешает позвать на помощь четырех русских, которые предподсчитают все такие квадратики.
4. Пусть $k = \frac{\log n}{2+\epsilon}$. Тогда алгоритм работает за $O((n^{\frac{4}{2+\epsilon}} + (\frac{n}{k})^2 \dots$. Тут у меня в конспекте дыра, ребят, выручайте.

Глава 12

Метод Гаусса, Гаусс по не простому модулю, Базисы

12.1. Метод Гаусса

12.1.1. Основное

У матрицы можно менять строки и столбцы местами, складывать строки и столбцы. При этом определитель понятно как меняется.
Чтобы привести матрицу к треугольному виду, будем по очереди занулять элементы i-ого столбца под главной диагональю, вычитая i-ую строку, домноженную на нужный коэффициент. Чтобы привести к трапециевидной, будем вычитать элементы над и под главной диагональю.
К треугольному виду приводим за $\sum_{k=1}^{n} k^2 = \frac{n^3}{3}$ умножений, к трапециевидному за $\sum_{k=1}^{n} nk = \frac{n^3}{2}$.

12.1.2. Определитель

Определитель треугольной матрицы = произведение элементов на диагонали.

12.1.3. Система

Если привести матрицу к трапециевидной, решением будет для $i \leqslant n$ $x_i = \frac{b_i - \sum_{j>i} x_j a_{ij}}{a_{ii}}$, если к диагональной, то $x_i = \frac{b_i - \sum_{j \geqslant n} x_j a_{ij}}{a_{ii}}$, x_j при $j > n$ – свободные переменные. Если хотим менять столбцы (а иначе у нас может получиться столбец нулей, портящий трапециевидность/диагональность), то надо поддерживать перестановку переменных.
У нас могут быть свободные переменные, все решения образуют линейное пространство, базис которого мы нашли.
На вещественных числах у Гаусса плохая точность на некоторых тестах, например матрица Гильберта: $a_{ij} = \frac{1}{i+j}$. Чтобы улучшить точность можно ставить максимальный по модулю элемент из правой нижней подматрицы на место того, на который собираемся делить. Еще можно использовать *long double* или *BigDecimal*.

12.2. Гаусс по не простому модулю

Все ранее написанное работает в поле (где можно делить). Теперь посмотрим, что произойдет в кольце по не простому модулю.
Вместо деления будем с помощью вычитания строки домноженной на коэффициент (просто домножать строку на константу нельзя) заменять строки, начинающиеся на a и b на эквивалентные, начинающиеся на $gcd(a, b)$ и 0. Это можно делать за $O(n^3 T(gcd))$, вычитая и домножая строки по

ходу алгоритма Евклида ($a\%b = a - b \cdot \lfloor \frac{a}{b} \rfloor$) или за $O(n^2(n+T(gcd)))$, найдя расширенным алгоритмом Евклида линейное представление gcd и нуля (новые строки получаются как суммы старых с найденными коэффициентами). Так можно найти определитель.
Привести матрицу к диагональному виду не получится (например на матрице $((1,1),(0,2))$), т.к. мы не можем домножать строку на число.
Заметим, что если применить этот метод к кольцу Z, то числа могут расти экспоненциально.

12.3. Базисы

12.3.1. Основное

Можно найти базис линейного пространства (подпространства R^n) Гауссом, приведя матрицу из векторов-строк пространства к трапециевидной. Первые $rankA$ строк полученной матрицы будут базисом ($dimU = rankA$).

Def 12.3.1. Базис ортогональный, если $\forall i \neq j < v_i v_j >= 0$, а $< v_i v_i > \neq 0$.

Def 12.3.2. Базис ортонормированный, если он ортогональный и $\forall i < v_i v_i >= 1$.

В ортонормированном базисе любой вектор z можно представить в виде суммы $\alpha_i v_i$, где $\alpha_i =< z, v_i >$.

12.3.2. Ортогонализация Грама-Шмидта

Хотим за $O(n^3)$ получить из произвольного базиса ортонормированный. Рассматриваем вектора по очереди, делаем каждый следующий ортогональным предыдущему и делим на длину (нормируем). $u_k = norm(v_k - \sum_{i=1}^{k-1} < u_i v_k > u_i)$

Дополнение базиса до полного: Берем случайный вектор. Скорее всего нам повезло и он линейно независимый с данными.

Проверим, так ли это. Если нет, то возьмем другой случайный вектор.

Проверка, лежит ли вектор в данном подпространстве:

1. Дописываем вектор к данной системе векторов и проверяем, что данный вектор линейно зависим с предыдущими. Например, запускаем Гаусса.

2. Сделаем базис ортогональный и разложим вектор в данном базисе. Если полностью разложился, значит принадлежит.

Расстояние от точки до подпространства:

1. Сделать базис ортогональным. Вычесть из точки проекции на все базисные вектора и останется нормаль к пространству.

2. Ax — точка в подпространстве. Скалярное произведение со всеми векторами A равно 0. $A^T(Ax - b) = 0 \Leftrightarrow x = (A^T A)^{-1} A^T b$.

   Такое же уравнение можно получить сказав, что $(Ax - b)^2 \to min$.

Глава 13

Вероятности и изоморфизм графов

13.1. Окончание Гаусса

```
bitset<N> a[N];
int k = 0;
int pos[N];

void AddVector(bitset<N> v) {
    for (int i = 0; i < k; i++) {
        if (v[pos[i]])
            v ^= a[i];
    }

    int i = 0;
    while (i < n && !v[i]) // Эти 3 строки -- поиск любого ненулего бита
        i++;

    if (i < n)
        a[k] = v, pos[k++] = i; // добавили вектор в базис
}
```

Массив `pos` отвечает за следующее, `pos[i]` – первый ненулевой бит (т.е. координата) у `i`-го вектора в нашем базисе (`a[N]`). `k` – их количество на данный момент. Т.е. добавляя новый вектор `v` в базис, мы смотрим на наши текущие вектора в базисе (т.е. на данный момент ещё может и не базисе, а просто системе линейно независимых векторов) и, если у нашего вектора есть координата, которая является первой для какого-то вектора, то надо тот вектор из нашего `v` вычесть. Т.о. мы приводим нашу матрицу к диагональному виду. Если после всех таких вычитаний у `v` осталась какая-то ненулевая координата, то после его добавления наша система попрежнему останется линейно независимой, так что добавляем этот вектор.

Восстановление ответа (т.е. решение системы уравнений – разложение вектора b в нашем базисе):

```
x = 0; // x -- bitset
for (int i = k - 1; i >= 0; i--) {
    int j = pos[i];
    x[j] = a[i][N - 1]; // Если записывать A * x = b, то x[j] = b[j]
    for (int t = i + 1; t < k; t++) {
        x[j] ^= x[pos[t]] & a[t][pos[t]];
```

```
7       }
8   }
```

13.2. Вероятности

13.2.1. Марковский процесс

Пусть есть какой-то ориентированный граф с выделенной стартовой вершиной. На каждом ребре написана вероятность перехода по нему, если мы стоим в вершине, которая является его началом. При этом, конечно, выполнено $\forall v \in V \ \sum p(v \to x) = 1$.
Наша задача состоит в том, чтобы определить, где мы с какой вероятностью окажемся через k шагов.

1. Динамическим программирование за $O(k \cdot E)$

   $$f[k+1, x] = \sum_{v \to x} f[k, v] \cdot p_e$$

   где p_e – вероятность перехода по ребру e из вершины v, а f – собственно сами вероятности. Изначально f имеет вероятность 1 на позиции s (s – стартовая вершина) и 0 на остальных.

2. k большое $O(\log k \cdot V^3)$
   Заметим, что

   $$f_{k+1} = P \cdot f_k$$

   где P – полная матрица вероятностей переходов для всех ребер.
   Тогда мы получаем, что

   $$f_k = P^k \cdot f_0$$

   Теперь осталось только возвести P в степень k быстрым возведением в степень за $O(\log k \cdot V^3)$ (V^3, т.к. размер матрицы P, как и размер просто матрицы смежности, – это V^2).

3. $k = \infty$

Теорема 13.2.1. Утверждение без доказательства — у такого процесса существует предел.

▶ Без доказательства ◀

Существует предел – в том смысле, что на бесконечности он более-менее зацикливается (более-менее, т.к. процесс все же случайный и некоторые флуктуации происходить все же могут).
Заметим, что мы уже умеем приближенно это решать способом "скажем, что $k = 2^{magic}$ и решим задачу вторым способом за $magic \cdot V^3$". Это, конечно, хорошо, но можно решать быстре: По сути, нам надо найти такой вектор f, что $P \cdot f = f$. Ну заметим, тогда, что f – собственный вектор P для $\lambda = 1$. Так же для f должно выполняться, что $f_0 + f_1 + ... f_{n-1} = 1$. Задачу поиска собственного вектора для определенного собсвенного числа можно решать Гауссом за V^3, решая систему $(A - \lambda E) \cdot x = 0$. При этом стоит заметить, что у метода решения Гауссом есть и свой минус – потеря точности.

13.2.2. Решение СЛАУ – метод итераций

Пусть есть СЛАУ вида

$$A \cdot x = b$$

Тогда заметим, что мы можем перепиать это вот так:

$$(A - E + E) \cdot x = b$$

$$(A + E) \cdot x - b = x$$

Тогда будем делать так, возьмем $x_0 = random\_vector$, а дальше делаем

$$x \to (A + E) \cdot x - b$$

и так пока не сойдется.

Задача – научиться быстро считать $x \to Ax + b$ и так k раз

$$\begin{pmatrix} a_{11} & \dots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \dots & a_{nn} & b_n \\ 0 & \dots & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = Ax + b$$

Т.е. просто возводим эту матрицу, дополненную столбцом b и строкой $00 \dots 01$, в степень k и умножаем на x.

13.2.3. Алгоритм Видемана

$$A \cdot x = b,\ b \neq 0$$

За $O(n \cdot k)$, где k – количество ненулевых чисел в A.
Если он вам когда-нибудь понадобится, он описан на 287 странице вот этой книжки:
http://www.ict.edu.ru/ft/002416/book.pdf
Кстати, стоит заметить, в этой книге так же описаны и некоторые другие темы из алгебры (например, БПФ).

13.3. Проверка на изоморфность

13.3.1. Проверка двух DFA на равенство

DFA – конечный детерменированный автомат

Нужно проверить 2 автомата на равенство. Т.е. что они равны с точностью до нумерации вершин и ребер.

Алгоритм:
1. Встали в начальные вершины.
2. Запустили синхронный dfs.
Если мы нигде не сломались по пути и у каждой вершины в итоге есть пара, то автоматы равны.

13.3.2. Проверка двух DFA на эквивалентность

Задача:
Проверить эквивалентность двух автоматов. Т.е. что они принимают один и тот же набор строк. (A_1 принимает $s \Leftrightarrow A_2$ принимает s)

Эту задачу будем решать в 2 шага:
1. Построить минимальный автомат, эквивалентный A_1; минимальный, эквивалентный A_2
2. Сравним их на равентсво

Теорема 13.3.1. $A_1 \sim A_2 \Rightarrow MinA_1 = MinA_2$

▶ Без доказательства ◀

13.3.3. Алгоритм Хопкрофта

Построим минимальный автомат, эквивалентный данному:
Заметим, что если мы объединим две эквивалентные вершины в одну(эквивалентными называются те, вершины, начиная из которых мы принимаем одно и то же множество слов), то новый автомат будет эквивалентен изначальному, т.к. он будет распознавать все то же множество слов. Будем разбивать множество на классы эквивалентности. Когда мы все разобьем на классы, эти классы и будут состоянимями нашего нового, уже минимального, автомата. Алгоритм будет следующим:

1. Построим из матрицы переходов $\delta(v,a), v \in A, a \in \sum$ обратную матрицу $\delta^{-1}(v,a) = \{t|\delta(t,a) = v\}$. За $\delta^{-1}(F,a)$ обозначим полный прообраз F при переходе по ребрам с буквой a (т.е. $\{t|\delta(t,a) \in F\}$)

2. Начальное разбиение множества $B = \{T, V \setminus T\}$, где T – множество терминальных вершин, а V – множество всех вершин автомата.

3. Будем поддерживать множество L. Изначально в него положим все пары $\langle F, a\rangle$ и $\langle V \setminus F, a\rangle, \forall a \in \sum$

4. Пока множество L непусто, извлекаем очередной его элемент $\langle C, a\rangle$ и меняем множество B следующим образом: берем каждое множество R из B и разбиваем его на 2 – первое будет множеством R_1, т.ч. $\delta(R_1, a) \in C$, и множество R_2, т.ч. $\delta(R_2, a) \cap C = \emptyset$. Если они оба не пустые, то в множество B, вместо R, войдут R_1 и R_2 (R из B мы удаляем), а в L нужно будет добавить меньшее из множеств R_1 и R_2 (т.е. пару $\langle R_1, a\rangle$, если $R_1 < R_2$). Если же кто-то из множеств R_1 и R_2 оказался пустым, то просто оставляем R в B, ничего не добавляя в L.

5. На выходе получаем множество B – множество классов эквивалентностей вершин.

Теорема 13.3.2. Можно показать корректность алгоритма:

▶ Пусть есть вершины u и v, которые находятся в одном классе в B, но при этом они не эквивалентны. Пусть тогда для какой-то $a \in \sum$ их образы неэквивалентны и $\delta(u,a)$ и $\delta(v,a)$ лежат в разных классах (если они лежат в одном классе, то переидем к рассмотрению $\delta(u,a)$ и $\delta(v,a)$, вместо u и v). Рассмотрим тот первый момент времени, когда какой-то единый класс, содержащий $\delta(u,a)$ и $\delta(v,a)$ был разделен на два и эти вершины, $\delta(u,a)$ и $\delta(v,a)$, оказались в разных классах. В этот момент одно из множеств R_1 и R_2 (содержащие $\delta(u,a)$ и $\delta(v,a)$ соответсвенно) было добавленно в L. Но это означает, что потом когда это множество будет извлекатся из L, то множество, которое якобы содержит u и v должно было разделится на

2, первое из которых не содержит вершины v, а второе не содержит вершины u. Пришли к противоречию. ◀

Видно так же, что алгоритм остановится, т.к. когда мы разбиваем множество R на R_1 и R_2, то $|R_1| < |R|$ и $|R_2| < |R|$. Понятно, что уж как минимум, когда мы дойдем до того, что в B лежат только множества мощности 1, мы уже ничего не будем делить в B и ничего не будем, соответственно, добавлять в L.

Этот алгоритм алгоритм работает за $O(n \log n)$. Доказательство, а заодно и сам алгоритм, вы можете прочитать в викикоспектах или же в оригинальной статье:
http://i.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf

Мы докажем только то, что он работает за $O(|\sum| \cdot n^2)$

▶ Заметим, что в каждый момент времени, каждая вершина автомата находится только в одном множестве из L. Теперь посмотрим на очередное множество C, которое мы достали из L. После того, как мы разделим какой-то R на R_1 и R_2 (пусть $|R_1| \leqslant |R_2|$) и R_1 в L. Понятно, что $|R_1| < |R|$. Значит, каждая вершина автомата окажется в каком-то множестве в L не более n раз. Значит, каждое ребро будет рассмотрено $O(n)$ раз. Всего ребер $O(|\sum| \cdot n)$, а значит и время работы алгоритма составит $O(|\sum| \cdot n^2)$. ◀

Все, построив алгоритмом Хопкрофта минимальный автомат, эквивалентный первому, $MinA_1$ и так же для второго $MinA_2$, мы проверяем, что $MinA_1 = MinA_2$

13.3.4. Изоморфизм графов

Проверить, что G_1 и G_2 совпадают с точностью до нумерации вершин.
Эта задача в общем случае имеет субэкспоненциальное решение за $O(2^{\sqrt{n \log n}})$, о чем можно прочитать прямо вот тут:
http://en.wikipedia.org/wiki/Graph_isomorphism_problem

Но почти всегда за $O(n^3 \log n)$ работает техника измельчения (это то, что мы только что наблюдали в алгоритме Хопкрофта).

1. $type_0[v] = 1$, для каждого $v \in V$

2. $map < SortedType > \Rightarrow newType$

В конце для каждого типа проверяем, что
$count_1[type] \neq count_2[type] \Rightarrow G_1 \neq G_2$.

Глава 14

Длинная арифметика. Быстрое преобразование Фурье

14.1. Быстрое преобразование Фурье

14.1.1. Цель

Цель: перемножить два многочлена.
$P(x) = \sum_{i=0}^{N-1} a_i x^i$, $\deg P = N$.
$Q(x) = \sum_{i=0}^{M-1} b_i x^i$, $\deg Q = M$.
Наивный алгоритм: $O(NM)$. Хотим быстрее.

Интерполяция: восстановление многочлена по N различным по x точкам. Наивный алгоритм: интерполяция Лагранжа за $O(N^2)$. Построим многочлен, который для всех x_j, кроме x_i $P(x_j) = 0$, $P(x_i) = 1$. $Omega_i(x) = \frac{\prod_{i \neq j} x - x_j}{\prod_{i \neq j} x_i - x_j}$. $P(x) = \sum_{i=1}^{N} Omega_i(x)$.

Экстраполяция: посчитать значение многочлена в N точках. Наивный алгоритм: $O(N^2)$ — схема Горнера: $P(x) = a_0 + x(a_1 + x(a_2 + x(\cdots + x(a_N) \ldots)))$.

Итак, есть биекция между многочленами степени меньше N и наборами из N пар (x_i, y_i).

14.1.2. Описание алгоритма

Будем считать произведение многочленов P и Q следующим образом:

1. Экстраполируем P, Q. Наивный алгоритм: $O((N+M)^2)$, хотим улучшить.
2. Перемножим поточечно. Наивный алгоритм: $O(N+M)$.
3. Интерполируем результат. Наивный алгоритм: $O((N+M)^2)$, хотим улучшить.

Далее будем считать, что $N, M < n = 2^k$ — добьём старшие коэффициенты нулями до одной и той же степени двойки.

Мы вольны выбирать x_i. Выберем x_i как все комплексные корни n-ой степени из 1. Заметим, что каждый второй корень степени 2^k является корнем степени 2^{k-1}. Воспользуемся этим и применим метод «разделяй и властвуй».

Разделим индексы на чётные и нечётные. $A(x) = (a_0 + a_2 x^2 + \ldots) + x(a_1 + a_3 x^2 + \ldots) = B(x^2) + xC(x^2)$, где B и C — многочлены степени 2^{k-1}.

Пусть w_n — первый из корней n-ой степени из 1. Хотим посчитать $A(w_n^k)$ для всех $k = 0 \ldots n-1$. $w_n^k = e^{k\frac{2\pi i}{n}}$. $w_n^{2k} = e^{2k\frac{2\pi i}{n}} = e^{k\frac{2\pi i}{n/2}} = w_{n/2}^k$ — то есть при спуске осуществляется переход от k к $k-1$ — и в смысле степени ($2^k \to 2^{k-1}$), и в смысле корней (от корней степени $n = 2^k$ перешли к корням степени $n/2 = 2^{k-1}$).

$k < n/2$. Пусть $b_k = B(w_{n/2}^k)$, $c_k = C(w_{n/2}^k)$ — они посчитаны рекурсивно. Тогда $A(w_n^k) = b_k + w_n^k c_k$.

$n > k \geqslant n/2$. $b_k = B(w_{n/2}^k) = B(w_{n/2}^{n/2} w_{n/2}^{k-n/2}) = B(w_{n/2}^{k-n/2})$, $c_k = C(w_{n/2}^{k-n/2})$.

Такое преобразование называется преобразованием бабочки. Всё вместе — это и есть прямое преобразование Фурье.

Теперь обратно.

$Y = M \times A$, $M_{ij} = x_k^j = e^{kj\frac{2\pi i}{n}}$. Выпишем матрицу, это матрица Вандермонда. Она невырождена, поэтому существует обратная M^{-1}.

Оказывается, $M_{kj}^{-1} = \frac{1}{n} e^{-kj\frac{2\pi i}{n}}$. Проверяем: $MM_{kk}^{-1} = \sum l = 1^n \frac{1}{n} e^{kj\frac{2\pi i}{n} - kj\frac{2\pi i}{n}} = \frac{n}{n} = 1$. $MM_{kj}^{-1} = 0$ — предлагается проверить самим.

Таким образом, чтобы сделать обратное преобразование Фурье, нужно подставить комплексно сопряжённые корни в прямое преобразование Фурье, и умножить в конце на $\frac{1}{n}$.

Заметим, что $\bar{x}_i = x_{n-i}$. Поэтому можно просто перевенуть список корней (кроме $x_0 = 1$), и на нём запустить прямое преобразование.

14.1.3. Реализация

$O(n \log n)$ получили, но хотим ускорить.

Будем делать `FFT` на месте — получаем массив коэффициентов `complex* A`, изменяем его так, что в ячейках будут значения многочлена.

Тип `complex` можно использовать из `std`, но он медленнее, чем рукописный.

Сейчас $O(n)$ дополнительной памяти — один дополнительный массив, который используем, чтобы разделить на чётные и нечётные.

После этого вызываем рекурсию, затем нужно сделать преобразование бабочки. Пусть $x = a_k$, $y = a_{k+n/2}\tilde{w}$, тогда $a_k = x + y$, $a_{k+n/2} = x - y$. $\tilde{w}$ каждый раз умножается на w.

Можно поддерживать указатели `*x, *y`, чтобы не заводить лишних переменных. Тогда на каждой итерации:

```
1 *y *= w, x += y, y = x - 2 * y;
```

Корни $\tilde{w}$ можно предподсчитать.

Это была рекурсивная схема. Избавимся от рекурсии.

Проследим за тем, как распределяются числа по массиву: в каком порядке они будут на самом нижнем уровне рекурсии? Это можно понять по битовой записи. На k-ом шаге попадёт число в левую или правую половину в зависимости от k-ого с конца бита. Таким образом, перестановка — это отсортированные по *перевёрнутой битовой записи* числа. Заметим, что числа, получающиеся при переворачивании битов чисел $0 \dots n-1$ — это числа $0 \dots n-1$ в некотором порядке. Тогда сортировка — это просто расстановка чисел в нужные ячейки: $A[rev[i]]$ (массив `rev` умеем считать за $O(n)$).

Избавились от рекурсии.

Дополнительная память тоже не нужна: заметим, что наша перестановка просто меняет местами i и $rev[i]$. Тогда за один проход мы сможем применить перестановку:

```
1 for(int i = 0; i < N; ++i)
2     if(i < rev[i])
3         swap(A[i], A[rev[i]]);
```

Смешанное преобразование Фурье: если размер блока на уровне небольшой (например, $< 2^{11}$), то сначала перебираем степень w, потом перебираем номер блока (так можно делать, потому что небольшие скачки по памяти); если размер блока на уровне большой, то для каждого блока перебираем степень w.

14.1.4. Перемножение длинных чисел

Перемножение длинных чисел — это перемножение многочленов, где основание системы счисления — $x = B$. В конце — после перемножения — округляем до ближайшего целого, нужно ещё сделать нормализацию числа, то есть выполнить переносы через разряды. Оценим максимальное число, которое получится в коэффициенте: сумма n слагаемых, каждое из которых $< B^2$.

Пример: `long long`, $n = 10^6 \Longrightarrow B = 10^6$ — основание системы счисления.

14.1.5. FFT в целых числах

Хотим избавиться от погрешности и от вещественных чисел вообще. Будем работать не в $\mathbb{C}$, а в $\mathbb{Z}_p$, где $p \in \mathbb{P}$ и $p = 2^k \cdot c + 1$. $2^k \geqslant n$. Например, $p = 3 \cdot 2^{18} + 1$.

Существует первообразный корень g по модулю p такой, что $\{g^0, g^1, \dots, g^{p-2}\} = \{1, 2, \dots, p-1\}$. g можно искать наивно: утверждается, что оно небольшое.

Положим теперь $w_{2^k} = g^c$. Оно удовлетворяет всем необходимым свойствам. При этом $\frac{1}{n} = n^{p-2}$.

14.2. Длинная арифметика и использование Фурье

3 способа хранения многочленов:

1. vector<Complex> Степень $\leftrightarrow$ размер вектора

2. ```
   struct{
       Complex[N];
   };
   ```

   «-»: долго выполняются операции с маленькими многочленами(считается, что степени всех многочленов N).

3. ```
   struct{
       int n;
       Complex[N];
   };
   ```

   Быстрее, чем первый, т.к. не динамический. В n храним степень многочлена, поэтому операции с маленькими многочленами выполняются за нормальное время.

Целые числа: Добавляется знак. Добавляется операция сравнения(для определения знака). Храним от младшего разряда к старшим.

```
struct{
    int n;
    int[N];
    sign;
};
```

Вещественные числа: Храним, как целые + степень. Поэтому добавляется ещё одно поле int pow. $123.765 \to 123765 \cdot B^{-3}$, где B - база системы счисления. Добавляем операцию выравнивания(по точке).

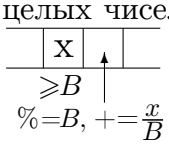
Операции:

- Перемножение.

1. *Целые*

   Наивная реализация перемножения двух многочленов:

```
for (int i = 0..n-1)
    for (int j = 0..n - 1)
        r[i + j] += a[i]*b[i];
```

   Для целых чисел нужно ещё обрабатывать переносы в следующий разряд. Нормализация: 

   Обыкновенный перенос в следующий разряд: в текущей ячейке число берём по иодулю, а целую часть от деления прибавляем к следующему разряду. Если nB^2 не влезает в long long, то нормальзацию надо делать на каждой итерации:

```
for (int i = 0..n - 1)
    for (int j = 0..n - 1)
        if ((r[i + j] += a[i]*b[j]) >= B) { Normalization }
```

   Иначе, делаем нормальзацию один раз в конце. Можно делать что-то среднее:

```
if (r[i + j] >= B*B)
    r[i+j] -= B^2;
    r[i + j + 1]++;
```

   Инвариант: в каждой ячейке число не больше B^2. Так как каждый раз прибавляем не больше чем B^2, то чтобы сохранить инвариант один раз вычесть будет достаточно. Такой метод нормализации ускорит программу, поскольку, во-первых, мы её реже делаем, а во-вторых, не тратим лишнюю операцию взятия по модулю.

   Для Фурье обязательно nB^2 должно влезать в long long.

- gcd

1. *Целые*

   Что мы знаем? Мы знаем только алгоритм Евклида. $\gcd(F_n, F_{n-1}) = \gcd(F_{n-1}, F_{n-2}) = ... = 1$ - худший случай. $F_n \approx \varphi^n \Rightarrow$ он работает за $\mathcal{O}(\log n)$. Если взятие по модулю работает за $\mathcal{O}(1)$. Эту операцию с длинными числами мы пока делать не умеем. И даже когда научимся, то это будет долго. Поэтому используем бинарный алгоритм Евклида:

   A, B. Если оба делятся на 2 $\Rightarrow$ делим, gcd *= 2. Если ровно один делится на 2 - делим. Если ни один не делится на 2, то вычитаем из одного другой и затем выполняем вторую операцию(то есть у нас будет 1 чётное и 1 нечётное, чётное делим пополам).

   Таким образом, каждая итерация работает за $\mathcal{O}(1)$ и уменьшает одно из чисел хотя бы вдвое. Значит, работает за $\mathcal{O}(\log n)$.

   Чтобы реализовать этот алгоритм надо научится умножать на константу и делить на константу. Мы будем делать это за $\mathcal{O}(\log N)$. Таким образом, алгоритм Евклида будет работать за $\mathcal{O}(n^2)$, где n - количество цифр в числе.

2. *Многочлены*

   Обычный(небинарный) алгоритм Евклида. Нет никаких переносов. Надо научится брать многочлен по модулю $\leftrightarrow$ деление многочленов. Будет дальше.

- Умножение, деление на константу

1. *Многочлены*

   Очевидно.

2. *Целые*

   Умножение: обычное умножение + нормализация.

   $\boxed{a|b|\ \ \ }$ | c
   $\underbrace{a - \lfloor \frac{a}{c} \rfloor c}_{B(\dots)} + b$ | $\lfloor \frac{a}{c} \rfloor$ ...

   Деление($c < B$): ... - обычное деление в столбик.

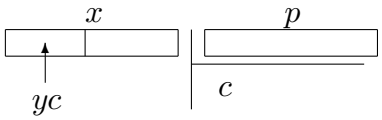
   Если $c > B$, то надо будет просто группировать друг с другом больше блоков. Таким образом, работает за $\mathcal{O}(\log N)$.

- Деление

  1. *Многочлены*

     $n - m$ раз подбираем коэффициент для старшего разряда, вычитаем за $\mathcal{O}(m)$. Итого: $\mathcal{O}((n-m)m)$.

  2. *Целые числа*

     c - подбираем бинпоиском.

     $(n-m)\log Bm$
     ↑ ↑ ↑
     #раз BS вычитание

     Попробуем очередное c вычислить за $\mathcal{O}(1)$. Рассмотрим, $C = \lfloor \frac{P}{Y} \rfloor$ - по первых несколько цифр чисел Y и P. Утверждается, что достаточно 2-х первых цифр, чтобы приближение C к c было достаточно точным. Оценим $\frac{P}{Y}$

$$p = \overline{a_1 a_2 \dots}, y = \overline{b_1 b_2 \dots}$$
$$\overline{a_1 a_2 0000 \dots 00} \leqslant P \leqslant \overline{a_1 a_2 9999 \dots 99}$$
$$\overline{b_1 b_2 0000 \dots 00} \leqslant Y \leqslant \overline{b_1 b_2 9999 \dots 99}$$
$$\frac{a_1 B + a_2 + 1}{b_1 B + b_2} < \frac{P}{Y} < \frac{a_1 B + a_2}{b_1 B + b_2 + 1}$$
$$\left| \frac{a_1 B + a_2}{b_1 B + b_2 + 1} - \frac{a_1 B + a_2 + 1}{b_1 B + b_2} \right| = \frac{1 + b_1 B + b_2 + a_1 B + a_2}{(b_1 B + b_2)(b_1 B + b_2 + 1)} \leqslant 5$$

так как $1 + b_1 B + b_2 + a_1 B + a_2 \leqslant 5B^2, (b_1 B + b_2)(b_1 B + b_2 + 1) \geqslant B^2$

$\Rightarrow c = C \pm 5$ Таким образом, чтобы подобрать c берём $\tilde{c} = \frac{a_1 B + a_2}{b_1 B + b_2 + 1}$ и перебираем 2-3 числа вперёд, пока не найдём нужное.

Всё это описано в Кнуте.

Проблема:

Неизвестно, что делать, если при прибавлении 1 появится новый разряд.

На текущий момент мы умеем делать следующее:

	$+, -, <$	$*$	gcd	$*c, /c$	$\%, /$
Многочлены	$\mathcal{O}(n)$	$n \log n$	n^3	$\mathcal{O}(n)$	n^2
Целые(вещественные)	$\mathcal{O}(n)$	$n \log n$	n^2	$\mathcal{O}(n)$	$n^2 \log B$, n^2

- Корень(для целых, вещественных)

Первый вариант сделать втупую: деление и корень не сильно отличаются. Бинпоиском подбираем корень(за $\mathcal{O}(\log N)$), а при проверке перемножаем Фурье. Таким образом, асимтотика $\mathcal{O}(nn \log n)$

Но мы хотим быстрее: метод Ньютона. Он находит 0 функции $f(x)$.

Пусть $\bar{x}$ - приближённое решение. (для коря стоит брать число $1000\ldots00$, где $n/2$ нулей)

$\bar{\bar{x}} = \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})}$ - уточнение решения.

$f(x) = A - x^2$

$\bar{\bar{x}} = \bar{x} - \frac{A-\bar{x}^2}{-2\bar{x}} = \bar{x} + \frac{A-\bar{x}^2}{2\bar{x}} = \frac{\bar{x}^2+A}{2\bar{x}} = \frac{1}{2}(\bar{x} + \frac{A}{\bar{x}})$ Складываем и делим на константу мы быстро. Значит время работы зависит от времени деления.

Для хроших функций метод Ньютона удваивает точность(количество правильных цифр). Наша функция хорошая $\Rightarrow$ нужно $\mathcal{O}(\log n)$ итераций. Значит корень найдём за $\mathcal{O}(DIV \cdot \log n)$.

Точность:

$$\text{Пусть } \sqrt{A} = z \Rightarrow \frac{A}{z} = z$$

$$z + \varepsilon \to 1/2(z + \varepsilon + \frac{A}{z+\varepsilon})$$

$$\frac{A}{z+\varepsilon} = \frac{A}{z} - \varepsilon\frac{A}{z(z+\varepsilon)}$$

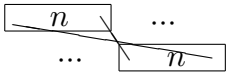
$$\varepsilon\frac{A}{z(z+\varepsilon)} = \varepsilon\frac{1}{1+\varepsilon/z} \approx \varepsilon(1-\varepsilon/z)$$

$$\text{Итого: } 1/2(z + \varepsilon + \frac{A}{z+\varepsilon}) = z - 1/2\frac{\varepsilon^2}{z}$$

$$\varepsilon \to \varepsilon^2$$

Хотим убыстрить деление!

$\frac{A}{B} = AB^{-1}$ Но в B^{-1} бесконечное количество знаков. Нам достаточно $\geqslant n$ где n - количество знаков A. Тогда погрешность будет $\leqslant 10^{-n}10^n = \pm 1$



- Нахождение обратного.

1. *Целые.* И снова метод Ньютона!

$$f(x) = \frac{1}{x} - B$$

$$\bar{\bar{x}} = \bar{x} - \frac{\frac{1}{\bar{x}} - B}{-\frac{1}{\bar{x}^2}} = \bar{x} + \bar{x} - B\bar{x}^2$$

Точность:

$$\text{Пусть } z = B^{-1} \Rightarrow B = 1/z$$

$$z + \varepsilon \to 2z + 2\varepsilon - B(z+\varepsilon)^2$$

$$2z + 2\varepsilon - z - 2\varepsilon + \frac{\varepsilon^2}{z} = z + \frac{\varepsilon^2}{z}$$

$$\varepsilon \to \varepsilon^2$$

Теперь уточнение делаем через умножение, что мы умеем достаточно быстро. Итераций хватает $\log n$, так как за каждую итерацию точность увеличивается в два раза. $\Rightarrow$ получаем асимптотику $\log n \cdot n \log n = n \log^2 n$. Это приблизительная оценка. На самом деле алгоритм работает быстрее:

Давайте перемножать не весь многочлен, а только важные нам знаки, в зависимости от текущей точности. В изначальном приближении - 1 знак. Его мы как-нибудь сможем найти. Далее 2 знака, 4 знака, ... , n знаков.

Логарифм каждого числа округлим грубо до $\log n$. Тогда:

$\mathcal{O}(\log n(1 + 2 + \cdots + 2^k)) = \mathcal{O}((2^{k+1} - 1) \log n) = \mathcal{O}(n \log n)$, где $2^k = n$.

Кстати, точно так же можно улучшить время извлечения корня. То есть сначала с точностью до 1 знака, потом до 2 и так далее.

Тогда $\log n$ внешних итераций. Для каждой нужно посчитать в соответствующей точности $\bar{x}^{-1}$. То есть $\log n(n \log n) = n \log^2 n$.

Табличка теперь выглядит так:

	$+, -, <$	$*$	gcd	$*c, /c$	$\%, /$	$\sqrt{}$
Многочлены	$\mathcal{O}(n)$	$n \log n$	n^3	$\mathcal{O}(n)$	n^2	$\times$
Целые(вещественные)	$\mathcal{O}(n)$	$n \log n$	n^2	$\mathcal{O}(n)$	$n^2 \log B$, n^2, $n \log n$	$n \log^2 n$

2. *Многочлены*

$$\frac{1}{Q(x)} = 1$$

Например $(1 + x + \cdots + x^n)(1 - x) = 1 - x^{n+1} \xrightarrow[n \to \inf]{} 1$

$$\frac{1}{1 - x} = 1 + x + ...$$

Теорема 14.2.1. Обратный к многочлену существует тогда и только тогда, когда его младший член не 0.

Обратный к многочлену, разумеется, бесконечен, но наша цель - вычислить несколько первых членов.

$$(a_0 + a_1 x + \cdots + a_n x^n)(b_0 + b_1 x + ...) = 1$$

$$\begin{cases} a_0 b_0 = 1 \\ a_0 b_1 + a_1 b_0 = 0 \quad \text{- это решается за } \mathcal{O}(n^2) \text{ динамикой.} \\ ... \end{cases}$$

Назовём calc(a, $k/2$) - многочлен, составленный из первых k коэффициентов $1/a$. Тогда мы снова можем применить метод Ньютона!

$\bar{x} = \text{calc}(a, k/2)$. Тогда $\bar{\bar{x}} = 2\bar{x} - B\bar{x}^2$. $B = a$, из него можно взять первые k коэффициентов, так как остальные не влияют.

Остальное аналогично числам. И, соответственно, асимптотика тоже такая же.

- Теперь можно ускорить деление многочленов

  Если младший член Q не ноль, то $\frac{P}{Q} = P \cdot \frac{1}{Q}$, что мы можем вычислить за $\mathcal{O}(n \log n)$.

  Но что делать если Q необратимо? Индексы означают степень.

$$P_n = Q_m \cdot R_{n-m} + C_{m-1}$$
$$P(1/x) = Q(1/x)R(1/x) + C(1/x)| \cdot x^n$$
$$x^n P(1/x) = x^n Q(1/x)R(1/x) + x^n C(1/x)$$
$$x^n P(1/x) \Leftrightarrow \text{Разворачивание коэффициентов}$$
$$P^{rev}(x) = Q^{rev}(x)R^{rev}(x) + C^{rev}x^{n-(m-1)}$$

$Q^{rev}(x)$ - с ненулевым младшим коэффициентом. Почти победа! Обратим.

$$\frac{1}{Q^{rev}}(x)P^{rev}(x) = R^{rev}(x) + C^{rev}(x)x^{n-m+1} \cdot \frac{1}{Q^{rev}}$$

R^{rev} - степени $n-m$ и его мы хотим найти, то есть найти первые $n-m+1$ его коэффициент. Заметим, что у последнего слагаемого за счёт домножения на x^{n-m+1} первын $n-m$ коэффициент равен 0. Значит, его можно вычеркнуть. Далее алгоритм прост. Считаем произведение слева, берём первые $n-m+1$ коэффициент и разворачиваем R обратно.

В табличку добавится деление многочленов за $n \log n$.

Глава 15

Теория чисел

15.1. Системы счисления

Хотим быстро переводить числа. Будем делать это методом "разделяй и влавствуй".
Пусть в двоичной системе счисления $2n$ цифр.

```
Go(A, 2n)
    calc(2^n)            // n \log n
    A_1 = div(A, 2^n)        // n \log n
    A_0 = A - A_1 * 2^n      // n \log n
    x_0 = Go(A_0, n)
    x_1 = Go(A_1, n)
    return x_1x_0
```

Время работы: $O(n \log^2 n)$

15.2. Факторизация

Хотим разложить число на простые множители.
Лучшее время, за которое умеют сейчас: $O(const^{\sqrt[3]{k}})$, где k — длина числа $(\log n)$.

15.2.1. Алгоритм, основанный на эвристике Полларда $O(e^{\frac{1}{4}k})$

Рассмотрим две последовательности: x, y:

1. $n = pq, p <= q$
2. $x_0 = rand()$
3. $x_{i+1} = f(x_i)$ — от 0 до $n-1$, значит зациклится.
4. $y_i = x_i \bmod p$
5. Парадокс дней рождений: период $x = \Theta(\sqrt{n})$
6. T_p — период $y = \Theta(\sqrt{p}) = \Theta(n^{\frac{1}{4}})$
7. S_p — предпериод y

```
seed
rand()
    seed *= 239
    return seed
f(x)
    return (x^2 + 3) % n
```

Тогда:

1. $diff := x_{S_p+T_p} - xS_p$
2. $diff \vdots p, \not\vdots n$
3. $gcd(diff, n) \neq 1, n$

```
if (not Prime(n))
while (1)
    x_0 = rand()
    i = 0, j = 1
    while gcd(x_j - x_i, n) == 1    // если n не простое, то конечен
        i++, j +=2          // ищем период
    if gcd(x_j - x_i, n) != 0 /*n*/     // \log n
        break;
```

15.3. Проверка на простоту

15.3.1. Тест Ферма

Лемма 15.3.1. n — простое Ra $\forall a = 1..n-1 : a^{n-1} \equiv 1(n)$

```
test(n)
    a = rand [2...n-1]
    return a^{n-1} % n == 1
```

Возвращает составное Ra точно составное, иначе может ошибиться.
Числа Кармайкла:
$\forall a = 1..n-1 : a^{n-1} \equiv 1(n)$, *но непростое*
То есть в обратную сторону теорема Ферма не работает.
Если число не является числом Кармайкла, то с вероятностью $\frac{1}{2}$ алгоритм корректно определит простоту. Но все числа Кармайкла состоят больше, чем из двух простых множителей. Значит, запустим проверку на это перед тестом Ферма. (пользуемся тем, что если число является числом Кармайкоа, то состоит хотя бы из трез множителей, значит, один из них точно меньше кубического корня n).

```
for (i = 2; i^3 <= n; ++i)
    if (n % i == 0)
        not prime
```

Теперь ускорим этот алгоритм в $\log n$ раз: будем делить только на простые числа, их предподсчитаем заранее.

15.3.2. Тест Миллер-Рабина

$$Time = O(\log n)$$

Определяет простоту с вероятностью $\geqslant \frac{1}{4}$.

Если p — простое, то у уравнения $x^2 \equiv a(p)$ не больше двух решений. (одно, если $a = 0$)

$$Q(x) = x^2 - a = (x - x_0)(x + x_0); x \neq \pm x_0 RaQ(x) \neq 0$$

Т.о., если $x^2 \equiv 1(p)$ и p — простое, то $x = \pm 1 \Rightarrow$ если найдем другой корень, то число не простое:

```
if (abs(y) != 1 && y*y % p == 1)
    p не простое
```

Рассмотрим $a^{p-1}, p - 1 = 2^s t, t$— *нечетное*

$$a^t -> a^{2t} -> a^{4t} -> ... -> a^{2^s t} = a^{p-1} = 1$$

Берем случайное a и врзвожим в степень $p - 1$

Берем первую позицию, где получилась единица, значит, предыдущая не единица, значит, если и не -1, то не простое.

Пояснение: если в конце не 1, то алгоритм и так бы сказал, что не простое, а мы рассматриваем ситуацию, когда алгоритм мог ошибиться.

Итак, узнали рандомизированный алгоритм за $\log n$, существует детерминированный алгоритм за $Poly(\log n)$, но он сложный.

15.4. Решето Эратосфена

15.4.1. Классическое: $O(n \log \log n)$

Берем и вычеркиваем числа:

```
for (i = 2; i < n; ++i)
    if !mark[i]
        for (j = i+i; j <= n; j += i)
            mark[j] = 1
```

Верим, что k-тое простое число $k \log k$

15.4.2. Оптимизация

```
for (i = 2; i^2 <= n; ++i)
    if !mark[i]
        for (j = i^2; j <= n; j += i)
            mark[j] = 1
```

15.4.3. Оптимизированная оптимизация

$$Time = O(n)$$

Сейчас проблема в том, что в числа входим несколько раз. Значит, будем хранить минимальный делитель: $d[x]$. Очевидно, что он простой.

Рассмотрим $y = x * (prime_i <= d[x])$. Тогда $d[y] = prime_i$; x и y однозначно определены друг через друга.

p — массив, в который сохраняем простые числа.

```
for (x = 2; x <= n; ++x)
    if !d[x]
        d[x] = x, p[pn++] = x
    for (i = 0; i < pn && p[i] <= d[x] && x * p[i] <= n; ++i)
        d[x * p[i]] = p[i]
```

15.5. Диофантово уравнение

$$ax \equiv b(m)$$

Существует решение, если $b \boxtimes \gcd(a, m)$ Ra $ax_0 + my_0 = (a, m)$

Классрешений : $x_0 \frac{b}{(a,m)} + \frac{m}{(a,m)}k$, *где* $k \in \mathbb{Z}$

Если хотим узнать число, достаточно посчитать его остатки по каим-либо модулям, а потом воспользоваться китайской теоремой об остатках. Пусть L — длина числа.

К.Т.О.:

$$x \equiv a_1(m_1)$$
$$x \equiv a_2(m_2)$$

$$(m_1, m_2) = 1$$

Тогда:

$$x = a_1 m_2 m_2^{-1}(m_1) + a_2 m_1 m_1^{-1}(m_2)(m_1 m_2)$$

15.6. RSA

Мотивация: всем известны системы, в которых для шифрования и дешифрования нужен один и тот же секретный ключ (например, XOR с ключом). Однако бывает и асимметричное шифрование: есть отдельные ключи для шифрования и дешифрования, можно распространять только один. Тогда, например, можно ключ для шифрования оставить у себя, а ключ для дешифрования отдать всем — тогда можно будет проверять, что автор сообщения действительно имеет при себе «секретный ключ». Система RSA — один из первых примеров такого шифрования и пример, зачем модульная арифметика нужна в «реальной жизни».

У системы RSA есть один основной параметр: большое число n (модуль). Сообщениями являются вычеты по модулю n. Для шифровки сообщения выбирается некоторое число e («открытая экспонента»), сообщение m шифруется так:

$$m_{enc} = m^e \mod n$$

Ключом для шифрования является пара (n, e), а само шифрование можно выполнить при помощи быстрого возведения в степень. Теперь попробуем расшифровать. Попробуем найти такое d, что $(m^e)^d = m \mod n$.

$$\begin{aligned}(m^e)^d &= m \mod n\\ m^{ed} &= m \mod n\\ ed &= 1 \mod \varphi(n)\end{aligned}$$

Зная e и $\varphi(n)$ при помощи алгоритма Евклида несложно найти требуемое d. Заметим, что это верно даже в случае $m = 0$ (просто получаются везде нули). Таким образом, ключом для дешифрования является пара (n, d).

Вопрос в том, насколько эта система надёжна (в частности, по одной паре нельзя получить другую) и при каких значениях n. Вообще говоря, пары симметричны (экспонента в одной является обратной к другой) и при наличии одной пары и $\varphi(n)$ мы легко вычисляем вторую. Значит, надо, в частности, затруднить вычисление $\varphi(n)$. Из-за этого выбирать n простым плохо: $\varphi(p) = p - 1$. Точно так же плохо выбирать n, которое хорошо факторизуется: факторизуем $\Rightarrow$ узнаем $\varphi(n)$. Обычно выбирают $n = pq$, где p и q — большие простые числа примерно одинакового размера. В таком случае получается $\varphi(n) = (p-1)(q-1)$. Утверждается, что в таком случае уже достаточно сложно получать сообщение по m^e, зная только (n, e).

Теорема 15.6.1. Задачи факторизации числа n и представления n в виде $n = pq$ (где $p, q > 1$) эквивалентны, то есть одну можно свести к другой за полиномиальное время.

▶
- Если у нас есть факторизация n, то представить его в виде $n = pq$ можно очевидным образом: берём любой нетривиальный делитель n, называем его p, а $q = \frac{n}{p}$. Всё это можно сделать за полиномиальное время.
- Если у нас есть оракул, который умеет представлять n в виде $n = pq$, то мы можем факторизовать n рекурсивно: вызвали оракула, запустили факторизацию от p и q. Можно построить бинарное дерево рекурсивных вызовов, в листах которого будут простые числа и заметить, что внутренних вершин не больше, чем листьев. То есть вызовов оракула будет не больше, чем простых делителей у n (с учётом кратности), то есть не более $\log n$.

◀

Теорема 15.6.2. Если $n = pq$, где p и q простые, то задачи вычисления $\varphi(n)$ и факторизации n эквивалентны.

▶ По факторизации очевидно можно построить $\varphi(n)$.

А если у нас есть $a = \varphi(n) = (p-1)(q-1)$, то:

$$\begin{aligned}a &= (p-1)(q-1)\\ a &= pq - p - q + 1\\ a &= n - p - q + 1\\ q &= n - p - a + 1\end{aligned}$$

$$\begin{cases}q = n - p - a + 1\\ n = pq\end{cases} \Rightarrow n = p(n - p - a + 1)$$

$$\begin{aligned}n &= p(n - a + 1) - p^2\\ p^2 + p(a - n - 1) + n &= 0\end{aligned}$$

Получили квадратное уравнение на p, его можно честно решить по формуле. Получим два решения: p и q. ◀

15.7. Первообразные корни

Рассмотрим систему вычетов по простому модулю p. Мультипликативная группа по модулю p состоит из ровно $p-1$ элемента — это просто все не-нули. Однако с точки зрения умножения её так описывать не слишком удобно.

Def 15.7.1. g называется *первообразным корнем* по модулю p, если любой ненулевой вычет x можно представить в виде $x = g^k$ для некоторого k.

Теорема 15.7.1. По простому модулю p существует хотя бы один первообразный корень.

Замечание 15.7.1. На самом деле первообразные корни также существуют и по модулям 2, 4, p^k и $2p^k$. По модулю 2 его найти просто — это единица.

Теорема 15.7.2. g — первообразный корень тогда и только тогда, когда среди чисел g^1, g^2, ..., g^{p-1} единица встречается в первый раз на месте g^{p-1}. Заметим, что $g^{p-1} = 1$ всегда по малой теореме Ферма (или теореме Эйлера).

▶ $\Rightarrow$: Если для некоторого k имеем $g^k = 1$, то у нас g^x принимает всего k различных значений (так как после k всё зацикливается). Значит, $k \geqslant p-1$ по определению первообразного корня. С другой стороны, $k \leqslant p-1$, так как $g^{p-1} = 1$.

$\Leftarrow$: Пусть g не является первообразным корнем, тогда какое-то число не встретилось в последовательности степеней, значит, какое-то встретилось дважды, то есть:

$$\exists 1 \leqslant k_1 < k_2 \leqslant p-1 \colon g^{k_1} = g^{k_2}$$
$$g^{k_1} = g^{k_2}$$
$$g^{k_1-k_1} = g^{k_2-k_1}$$
$$g^0 = g^{k_2-k_1}$$
$$1 = g^{k_2-k_1}$$

То есть получаем, что единица должна была встретиться уже на месте $k_2 - k_1$, что строго меньше $p-1$, противоречие.

◀

Теорема 15.7.3. В мультипликативной группе по модулю p есть ровно $\varphi(p-1)$ первообразных корней.

▶ По теореме 15.7.1 хотя бы один корень точно есть, назовём его ω. Давайте возьмём какое-нибудь число из мультипликативной группы, скажем, ω^k. По теореме 15.7.2 оно будет являться первообразным корнем тогда и только тогда, когда в последовательности $(\omega^k)^1$, $(\omega^k)^2$), ... единица в первый раз встречается на месте $p-1$:

$$(\omega^k)^a = 1 \mod p$$
$$\omega^{ka} = 1 \mod p$$
$$\omega^{ka} = \omega^0 \mod p$$
$$ka = 0 \mod (p-1)$$

Минимальное ненулевое a, являющееся решением, равно $\frac{p-1}{\gcd(k,p-1)}$, что равно $p-1$ только в случае взаимной простоты k и $p-1$. Таким образом, существует всего $\varphi(p-1)$ различных k, которые являются первообразными корнями. Обращаю внимание, что тут нам совершенно неважно, какой именно первообразный корень взять в начале — мы пользуемся тем, что существует какой-нибудь и что остальные через него выражаются. ◀

Давайте научимся искать первообразный корень.

15.7.1. Поиск «в лоб»

Например, можно искать его «в лоб»: перебираем вычеты от меньших к большим и для каждого проверяем условие теоремы 15.7.2. Очевидно, что это будет работать за $O(pg)$, где g — минимальный первообразный корень, но на самом деле это даже $O(p)$ (без доказательства).

15.7.2. Быстрая проверка

Чтобы ускорить, давайте для начала научимся быстрее проверять первообразность. В каких вообще случаях может так случиться, что $g^a = 1$ (и a минимально), при условии, что $g^{p-1} = 1$? Вспомним доказательство теоремы 15.7.2: такое a имеет вид $\frac{p-1}{d}$, где d — некоторый делитель $p-1$. Но тогда $g^{ak} = 1$ для любого k. То есть, вообще говоря, достаточно будет проверить, что $g^a \neq 1$ для всех a вида $\frac{p-1}{p_i}$, где p_i — простые делители $p-1$. Если нам известна факторизация $p-1$ и в ней m различных простых делителей, то быстрым возведением в степень для каждого мы можем проверить, что $g^a \neq 1$, получим асимптотику $O(m \log p)$. Очевидно, что $m = O(\log p)$ (так как каждый простой делитель хотя бы 2) и получается оценка $O(\log^2 p)$. Хотя на самом деле, конечно, различных простых сильно меньше логарифма (в среднем их $O(\log \log n)$, теорема Харди-Рамануджана).

```
int powmod(int a, int b, int mod);

int p;
vector<int> p1_divisors; // простые делители p-1

bool check(int g) {
    for (int x : p1_divisors) {
        if (powmod(g, (p - 1) / x, p) == 1) {
            return false;
        }
    }
    return true;
}
```

Теперь мы можем находить ответ за $O(g \log^2 p)$. Гипотеза: первообразный корень не больше, чем $O(\log p)$ (без доказательства). То есть уже умеем за $O(log^3 p)$:

```
int g = 2;
while (!check(g)) g++;
```

15.7.3. Рандомизированный алгоритм

Так как мы знаем, что корней всего $\varphi(p-1)$, то тыкая в случайное число, мы попадаем с вероятностью $\frac{\varphi(p-1)}{p-1}$. Значит, матожидание числа шагов — $\frac{p-1}{\varphi(p-1)}$. Если $p-1 = p_1^{k_1} p_2^{k_2} \ldots$, то $\frac{p-1}{\varphi(p-1)} = \frac{p_1 p_2}{(p_1-1)(p_2-1)\ldots}$, что довольно немного. Непример, если у $p-1$ мало простых делителей (например, такое бывает при выборе модуля для Фурье вида $p = 2^k \cdot l$), то это совсем быстро. А точнее, $\frac{p-1}{\varphi(p-1)} = O(\log \log (p-1))$ (без доказательства). На каждом шаге проверка за $O(\log^2 p)$, вообще красота, получаем честный рандомизированный $O(\log^2 p \log \log p)$

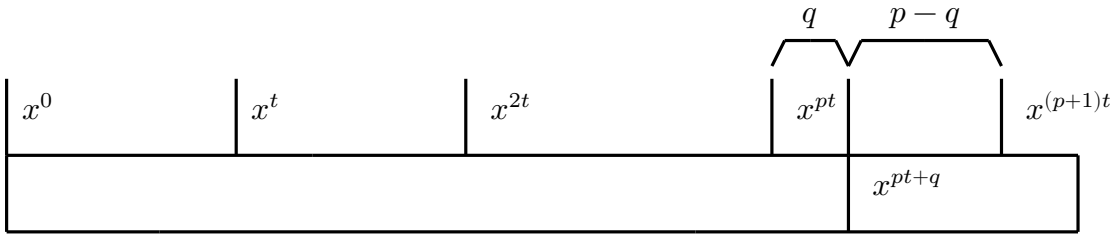
```
for (;;) {
    g = randint(1, p - 1);
    if (check(g)) break;
}
```

15.8. Дискретное логарифмирование

У нас появились первообразные корни. Разумеется, хочется научиться по числу a и первообразному корню g быстро находить такое k, что $a = g^k$. Можно немного обобщить и получим задачу дискретного логарифмирования по произвольному модулю m и произвольному основанию x.

Решение «в лоб» — возвести число x (основание) во все степени от 0 до $\varphi(m) - 1$, посмотреть, получилось или нет. На самом деле нам даже не надо знать $\varphi(m)$ — просто по очереди перебираем степени, пока не зациклимся, а зациклиться мы можем, только встретив единицу. Получаем время работы $O(\varphi(m))$ (даже если не знаем $\varphi(m)$).

Теперь применим сюда meet-in-the-middle (можно также рассматривать как корневую оптимизацию), получим алгоритм Шенкса (baby-step giant-step). Идея такая: давайте искать ответ не в виде $a = x^k$ (где $0 \leqslant k$), а в виде $a = x^{pt+q}$, где t — некоторая константа, $0 \leqslant q < t$. Немного преобразуем: $a = (x^t)^p x^q$ Теперь предподсчитаем $(x^t)^0$, $(x^t)^1$, ..., всего $\approx \frac{\varphi(m)}{t}$ штук (или $\frac{m}{t}$, если не знаем $\varphi(m)$). Цель — чтобы для последней степени $x^{\alpha t}$ выполнялось $\alpha t \geqslant \varphi(m)$.



Сложим все эти «большие шаги» в хэш-таблицу (по числу a^{pt} умеем получать p). Потом возьмём и в лоб найдём такое $q' = p - q$, что $ax^{q'} = x^{(p+1)t}$ (если $q = 0$, то мы сразу остановимся в $q' = 0$. Так как $\alpha t \geqslant \varphi(m)$, то такое q' обязательно найдётся (так как за каждым числом когда-то последует элемент хэш-таблицы), причём $q' < t$. Теперь легко найти соответствующие p и q.

То есть при наличии хэш-таблицы мы умеем логарифмировать за $O(t)$ операций. А хэш-таблицу можно построить за $O(\frac{m}{t} + \log t)$ операций (сначала нашли x^t быстрым возведением в степень, а потом посчитали его $\frac{m}{t}$ степеней «в лоб»). Положим $t = \sqrt{m}$ и получим алгоритм за $O(\sqrt{m})$. Заметим, что мы не пользовались про модуль вообще ничем: алгоритм работает для составных, умеет выдавать «ответа не существует».

15.9. Извлечение корня

Хотим решить сравнение по модулю m относительно x: $x^k = a$, a и k — константы. Мы умеем это делать, если у нас существует первообразный корень g и $a = g^\alpha$ (то есть $(a, m) = 1$). Для начала находим первообразный корень и дискретный логарифм a по основанию g. Это можно делать, например, за корень (но бывают алгоритмы и быстрее).

Теперь хотим решить уравнение. Так как $(a, m) = 1$, то $(x, m) = 1$ (если, конечно, x существует). То есть существует некоторое β такое, что $x = g^\beta$:

$$(g^\beta)^k = g^\alpha \mod m$$
$$g^{k\beta} = g^\alpha \mod m$$
$$k\beta = \alpha \mod \varphi(m)$$

Теперь надо решить последнее сравнение по модулю $\varphi(m)$ относительно β, мы уже знаем, как это делать. Если решилось — нашли корень, если не решилось — корня не существует.