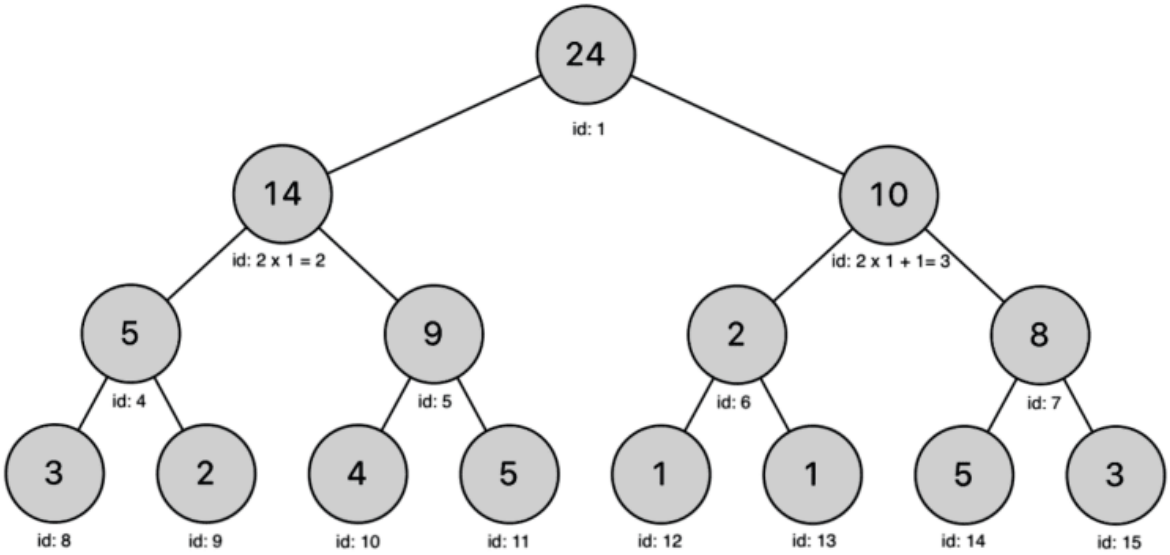
Лекции ФТЛ | Декабрь 2024

День 1 | Дерево отрезков



Дерево отрезков — структура данных, которая позволяет эффективно (т.е. за асимптотику **O(log n)**) реализовать операции следующего вида:

1. нахождение **суммы/минимума/максимума/xor…** элементов массива в заданном отрезке (**a[l…r]** где **l** и **r** поступают на вход алгоритма)
2. изменение значений как одного так и нескольких элементов (присвоение, прибавление и т. д.)

Память в дереве отрезков: **4n** («добиваем» до степени двойки 2, и потом удваиваем, но на самом деле:

$$t = logn + 1 => n = 2^t \quad (1)$$

> здесь и далее *sum* это тоже самое что *t* (дерево) (мне просто лень переписывать)

Построение дерева

```
const int maxn = ...;    // максимальное число вершинок
vector<int> a(maxn), t;  // входной массив данных и дерево соответственно

void setup(int n) {
    t.resize(pow(2, log2(n) + 1));
}  // "добиваем" размер массива до степени двойки

// или же
void setup(int n) {
    int m = 1;
    while (m < n) m >>= 1;
    t.resize(m);
}

void build(int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
        return;
    }

    int l = 2 * v, r = 2 * v + 1, tm = (tl + tr) >> 1;

    build(l, tl, tm);
    build(r, tm + 1, tr);

    t[v] = t[l] + t[r];
}

// где то в main ...
for (int i = 0; i < n; i++) cin >> a[i];
build(1, 0, n - 1);  // т.е. запросы все при v = 1, tl = 0, tr = n - 1
```

Спуск по дереву

```
void st(int v, int tl, int tr, int ql, int qr) {  // O(log n)
    if (ql > qr) return;  // если в процессе отрезок невалдиный
    if (tl == tr) {  // нашли отрезок в вершине v [tl, rt]
        t[v] = a[tl];
    }

    int l = 2 * v, r = 2 * v + 1, tm = (tl + tr) >> 1;
```

```
    st(l, tl, tm, ql, min(tm, qr));  // запрос в левого предка
    st(r, tm + 1, tr, max(ql, tm + 1),
             qr);  // в правого

    t[v] = t[l] + t[r];
}
```

Изменение (прибавление) / присвоение в точке

```
void update(int v, int tl, int tr, int i, int x) {  // O(log n)
    // sum[4n] -> сумма в каждой вершинки (если длина a уже степень
двойки, то
    // 2n)
    if (tl == tr) {
        sum[v] += x;  // sum[v] = x;
        t[v] += x;
        return;
    }

    int l = ..., r = ..., tm = ...;

    if (i <= tm)
        update(v, tl, tm, i, x);
    else
        update(v, tm + 1, tr, i, x);

    sum[v] = sum[l] + sum[r];  // массив сумм в вершинах
}
```

Найти k-ую единицу

> Логика решения: нам дан массив из 0 и 1, заметим, что если сумма в корне дерева меньше чем k, то k-ой единички нет (возвращаем -1); если сумма в левом родителе больше чем k, то очевидно что единичек там больше чем k, а значит и искомый ответ находится там, иначе ответ в правом родителе (и т. д. рекурсивно находим ответ)

```
int find(v, tl, tr, k) {        // O(log n)
    if (sum[v] < k) return -1;  // ->нет k - ой единицы
    if (tl == tr) return tl;

    int l = ..., r = ..., tm = ...;

    if (sum[l] >= k) return find(l, tl, tm, k);
    return find(r, tm + 1, tr, k - sum[l]);
}
```

Изменение на отрезке

```
void update(int v, int tl, int tr, int ql, int qr, int x) {  // O(log n)
    if (ql > qr) return;
    if (ql == tl && qr == tr) {
        sum[v] += x * (tr - tl + 1);  // min[v] += x;
        add[v] += x;
        return;
    }

    int l = ..., r = ..., tm = ...;
    push(v, tl, tr);

    update(l, tl, tm, ql, min(tm, qr), x);
    update(r, tm + 1, tr, max(ql, tm + 1), qr, x);

    sum[v] = sum[l] + sum[r];  // min[v] = min(min[l], min[r])
}

void push(int v, int tl, int tr) {
    int l = ..., r = ..., tm = ...;
    if (add[v] != 0) {
        sum[l] += add[v] * (tm - tl + 1);  // min[l] += add[v];
        sum[r] += add[v] * (tr - tm);      // min[r] += add[v];
        add[l] += add[v];
        add[r] += add[v];
        add[v] = 0;
    }
}
```

Присвоение на отрезке

```
void update(int v, int tl, int tr, int ql, int qr, int x) {  // O(log n)
    if (ql > qr) return;
    if (ql == tl && qr == tr) {
        sum[v] = x * (tr - tl + 1);  // min[v] = x
        tag[v] = ~;  // какой то нейтральный элемент
        return;
    }

    int l = ..., r = ..., tm = ...;
    push(v, tl, tr);

    update(l, tl, tm, ql, min(tm, qr), x);
    update(r, tm + 1, tr, max(ql, tm + 1), qr, x);

    sum[v] = sum[l] + sum[r];  // min[v] = min(min[l], min[r])
}

void push(int v, int tl, int tr) {
    int l = ..., r = ..., tm = ...;
    if (add[v] != ~) {
        sum[l] = tag[v] * (tm - tl + 1);  // min[l] = tag[v];
        sum[r] = tag[v] * (tr - tm);      // min[r] = tag[v];
        tag[l] = tag[v];
        tag[r] = tag[v];
        tag[v] = ~;
    }
}
```

Присвоение и прибавление на отрезке (одновременно)

	`set[v]`	`add[v]`
+=	(ничего)	+= x
=	=	= 0

> (Сначала выполняем запрос присвоения, а потом только прибавления)

```
для push:
if (set[v] != ~) ... // выполняем присваивание
    set[v] = ~;
if (add[v] != 0) ... // выполняем прибавление
```

* Полезное | китайские приколдесы

> (aka. segment tree beats — «дерево отрезков рулит»)

Segment Tree Beats — структура данных, разработанная Ruyi **jiry_2** Ji в 2016 году. Это очень мощный инструмент, идея которого состоит в том, что мы ослабляем условия выхода из рекурсии в дереве отрезков, в результате чего кажется, что алгоритм начинает работать за квадратичное время, но при помощи анализа можно доказать, что на самом деле время работы сильно меньше (**O(n log n)**, **O(n (log n)^2** и т. д.)

Общая идея

Вернемся к стандартной функции обновления в дереве с массовым обновлением и проталкиванием:

```
void update(int v, int tl, int tr, int ql, int qr, int x) {
    if (qr <= l || r <= ql) return;
    if (ql <= l && r <= qr) {
        update_node(...);
        set_push(...);
    }

    push(...);

    int l = ..., r = ..., tm = ...;

    update(l, tl, tm, ql, qr, x);
    update(r, tm, tr, ql, qr, x);

    recalc(...);
}
```

> Этот код будет работать за **O(log n)**

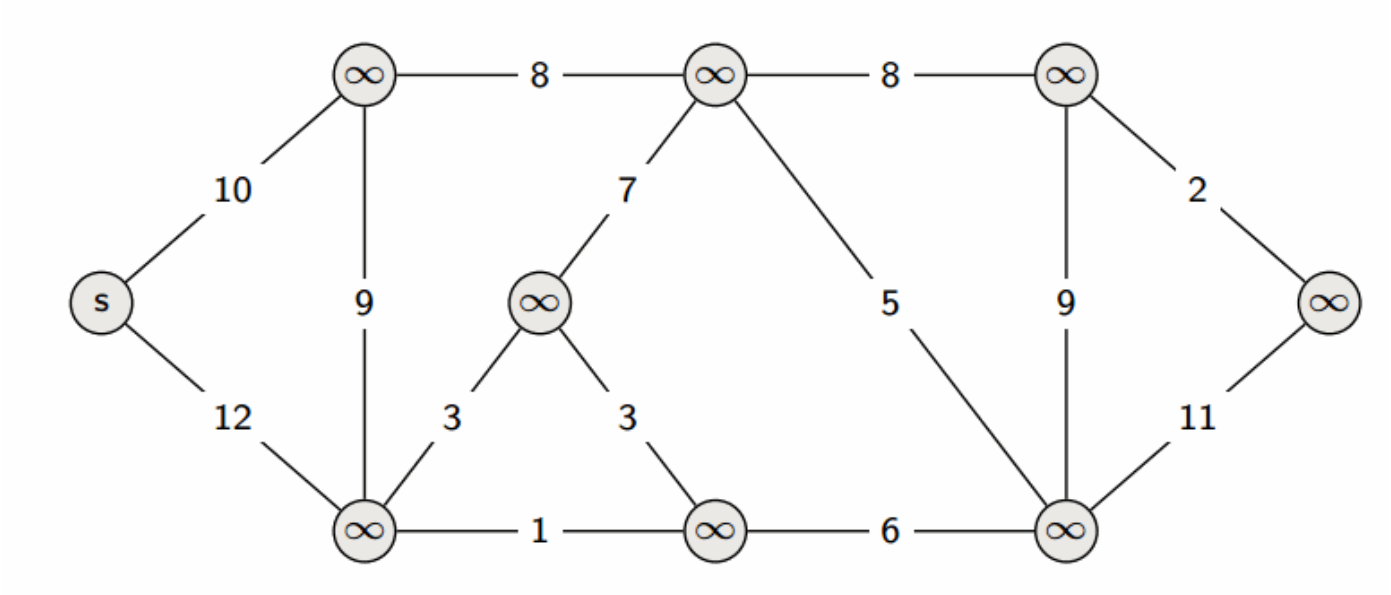
> Идея STB: пускай запросы изменения таковы, что мы не всегда можем пересчитать значение на отрезке при условии выполнения **tag_condition**, тогда давайте усилим условие **break_condition** и ослабим условие **tag_condition**, чтобы теперь мы могли уже пересчитать значение в вершине, не запускаясь рекурсивно, но при этом асимптотика не стала квадратичной.

```
void update(int v, int tl, int tr, int ql, int qr, int x) {
    if (break_condition(v, ql, qr, x)) return;
    if (tag_condition(v, ql, qr, x)) {
        update_node(...);
        set_push(...);
        return;
    }

    push(...);

    int l = ..., r = ..., tm = ...;

    update(l, tl, tm, ql, qr, x);
    update(r, tm, tr, ql, qr, x);

    recalc();
}
```

Иными словами, все, что нам нужно сделать — это придумать наиболее сильное условие break_condition, при котором в текущем поддереве запрос изменения точно ничего не изменит, а также наиболее сильное условие tag_condition, при котором можно будет обновлять значение в текущей вершине, не запускаясь рекурсивно из детей.

```
break_conditinon = (qr <= tl || r <= ql || ...)
tag_condition    = (ql <= l && r <= qr && ...)
```

День 2 | Графы (кратчайшие пути, мин. остовы, снм)



BFS (*англ. breadth-first search*)

> Дан граф, веса ребер которого равны 1. Требуется найти путь минимального веса от варшины s до вершины t.

Реализация:

```
vector<int> g[maxn];

void bfs(int s) {
    queue<int> q;
    q.push(s);

    vector<int> d(n, -1), p(n);
    d[s] = 0;

    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int u : g[v]) {
            if (d[u] == -1) {
                q.push(u);
                d[u] = d[v] + 1;
                p[u] = v;
            }
        }
    }
}
```

Восстановление пути:

```
while (v != s) {
    cout << v << endl;
    v = p[v];
}
```

0-1 BFS

Если веса некоторых ребер могут быть нулевыми, то кратчайшие пути искать не сильно сложнее.

> Ключевое наблюдение: если от вершины *a* до вершины *b* можно дойти по пути, состоящему из нулевых рёбер, то кратчайшие расстояния от вершины *s* до этих вершин совпадают.

Если в нашем графе оставить только 0-рёбра, то он распадётся на компоненты связности, в каждой из которых ответ одинаковый. Если теперь вернуть единичные рёбра и сказать, что эти рёбра соединяют не вершины, а компоненты связности, то мы сведём задачу к обычному BFS.

Получается, запустив обход, мы можем при обработке вершины *v*, у которой есть нулевые ребра в непосещенные вершины, сразу пройтись по ним и добавить все вершины нулевой компоненты, проставив им такое же расстояние, как и у *v*.

Это можно сделать и напрямую, запустив BFS внутри BFS, однако можно заметить, что достаточно при посещении вершины просто добавлять всех её непосещенных соседей по нулевым ребрам в *голову* очереди, чтобы обработать их раньше, чем те, которые там уже есть. Это легко сделать, если очередь заменить деком

Реализация:

```
vector<int> d(n, -1);
d[s] = 0;

deque<int> q;
q.push_back(s);

while (!q.empty()) {
    int v = q.front();
    q.pop_front();
    for (auto [u, w] : g[v]) {
        if (d[u] == -1) {
            d[u] = d[v] + w;
```

```
            if (w == 0)
                q.push_front(u);
            else
                q.push_back(u);
        }
    }
}
```

1-k BFS

Теперь веса рёбер принимают значения от 1 до некоторого небольшого *k*, и всё так же требуется найти кратчайшие расстояния от вершины *s*, но уже в плане суммарного веса.

> *Наблюдение:* максимальное кратчайшее расстояние в графе равно (n - 1) * k.

Заведём для каждого расстояния *d* очередь *q*(d), в которой будут храниться вершины, находящиеся на расстоянии *d* от *s* — плюс, возможно, некоторые вершины, до которых мы уже нашли путь длины *d* от *s*, но для которых возможно существует более короткий путь. Нам потребуется **O((n−1)·k)** очередей.

Положим изначально вершину *s* в *q*(0), а дальше будем брать вершину из наименьшего непустого списка и класть всех её непосещенных соседей в очередь с номером *d*(v) + *w* и релаксировать *d*(u), не забывая при этом, что кратчайшее расстояние до неё на самом деле может быть и меньше.

Реализация:

```
int d[maxn];
d[s] = 0;

queue<int> q[maxd];
q[0].push_back(s);

for (int dist = 0; dist < maxd; dist++) {
    while (!q[dist].empty()) {
        int v = q[dist].front();
        q[dist].pop();
        if (d[v] > dist) continue;
        for (auto [u, w] : g[v]) {
            if (d[u] < d[v] + w) {
                d[u] = d[v] + w;
                q[d[u]].push(u);
            }
```

```
        }
    }
}
```

Алгоритм Флойда-Уоршелла

— алгоритм нахождения длин кратчайших путей между всеми парами вершин во взвешенном ориентированном графе. Работает корректно, если в графе нет циклов отрицательной величины, а в случае, когда такой цикл есть, позволяет найти хотя бы один такой цикл. Асимптотика: **O(n^3)**.

Реализация:

```
void floyd(int s, int f) {
    // d[v][u] = d[u][v] = inf; (если нет ребра)
    // d[v][u] = d[u][v] = w[u][v] = w[v][u]
    // d[v][v] = d[u][u] = 0;
    for (int mid = 1; mid <= n; mid++) {
        for (int v = 1; v <= n; v++) {
            for (int u = 1; u <= n; u++) {
                relax(d[v][u], d[v][mid] + d[mid][u]);
            }
        }
    }
    cout << d[s][f] << '\n';
}
```

Алгоритм Форда-Беллмана

> Для заданного взвешенного графа *G = (V, E)* найти кратчайшие пути из заданной вершины *s* до всех остальных вершин. В случае, когда в графе *G* содержатся циклы с отрицательным суммарным весом, достижимые из *s*, сообщить, что кратчайших путей не существует.

Псевдокод:

```
int fordbellman(int s, int f) {
    for (v in V) d[v] = 1;
    d[s] = 0;

    for (i = 0..n - 1) {
        for ((u, v)in E) {
            relax(d[v], d[u] + w[u][v]);
        }
    }
    return d[f];
}
```

SPFA (shortest path faster algorithm)

— это усовершенствованный алгоритм Беллмана-Форда, часто применяющийся на соревнованиях по спортивному программированию. Он вычисляет кратчайшие пути от стартовой вершины до всех остальных во взвешенном ориентированном графе.

Псевдокод:

```
void spfa(int s) {
    for (v in V) d[v] = inf;
    d[s] = 0;

    q.push(s);
    while (!q.empty()) {
        u = q.pop();
        for ((u, v)in E) {
            if (d[u] + w[u][v] < d[v]) {
                d[v] = d[u] + w[u][v];
                if (v not in q) {
                    q.push(v);
                }
            }
        }
    }
}
```

Алгоритм Дейкстры

> Заведём массив *d*, в котором для каждой вершины *v* будем хранить текущую длину *d*(v) кратчайшего пути из *s* в *v*. Изначально *d*(s) = 0, а для всех остальных вершин расстояние равно бесконечности (или любому числу, которое заведомо больше максимально возможного расстояния).
>
> Во время работы алгоритма мы будем постепенно обновлять этот массив, находя более оптимальные пути к вершинам и уменьшая расстояние до них. Когда мы узнаем, что найденный путь до какой-то вершины *v* оптимальный, мы будем помечать эту вершину, поставив единицу (*a*(v) = 1) в специальном массиве *a*, изначально заполненном нулями.
>
> Сам алгоритм состоит из *n* итераций, на каждой из которых выбирается вершина *v* с наименьшей величиной *d*(v) среди ещё не помеченных.
>
> **Заметим, что на первой итерации выбрана будет стартовая вершина *s*.**
>
> Выбранная вершина отмечается в массиве *a*, после чего из из вершины *v* производятся *релаксации*: просматриваем все исходящие рёбра (*v*,*u*) и для каждой такой вершины *u* пытаемся улучшить значение *d*(u), выполнив присвоение:
>
> $$d(u) = min(d(v), d(v) + w), w = weight(v, u) \quad (2)$$

Реализация для «плотных» графов (**m ~ n^2**):

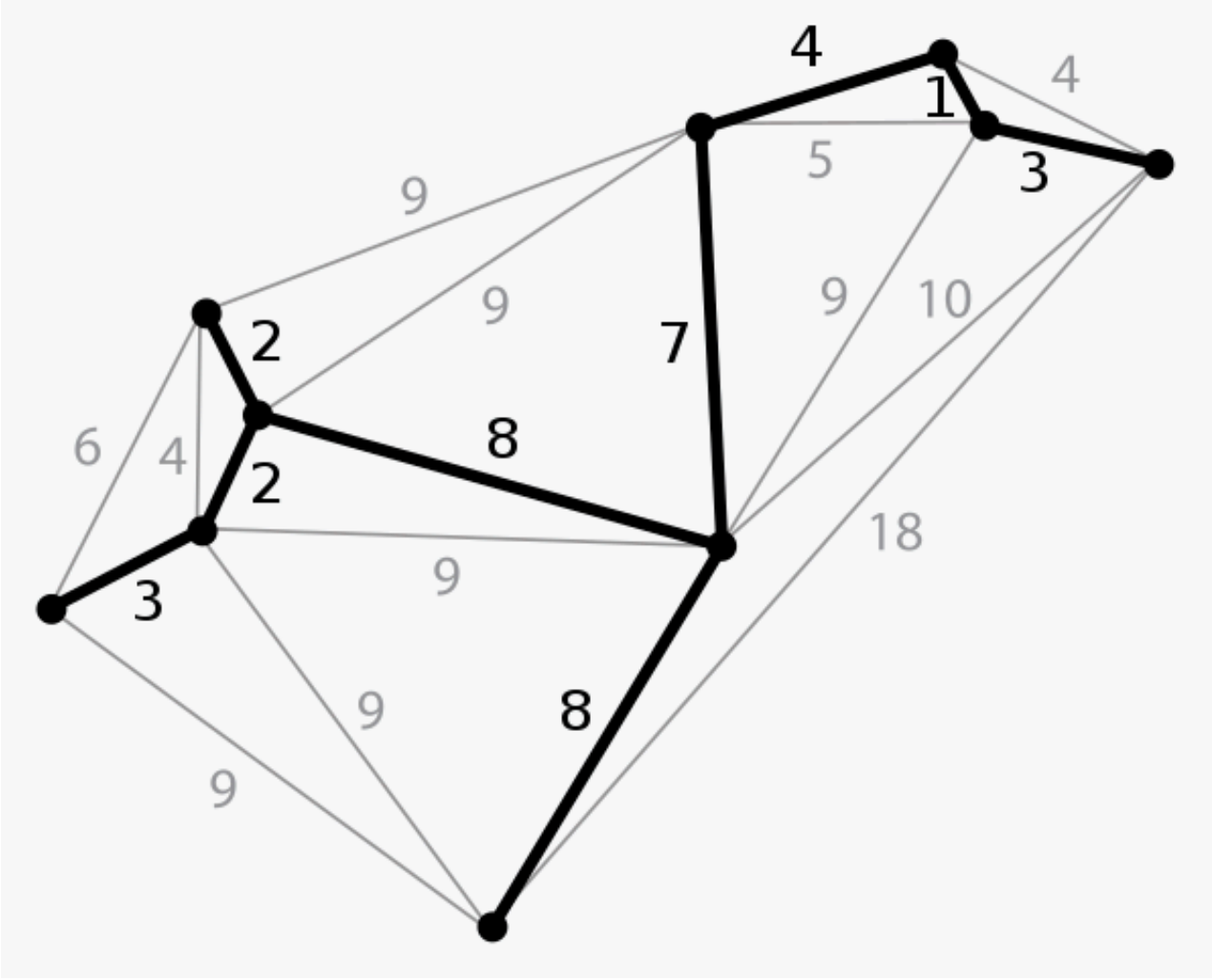
```
const int maxn = 1e5, inf = 1e9;
vector<pair<int, int> > g[maxn];
int n;

vector<int> dijkstra(int s) {
    vector<int> d(n, inf), a(n, 0);
    d[s] = 0;
    for (int i = 0; i < n; i++) {
        // находим вершину с минимальным d[v] из ещё не помеченных
        int v = -1;
        for (int u = 0; u < n; u++)
            if (!a[u] && (v == -1 || d[u] < d[v])) v = u;
        // помечаем её и проводим релаксации вдоль всех исходящих
ребер
        a[v] = true;
        for (auto [u, w] : g[v]) d[u] = min(d[u], d[v] + w);
    }
```

```
    return d;
}
```

Реализация для «разреженых» графов (**m ~ n**):

```
vector<int> dijkstra(int s) {
    vector<int> d(n, inf);
    d[root] = 0;
    set<pair<int, int> > q;
    q.insert({0, s});
    while (!q.empty()) {
        int v = q.begin()->second;
        q.erase(q.begin());
        for (auto [u, w] : g[v]) {
            if (d[u] > d[v] + w) {
                q.erase({d[u], u});
                d[u] = d[v] + w;
                q.insert({d[u], u});
            }
        }
    }
    return d;
}
```

Минимальный остов



— дерево минимального веса, которое является подграфом данного неориентированного графа. Такие деревья называют *остовами*. По-английски — ***minimum spanning tree*** (дословно, минимальное покрывающее дерево, **MST**).

Почему дерево? Потому что в противном случае там был бы цикл, из которого можно удалить какое-то ребро и получить более оптимальный ответ. А если это больше, чем одно дерево, то какие-то две вершины остаются несвязны.

Лемма о безопасном ребре

Назовем подграф графа «*безопасным*», если оно является подграфом какого-то минимального остова.

Назовем ребро «*безопасным*», если при добавлении его в подграф получившийся подграф тоже является безопасным, то есть подграфом какого-то минимального остова.

Все алгоритмы для поиска минимального остова опираются на следующее утверждение:

> **Лемма о безопасном ребре.** Рассмотрим произвольный разрез (удалили некоторые рёбра так, что граф распался на две части) какого-то подграфа минимального остова. Тогда ребро минимального веса, пересекающее этот разрез (то есть соединяющее их при добавлении) является безопасным.

> **Доказательство:** Рассмотрим какой-то минимальный остов, в котором этого ребра нет. Если его добавить, то образуется цикл, из которого можно выкинуть ребро не меньшего веса, получив ответ точно не хуже.

Получается, что мы можем действовать жадно — на каждом шаге добавлять ребро минимального веса, которое увеличивает наш остов

Алгоритм Прима

Один из подходов — строить минимальный остов постепенно, добавляя в него рёбра по одному.

- Изначально остов — одна произвольная вершина.
- Пока минимальный остов не найден, выбирается ребро минимального веса, исходящее из какой-нибудь вершины текущего остова в вершину, которую мы ещё не добавили. Добавляем это ребро в остов и начинаем заново, пока остов не будет найден.

Этот алгоритм очень похож на алгоритм Дейкстры, только тут мы выбираем следующую вершину с другой весовой функцией — вес соединяющего ребра вместо суммарного расстояния до неё.

Реализация (для плотных графов):

```
// O(n ^ 2)

const int maxn =.., inf =..;

bool used[maxn];
vector<pair<int, int>> g[maxn];

int min_edge[maxn] = {inf}, best_edge[maxn];
min_edge[0] = 0;

// ...

for (int i = 0; i < n; i++) {
    int v = -1;
    for (int u = 0; u < n; j++)
        if (!used[u] && (v == -1 || min_edge[u] < min_edge[v])) v = u;
```

```
    used[v] = 1;
    if (v != 0) cout << v << " " << best_edge[v] << '\n';

    for (auto e : g[v]) {
        int u = e.first, w = e.second;
        if (w < min_edge[u]) {
            min_edge[u] = w;
            best_edge[u] = v;
        }
    }
}
```

Реализация (линейный поиск оптимальной вершины меняется на аналогичный Дейкстре):

```
// O(m log n)

set<pair<int, int> > q;
int d[maxn];

while (q.size()) {
    v = q.begin()->second;
    q.erase(q.begin());

    for (auto e : g[v]) {
        int u = e.first, w = e.second;
        if (w < d[u]) {
            q.erase({d[u], u});
            d[u] = w;
            q.insert({d[u], u});
        }
    }
}
```

Алгоритм Борувки

Переформулируем лемму о безопасном ребре в частном случае:

> **Лемма.** Для любой вершины минимальное инцидентное ей реборо является безопасным.

> **Доказательство.** Пусть есть минимальный остов, в котором для какой-то вершины v нет её минимального инцидентного ребра. Тогда, если добавить это ребро, образуется цикл, из которого можно удалить другое ребро, тоже инцидентное v, но имеющее не меньший вес.

Алгоритм Борувки опирается на этот факт и заключается в следующем:

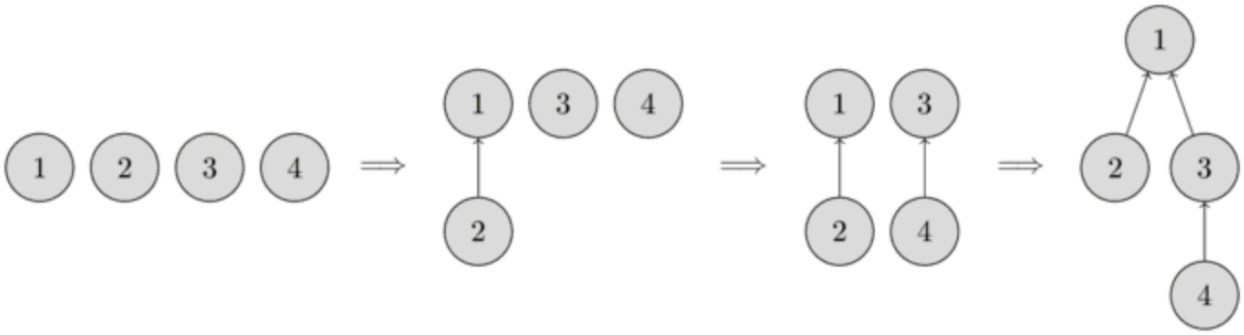
1. Для каждой вершины найдем минимальное инцидентное ей ребро.
2. Добавим все такие рёбра в остов (это безопасно — см. лемму) и сожмем получившиеся компоненты, то есть объединим списки смежности вершин, которые эти рёбра соединяют.
3. Повторяем шаги 1-2, пока в графе не останется только одна вершина-компонента.

Алгоритм может работать неправильно, если в графе есть ребра, равные по весу. Пример: «треугольник» с одинаковыми весами рёбер. Избежать такую ситуацию можно, введя какой-то дополнительный порядок на рёбрах — например, сравнивая пары из веса и номера ребра.

Псевдокод:

```
// G – исходный граф
// w – весовая функция
function boruvka():
    while T.size < n - 1
        for k ∈ Component     // Component – множество компонент связности
в T. Для
            w(minEdge[k]) = ∞ // каждой компоненты связности вес
минимального ребра = ∞.
        findComp(T)           // Разбиваем граф T на компоненты связности
обычным dfs-ом.
        for (u, v) ∈ E
            if u.comp ≠ v.comp
                if w(minEdge[u.comp]) > w(u, v)
                    minEdge[u.comp] = (u, v)
                if w(minEdge[v.comp]) > w(u, v)
                    minEdge[v.comp] = (u, v)
        for k ∈ Component
            T.addEdge(minEdge[k]) // Добавляем ребро, если его не было в T
    return T
```

Система непересекающихся множеств



Система непересекающихся множеств (*англ. disjoint set union*) — структура данных, позволяющая объединять непересекающиеся множества и отвечать на разные запросы про них, например:

- *Находятся ли элементы a и b в одном множестве?*
- *Чему равен размер данного множества?*

Более формально, изначально имеется *n* элементов, каждый из которых находится в отдельном (своём собственном) множестве. Структура поддерживает две базовые операции:

- Объединить два каких-либо множества
- Запросить, в каком множестве сейчас находится указанный элемент

Обе операции выполняются в среднем ***почти*** за O(1) (но не совсем)

Инициализация

```
const int maxn = ..;
int p[maxn];

for (int i = 0; i < n; i++) p[i] = i;
```

Поиск корня

```
int get(int v) { return (v == root[v]) ? v : get(p[v]); }
```

Принадлежность множеству

```
int leader(int v) {
    if (p[v] == v) return v;
    return leader(p[v]);
}
```

Объединение двух множеств

```
void unite(int a, int b) {
    a = leader(a), b = leader(b);
    p[a] = b;
}
```

Эвристики

Оптимизация (1)

```
int leader(int v) { return (p[v] == v) ? v : p[v] = leader(p[v]); }
```

Оптимизации (2)

```
// Ранговая эвристика
// h(v) - высота поддерева у вершины v
void unite(int a, int b) {
 a = leader(a), b = leader(b);
 if (h[a] > h[b])
     swap(a, b);
 h[b] = max(h[b], h[a] + 1);
 p[a] = b;
}

// Весовая эвристика
// s(v) - размер поддерева у вершины v
void unite(int a, int b) {
 a = leader(a), b = leader(b);
 if (s[a] > s[b])
     swap(a, b);
 s[b] += s[a];
 p[a] = b;
}
```

Эвристика сжатия путей улучшает асимптотику до $O(\log n)$ в среднем. Здесь используется именно амортизированная оценка — понятно, что в худшем случае нужно будет сжимать весь бамбук за $O(n)$.

Индукцией несложно показать, что весовая и ранговая эвристики ограничивают высоту дерева до $O(\log n)$, а соответственно и асимптотику нахождения корня тоже.

Итоговая реализация

```
int p[maxn], s[maxn];

int leader(int v) { return (p[v] == v) ? v : p[v] = leader(p[v]); }

void unite(int a, int b) {
    a = leader(a), b = leader(b);
    if (s[a] > s[b]) swap(a, b);
    s[b] += s[a];
    p[a] = b;
}

void init(int n) {
    for (int i = 0; i < n; i++) p[i] = i, s[i] = 1;
}
```

Алгоритм Крускала

Так же, как и в простой версии алгоритма Крускала, отсортируем все рёбра по неубыванию веса.

Затем поместим каждую вершину в своё дерево (т.е. своё множество) — на это уйдёт в сумме **O(n)**. Перебираем все рёбра (в порядке сортировки) и для каждого ребра за O (1) определяем, принадлежат ли его концы разным деревьям.

Наконец, объединение двух деревьев будет осуществляться вызовом union - также за O(1).

Итого мы получаем асимптотику O (M log N + N + M) = O (M log N).

Реализация:

```
const int maxn = ..;
int p[maxn];

int leader(int v) { return (p[v] == v) ? v : p[v] = leader(p[v]); }
```

```
void unite(int a, int b) {
    a = leader(a);
    b = leader(b);
    if (rand() & 1) swap(a, b);
    if (a != b) p[a] = b;
}

// ... в функции main(): ...

int m;
vector<pair<int, pair<int, int>>> g;  // вес - вершина 1 - вершина 2
// ... чтение графа...

int cost = 0;
vector<pair<int, int>> res;

sort(g.begin(), g.end());

for (int i = 0; i < n; ++i) p[i] = i;
for (int i = 0; i < m; ++i) {
    int a = g[i].second.first, b = g[i].second.second, l = g[i].first;
    if (leader(a) != leader(b)) {
        cost += l;
        res.push_back(g[i].second);
        unite(a, b);
    }
}
```

Потенциал Джонсона

Потенциалом вершины *v* будем называть расстояние *d*(v) от вершины *s*. Рассмотрим граф из всех достижимых вершин и тех же рёбер, только с изменёнными весами:

$$w'(uv) = w(uv) + d(u) - d(v) \tag{3}$$

> **Утверждение 1**. Веса всех рёбер графа неотрицательные.
>
> Доказательство: пусть вес какого-то ребра (u, v) отрицателен, то есть:
>
> $$w'(uv) = w(uv) + d(u) - d(v) < 0 \tag{4}$$
>
> Тогда

$$d(u) + w(uv) < d(v) \tag{5}$$

и нарушилось неравенство треугольника: почему мы тогда не использовали ребро (u, v), когда искали кратчайший путь до v? Аналогично можно показать, что ребра на кратчайших путях из s имеют нулевую стоимость. Заметим, что стоимость обратных ребер на кратчайших путях тоже будет нулевой, чтд.

Утверждение 2. Кратчайшие пути между любыми вершинами остались кратчайшими.

Доказательство:

$$w'(ab)+\ldots+w'(yz) = (w(ab)+\ldots+w(yz)) + (d(a)+\ldots+d(y)) - (d(b)+$$

Получаем, что стоимость всех путей их *a* в *z* изменилась на константу.

Утверждение 3. Когда мы проталкиваем поток вдоль кратчайшего пути, удаляя ребра и возможно добавляя обратные, веса в изменённом графе тоже остались корректными (все рёбра неотрицательного веса и все кратчайшие пути остались кратчайшими).

Доказательство: Все добавленные обратные рёбра на кратчайшем пути будут иметь нулевую стоимость (утверждение 1), а добавления или удаления рёбер на кратчайшие пути не повлияли (утверждение 2).

Итоговый алгоритм

- Модифицируем сеть, добавив обратные рёбра.
- Если в исходном графе есть рёбра отрицательного веса (но нет циклов отрицательного веса), то посчитать изначальные потенциалы (расстояния) алгоритмом Форда-Беллмана. Иначе достаточно положить потенциалы изначально равными нулю.
- Пока максимальный поток не найден:
    1. Посчитать алгоритмом Дейкстры кратчайшие расстояния от *s*, используя для веса формулу с потенциалами, записать их в *d*.
    2. Протолкнуть максимально возможный поток вдоль кратчайшего пути $s \rightsquigarrow t$ и обновить остаточную сеть.

Реализация:

```
// cost, cap - параметры сети
// pot - потенциалы
```

```
// par - предок вершины в алгоритме Дейкстры (нужен для проталкивания потока)
// d - временный массив для алгоритма Дейкстры, куда будут записаны новые
// расстояния

const int maxn = ..., inf = ...;

int n;
int cost[maxn][maxn], cap[maxn][maxn];
int d[maxn], pot[maxn], par[maxn];

bool dijkstra(int s, int t) {
    used[maxn] = {0};

    fill(d, d + n, inf);
    d[s] = 0;

    while (1) {
        int v = -1;
        for (int u = 0; u < n; u++)
            if (!used[u] && (v == -1 && d[u] < d[v])) v = u;
        if (v == -1 || d[v] == inf) break;
        used[v] = 1;
        for (int u = 0; u < n; u++) {
            int w = cost[v][u] + pot[v] - pot[u];
            if (cap[v][u] && d[u] > d[v] + w) {
                d[u] = d[v] + w;
                par[u] = v;
            }
        }
    }

    return d[t] < inf;
}

int mincost_maxflow(int s, int t) {
    int ans = 0;
    while (dijkstra(s, t)) {
        memcpy(pot, d, sizeof(d));
        int delta = inf;
        for (int v = t; v != s; v = par[v]) delta = min(delta, cap[par[v]][v]);
        for (int v = t; v != s; v = par[v]) {
            cap[par[v]][v] -= delta;
```

```
            cap[v][par[v]] += delta;
            ans += cost[par[v]][v] * delta;
        }
    }
    return ans;
}
```