# Алгоритмы и структуры данных.

Андрей Тищенко @AndrewTGk

2024/2025

**Семинар 3 сентября.**

## Оценка за работу на семинаре

Посещаемость и ответы у доски. Каждое посещение это $0.5$ балла, ответ у доски - $1$ балл (максимум $10$ баллов).
Задача про определение конечности списка. (рисуночки, идеи)
Задача: написать очередь через два стека.

---

```
struct queue {
    stack st1, st2;
    void push(int x);
    void pop();
    int front();
    void transition();
};
void push(int x) {
    st1.push(x);
}
void pop() {
    if (st2.empty()) {
        transition();
    }
    st2.pop();
}
void transition() {
    while (st1.size()) {
        st2.push(st1.top());
        st1.pop();
    }
}
```

---

Семинар 10 сентября

На лекции не разобрали случай $c > \log_b a$. Под этот случай подходит алгоритм Карацубы, но его разберём на следующей лекции. Вопрос в лекции про $k \log k = n \Rightarrow k = O(?)$. $k = O\left(\frac{n}{\log n}\right)$, подставим это:
$\frac{n}{\log n} \log \frac{n}{\log n} = \frac{n}{\log n}(\log |n| - \log\log n)$
$\log |n| - \log\log n = O(\log n) \Rightarrow \frac{n}{\log n}(\log |n| - \log\log n) = n\frac{O(\log n)}{\log n} = nO(1) =$
$= O(n)$

### Задача 1

Найти подотрезок с заданной суммой $S$. Разделяем отрезок на две половины.
Ответ может лежать в одной из половин или на из пересечении, тогда нужно сохранить в $map$ все суффиксы левого отрезка, просмотреть префиксы правого отрезка, для отрезков длины $sum$ нас интересует существование суффикса $S - sum$.

То есть формируем $map\ M$ и смотрим $M.count(S - sum) == 1$
$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = n \log n$ если отрезок полностью внутри одной половины. Иначе будет $T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n) \Rightarrow$
$\Rightarrow T(n) = n \log^2 n$.

## Задача 2

Даны $n$ точек, нужно найти две точки, расстояние между которыми минимально. Должно работать бытсрее $O(n^2)$.
Сделаем поворот плоскости таким образом, чтобы ни одна пара точек не лежала на одной вертикальной прямой. Проведём одну вертикальную прямую так, чтобы слева и справа от неё было поровну точек и эта прямая не пересекает ни одну точку. Пусть кратчайшее растояние между точками слева $d_1$, а справа - $d_2$.
Тогда обозначим $d = min(d_1,\ d_2)$. Проведём две параллельные прямые слева с справа от изначальной прямой на расстоянии $d$. Поделим образовавшуюся линию на кваратики со сторонами $\frac{d}{2}$, в каждом таком квадрате не больше одной точки, так как иначе расстояние между ними было бы меньше $d$, но мы рекуррентно поняли, что в любой половине минимальное расстояние $d$.
Тогда останется посмотреть не более 11 квадратов для каждой рассматриваемой точки. Однако для такого просмотра нужно отсортировать точки по высоте. На рекуррентных вызовах тоже нужно будет это сортировать, поэтому можно mergить результаты из двух половин. Тогда ассимптотика алгоритма будет:
$T(n) = 2T(\frac{n}{2}) + 11n = O(n \log n)$
Если на каджом шаге делать сортировку заново, то ассимптотика будет:
$T(n) = 2T(\frac{n}{2}) + 11n + n \log n = O(n \log^2 n)$, то есть медленнее.

## Задача 3

Дано несколько точек, нужно упорядочить их в полярных координатах.
$(v_i \times v_j) > 0 \Leftrightarrow i < j$
Но может быть проблема: $\exists i,\ j,\ k:\ i < j < k < i$

Семинар 17 сентября

---

```
#include <vector>

// Multiplying vectors of the same size
std::vector<long long> bMult(std::vector<long long> a, std::vector<long long> b) {
    int n = a.size();
    std::vector<long long> c(2*n, 0);
    for (int k = 0; k < 2*n; ++k) {
        for (int i = std::max(0, k - n + 1); i <= k && i < n; ++i) {
            c[k] += a[i]*b[k - i];
        }
    }
    return c;
}

std::vector<int> kar(std::vector<long long> a, std::vector<long long> b) {
    int n = a.size();
    int nHalf = n/2;
    if (n <= 32) {
        return bMult(a, b);
    }
    std::vector<long long> a0(nHalf, 0), a1(nHalf, 0), b0(nHalf, 0), b1(nHalf, 0);
    // a0(a.begin(), a.begin() + nHalf) - fill with iterators
    std::vector<long long> a0b0, a1b1, abab;
    // simple fill
    for (int i = 0; i < nHalf; ++i) {
        a0[i] = a[i];
```

```
            a1[i] = a[i + nHalf];
            b0[i] = b[i];
            b1[i] = b[i + nHalf];
        }
        a0b0 = kar(a0, b0);
        a1b1 = kar(a1, b1);
        // a0 will become a0 + a1 and b0 will become b0 + b1
        for (int i = 0; i < nHalf; ++i) {
            a0[i] += a1[i];
            b0[i] += b1[i];
        }
        abab = kar(a0, b0);
        abab -= (a0b0 + a1b1);
        std::vector<long long> ans(2*n, 0);
        for (int i = 0; i < n; ++i) {
            ans[i] += a0b0[i];
            ans[i + n/2] += abab[i];
            ans[i + n] += a1b1[i];
        }
        return ans;
    }
```

---

Пример длинной арифметики:
$x = 10: \quad 123 \cdot 225 = (3 + 2x + x^2) \cdot (5 + 2x + 2x^2) = 15 + 16x + 15x^2 + 6x^3 + 2x^4$
В многочленах ответ получили, перевод в число в векторах будет выглядеть примерно так:

| 15 | 16 | 15 | 6 | 2 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$\Rightarrow$

| 5 | 7 | 6 | 7 | 2 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$x = 100: \quad 1001 \cdot 1002 = 1003002 = (10x + 1)(10x + 2) = 100x^2 + 30x + 2$

| 2 | 30 | 100 | 0 |
|---|---|---|---|

$\Rightarrow$

| 2 | 30 | 0 | 1 |
|---|---|---|---|

. Но если это восстановить, получаем 10302, но это неправда, ведь 2 и 0, которые хранятся в векторе по факту являются 02 и 00:

| 02 | 30 | 00 | 1 |
|---|---|---|---|

$\Rightarrow 1003002$

## Задача

Даются числа $a_1,\ a_2, \ldots,\ a_n$ и $b_1,\ b_2, \ldots,\ b_m$.
Известно, что $0 \leqslant a_i, b_i \leqslant 10^5$, $n,\ m \leqslant 10^5$.
$\forall k \in \{0,\ 1, \ldots,\ 2 \cdot 10^5\},\ num[k] = \#\{(i,\ j) : a_i + b_j == k\}$
$A(x) = x^{a_1} + x^{a_2} + \cdots + x^{a_n}$
$B(x) = x^{b_1} + x^{b_2} + \cdots + x^{b_m}$
$C(x) = A(x) \cdot B(x) = c_0 + c_1 x + \cdots + \underbrace{c_k}_{=num[k]} x^k + \ldots$
$x^{a_i} \cdot x^{b_j} = x^k \Rightarrow c_k = \#\{(i,\ j) : a_i + b_j == k\}$, что нам и нужно.
Задача считается сложной и может появиться на коллоквиуме как вопрос на 9, 10 баллов.

## Алгоритм Штрассена

$a_{11}b_{11} + a_{12}b_{21} = d + d_1 + v_1 - h_1$
$a_{11}b_{12} + a_{12}b_{22} = h_1 + v_2$
$a_{21}b_{11} + a_{22}b_{21} = h_2 + v_1$
$a_{21}b_{12} + a_{22}b_{22} = d + d_2 + v_2 - h_2$
$d = (a_{11} + a_{22})(b_{11} + b_{22})$
$d_1 = (a_{12} - a_{22})(b_{21} + b_{22})$
$d_2 = (a_{21} - a_{11})(b_{11} + b_{12})$
$h_1 = (a_{11} + a_{12})b_{22}$
$h_2 = (a_{21} + a_{22})b_{11}$
$v_1 = a_{22}(b_{21} - b_{11})$
$v_2 = a_{11}(b_{12} - b_{22})$

Семинар 24 сентября.

## Задача 1

Пусть есть функция $rnd2()$, которая равновероятно возвращает 0 или 1.
а. На её основе написать $rnd4()$ и $rnd8()$

```
int rnd2() {...}
int rnd4() {
    return rnd2() + 2*rnd2();
}
int rnd8() {
    return rnd2() + 2*rnd2() + 4*rnd2();
}
```

б. На её основе написать $rnd3()$.

```
int rnd2() {...}
int rnd3() {
    int x[3] = {rnd2(), rnd2(), rnd2()};
    while (x[0] + x[1] + x[2] != 1) {
        x[0] = rnd2();
        x[1] = rnd2();
        x[2] = rnd2();
    }
    for (int i = 0; i < 3; i++) {
        if (x[i]) {
            return i;
        }
    }

}

int rnd3_author() {
    int x = 3;
    while(x == 3) {
        x = rnd(4);
    }
    return x;
}
```

Вероятность неудачи $rnd3() = \frac{5}{8}$, у $rnd3\_author() = \frac{1}{4}$.
Матожидание количества вызовов $rnd4()$ из $rnd3()$:
$E = \frac{3}{4} \cdot 1 + \frac{1}{4}(1 + E) \Rightarrow \frac{3}{4}E = 1 \Rightarrow E = \frac{4}{3}$

## Задача 2

Найти наименьшую обхатывающую окружность для n точек.
Можно решить вероятностным алгоритмом за ожидаемое время $O(n)$.
Выберем случайную точку, построим для неё оболочку (окружность радиуса 0), добавляем ещё одну, строим обхатывающую для пары точек.
Добвляем ещё одну точку. Возможны следующие исходы:

1. Точка попала в круг
2. Точка не попала в круг

Утверждение (без доказательства, оно сложное): точка, не попавшая в круг должна быть на границе новой окружность.

```
class Point {...};
class Circle {... bool owns(Point) ...};

Circle ansCircle = ...;

Circle minD(std::vector<Point> x) {
    k = x.size();
    if (ansCircle.owns(x[k - 1])) {
        return minD(x.pop_back());
    }
    return minD1(x[k - 1], x.pop_back())
}
Circle minD1(Point y, std::vector<Point> x) {
    k = x.size();
    Circle c = mind1(y, x.pop_back());
    if (c.owns(x[k - 1])) {
        return c;
    }
    return minD2(y, x[k - 1], x.pop_back());
}

Circle minD2(Point y1, Point y2, std::vector<Point> x) {
    k = x.size();
    if (!k) {
        return Circle(y1, y2);
    }
    c = minD2(y1, y2, x.pop_back());
    if (c.owns(x[k - 1])) {
        return c;
    }
    return circle(x[k - 1], y1, y2);
}
```

Матожидание minD2: $E\big(T_2(k)\big) = T_2(k-1) + \frac{k-1}{k} \cdot 0 + \frac{1}{k} 1 = O(k)$
Матожидание minD1: $E\big(T_1(k)\big) = E\big(T_1(k-1)\big) + \frac{k-1}{k} \cdot 0 + \frac{1}{k} \underbrace{T_2(k-1)}_{O(k-1)} =$

$= T_1(k-1) + \frac{O(k-1)}{k} = T_1(k-1) + O(1)$

Семинар 1 октября.

Нахождение $k$-ой порядковой статистики:

```
std::vector<int> a;
void kth_element(int lhs, int rhs, int k) {
    int cnt = rhs - lhs + 1;
    int idx = lhs + rand()%cnt;
    std::vector<int> less, more_eq(1, a[idx]);
    for (int i = lhs; i < rhs; i++) {
        if (i == idx) {
            continue;
        }
        if (a[i] >= a[idx]) {
            more_eq.push_back(a[i]);
        } else {
            less.push_back(a[i]);
        }
    }
    for (int i = 0; i < less.size(); i++) {
        a[lhs + i] = less[i];
    }
    for (int i = 0; i < more_eq.size(); i++) {
        a[lhs + less.size() + i] = more_eq[i];
    }
    if (less.size() == k - 1) {
        return;
    } else if (less.size() > k - 1) {
        kth_element(lhs, lhs + less.size() - 1, k);
    } else {
        kth_element(lhs + less.size() + 1, rhs, k - less.size() - 1);
    }
}
```

Сортировка 5 элементов за наименьшее количество сравнений.

```
int a1, a2, a3, a4, a5;
std::cin >> a1 >> a2 >> a3 >> a4 >> a5;
```

Вставлю фотографию с семинара.

Даны числа $a_1, \ldots, a_{16}$. Составить из них магический квадрат.

Сумма элементов в линии будет $S = \frac{\sum a_i}{4}$.

Выберем $Q = |s_1 - S| + |s_2 - S| + \cdots + |s_{10} - S| \to \min$, где $s_i$ — сумма элементов в $i$ линии.

Пусть $T_i = 100\,000 \cdot 0.99^i + C$ ($C$ — какая-то константа, нужно подбирать).

Пусть действием будет swap двух чисел.

$Q' < Q$ - делаем изменение. Иначе делаем с $p = e^{-\frac{Q'-Q}{T_i}}$

SkipList с операцией find (считаем, что на каждом уровне есть $-\infty$ и $+\infty$).

```
struct node {
    int key;
    node *next, *down;
}

node* find(node* b, int x) {
    node* l = b;
    while (true) {
        while (l->next && l->next->key < x) {
            l = l->next;
        }
        node* r = l->next;
        if (r->key == x) {
            return r;
        }
        if (!r->down) {
            return r;
        }
        l = l->down;
    }
}
```

Семинар 8 октября

Чилсенное интегрирование:

```
inline double f(const double x) {
    return 2*x*x*x + 5;
}

inline double sum(const double left, const double right) {
    return (right - left)*( f(left) + f(right) +
           4*f((left + right)/2) )/6;
}

double integral (const double left, const double right, int n) {
    double sum = 0;
    double current_left = left;
    double diff = (right - left)/n
    for (int x = 0; x < n; x++) {
        sum += sum(current_left, current_left + diff);
        current_left += diff;
    }
    return sum;
}
```

Численное интегрирование с сеткой переменной плотности.

```
constexpr int initialN = 10;
constexpr double eps = 1e-8;

inline double f(const double x) {
    return 2*x*x*x + 5;
}

inline double sum_v(const double left, const double right) {
    return f(left)*(right - left);
}

// from seminar
double intergral_v(double l, double r) {
    double int1 = sum_v(l, r);
    double diff = (r - l)/2;
    double int2 = sum_v(l, l + diff) + sum_v(l + diff, r);
    if (int1 - int2 < eps && int1 - int2 > -eps) {
        return int2;
    }
    return (intergral_v(l, l + diff) + intergral_v(l + diff, r));
}

// Modification to define minimal amount of points on line
double int_v(double left, double right, int n = initialN) {
    double to_ret = 0, cur_l = left;
    double h = (right - left)/n;
    for (int i = 0; i < n; i++) {
        to_ret += intergral_v(cur_l, cur_l + h);
        cur_l += h;
    }
    return to_ret;
}
```

Семинар 8 ноября.

## Задача

Раскрасить компоненты связности в неориентированном графе с помощью dfs.

```
std::vector<std::vector<int>> graph;
std::vector<int> color;
void dfs(int vertex, int current_color) {
    if (color[vertex]) {
        return;
    }
    color[vertex] = current_color;
    for (int neighbour : graph[vertex]) {
        dfs(neighbour, current_color);
    }
}

int main() {
    int cur_color = 1;
    for (int i = 0; i < color.size; i++) {
        if (!color[i]) {
            dfs(i, cur_color++);
        }
```

```
    }
}
```

Определить наличие цикла в неориентированном графе.

```
std::vector<int> used;
std::vector<std::vector<int>> graph;
bool dfs (int vertex, std::vector<int>& parent) {
    used[vertex] = 1;
    for (int neighbour : graph[vertex]) {
        if (!used[neighbour]) {
            parent[neighbour] = vertex;
            if (dfs(neighbour, parent)) {
                return true;
            }
        } else if (neighbour != parent[vertex]) {
            int x = vertex;
            while (x != neighbour) {
                std::cout << x << ' ';
                x = parent[x];
            }
            cout << x;
            return true;
        }
    }
    return false;
}
```

Определить двудольность графа, покрасить доли.

```
std::vector<int> used;
std::vector<std::vector<int>> graph;
bool dfs(int vertex, int color) {
    used[vertex] = color;
    for (int neighbour : graph[vertex]) {
        if (!used[neighbour]) {
            if (!dfs(neighbour, 3 - color)) {
                return false;
            }
        } else if (color == used[neighbour]) {
            return false;
        }
    }
    return true;
}
```

Написать dfs, считающий глбину и размер поддерева в каждой вершине.

```
std::vector<int> size;
std::vector<int> depth;
std::vector<int> parent;
std::vector<std::vector<int>> graph;
void dfs (int vertex) {
    size[vertex] = 1;
    for (int child : g[vertex]) {
        if (child != parent[v]) {
            parent[child] = vertex;
            depth[child] = depth[vertex] + 1;
            dfs(child);
            size[vertex] += size[child];
        }
    }
}

int main() {
    // read graph
    depth.resize(n);
    size.resize(n);
    parent.assign(n, root_index);
    depth[root_index] = 0;
    dfs(root_index);
}
```

Напишем bfs (посчитаем расстояние до корня).

```
std::vector<int> dist;
void bfs(int vertex) {
    dist.assign(n, -1);
    used[vertex] = 1;
    std::queue<int> q;
    q.push(vertex);
    dist[v];
    while (!q.empty()) {
        int cur = q.front();
        q.pop;
        for (int neighbour : graph[cur]) {
            if (dist == -1) {
                dist = dist[cur] + 1;
                q.push(neighbour);
            }
        }
    }
}
```

Напишем СНМ (DSU), работающее за $O(n \log n)$:

```
struct dsu {
    int n;
    std::vector<int> parent, size;
    dsu(int n): n(n) {
        parent.resize(n);
        iota(parent.begin(), parent.end(), 0); // fills vector incrementing values
        // from the beginning every node is its own parent
        size.assign(n, 1);
    }
};

int col(int a) { // color
    if (a == parent[a]) {
        return a; // color is index of root node
    }
    return col(parent[a]);
}

void unite(int a, int b) {
    a = col(a);
    b = col(b);
    if (a == b) {
        return;
    }
    if (size[a] < size[b]) {
        std::swap(a, b);
    }
    parent[b] = a;
    size[a] += size[b];
}
```

Высота каждого дерева размера $k$ будет $O(\log k)$, так как каждая дочерняя вершина когда-то подвешивалась к корню, то есть её размер был меньше, значит каждый ребёнок увеличивает не более чем в $\frac{k}{2}$ раз.

Напишем СНМ (DSU), работающее за $O(\alpha(n) \cdot n)$. Отличие только в return statement функции col (переподвешиваем все вершины на пути к корню):

```
struct dsu {
    int n;
    std::vector<int> parent, size;
    dsu(int n): n(n) {
        parent.resize(n);
        iota(parent.begin(), parent.end(), 0); // fills vector incrementing values
        // from the beginning every node is its own parent
        size.assign(n, 1);
    }
};

int col(int a) { // color
    if (a == parent[a]) {
        return a; // color is index of root node
    }
    return parent[a] = col(parent[a]);
}

void unite(int a, int b) {
    a = col(a);
    b = col(b);
    if (a == b) {
        return;
    }
    if (size[a] < size[b]) {
        std::swap(a, b);
    }
    parent[b] = a;
    size[a] += size[b];
}
```

Добавим функцию отката к $O(n \log n)$ реализации CHM. В результате выполнения unite(a, b) могут поменяться: a, b, size[a], size[b], parent[a], parent[b]. В нашей реализации parent[a] и size[b] не меняются, при этом size[a] можно восстановить, используя size[b], parent[b] был просто b, так как подвешиваются корни, a — не меняется. Итого, доостаточно знать только b.

```
struct dsu {
    ...
    std::vector<int> b_save;
};
int col(int a) { // color
    if (a == parent[a]) {
        return a; // color is index of root node
    }
    return col(parent[a]);
}
void unite(int a, int b) {
    a = col(a);
    b = col(b);
    if (a == b) {
        b_save.push_back(-1);
        return;
    }
    if (size[a] < size[b]) {
        std::swap(a, b);
    }
    parent[b] = a;
    b_save.push_back(b);
    size[a] += size[b];
}
void rollback() {
    int b = b_save.back();
    b_save.pop_back();
    if (b == -1) {
        return;
    }
    int a = parent[b];
    parent[b] = b;
    size[a] -= size[b];
}
```

Нарисовать график из галереи (если тут нет графика, пишите).

$sum[v] = \sum_{col=1}^{N} dist(v,\ col)$

Храним map<int, int> M[N], где M[col] = dist(v, col)

Квадратичное решение:

У некоторой вершины несколько детей, у каждого

Улучшение до $O\left(\sum_v sz[v]\right) = O(n^2)$ (так как у нас несбалансированное дерево)

Можно менять местами вершины разных размеров, таким образом приливаем меньшие части дерева к большим.

То есть swap(M[v], M[w]). При этом нужно как-то обновить расстояния от w до её детей (так как на путь добавилась ещё одна вершина v). Для этого добавим массив add[v].

Будет комбинация операций:

```
swap(M[v], M[w]);
add[v] = add[w] + 1;
sum[v] = sum[w] + M[v].size;
```

Теперь выполняем за $O(n \log n \cdot$ опер.$)$, где опер. — операция в

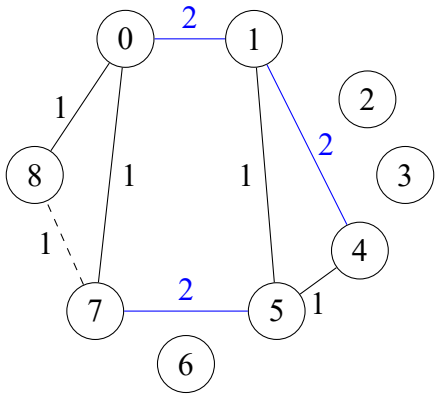
Семинар 22 ноября.

## Задача

Как понять, что ребро лежит в минимальном остове?

Построим какой-то минимальный остов. Если наше ребро вошло, то оно очевидно входит. Если нет, то рассмотрим цикл, который оно образует с рёбрами минимального остова. Если вес какого-то ребра из цикла равен весу нашего ребра, значит мы можем его поместить в минимальный остов.

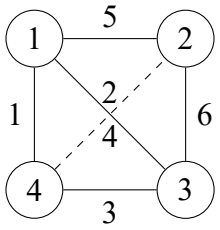
## Задача




Берём вершины $0,\ 1,\ 4,\ 5,\ 7,\ 8$. Ребро между $1$ и $4$ брать в минимильный остов не будем.

## Задача

Дан граф:



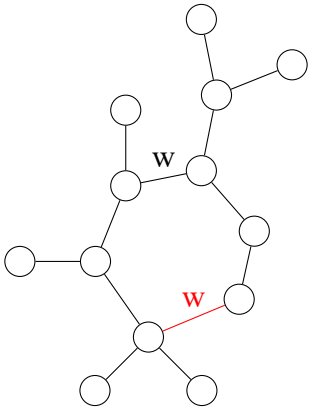

Ребро 2 соединяет вершины $1,\ 3$. Ребро 4 соединяет $2,\ 4$.
Веса рёбер:

| Ребро | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Вес | 7 | 7 | 7 | 9 | 9 | 10 |

Докажем утверждение:
Любая из перестановок рёбер:
$1,\ 2,\ 3,\ 5,\ 4,\ 6$
$1,\ 2,\ 3,\ 4,\ 5,\ 6$
$\left.\begin{matrix} 1,\ 3,\ 2 \\ 2,\ 1,\ 3 \\ 2,\ 3,\ 1 \\ 3,\ 1,\ 2 \\ 3,\ 2,\ 1 \end{matrix}\right\} \times 2$
Даст нам все возможные минимальные остовы для этого графа хотя бы один раз.
Рассматриваем классы эквивалентности по весу ребра. В каждом классе любая перестановка даёт валидный минимальный остов, ч.т.д.

## Задача

Можно ли из одного остова получить другой операциями (удаление ребра, добавление нового ребра).
Рассмотрим граф:




Пусть у нас есть список отсортированных по весу рёбер. Рассмотрим компоненты связности с рёбрами одинакового веса, в таком случае веса в этой задаче можно не рассматривать.
Возьмём два произольных минимальных остова, склеим их общие вершины.
Получаем ситуацию с рёбрами без весов и без общих рёбер. Возьмём какой-то лист из первого остова, отсоединим его и соединим ребром из второго остова. Повторяем алгоритм (удаляем общие рёбра), теперь нам нужно соединить остовы размером на 1 меньше. По индукции легко доказать.

Семинар 29 ноября.

Алгоритм Прима за $O(n^2)$:

```
int d[N] = INT_MAX, used[N] = 0; // N - amount of vertices
std::vector<std::vector<pair<int, int>>> g;

int main() {
    int s; // start vertex
    sum = 0;
    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if ((v == -1 || d[j] < d[v]) && !used[j]) {
                v = j;
            }
        }
        sum += d[v];
        used[v] = true;
        for (auto [vertex, weight] : g[v]) {
            if (!used[vertex] && d[vertex] > weight) {
                d[vertex] = weight;
            }
        }
    }
}
```

Превратим это в алгоритм Дейкстры за $O(n^2)$:

```
int d[N] = INT_MAX, used[N] = 0; // N - amount of vertices
std::vector<std::vector<pair<int, int>>> g;

int main() {
    int s; // start vertex
    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if ((v == -1 || d[j] < d[v]) && !used[j]) {
                v = j;
            }
        }
        used[v] = true;
        for (auto [vertex, weight] : g[v]) {
            if (!used[vertex] && d[vertex] > weight + d[v]) {
                d[vertex] = weight + d[v];
            }
        }
    }
}
```

Напишем Дейкстру за $O(m \log n)$

```
int d[N] = INT_MAX, used[N] = false; // N - amount of vertices
std::set<std::pair<int, int>> g;

int main() {
    int s; // start vertex
    g.insert({d[s], s});
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (v == -1 || d[j] < d[v]) {
                v = q.begin().second;
                q.erase(q.begin());
            }
        }
        used[v] = true;
        for (auto [u, w] : g[v]) {
            if (d[u] > w + d[v]) {
                q.erase({d[u], u});
                d[u] = w + d[v];
                q.insert({d[u], u});
            }
        }
    }
}
```

Прим за $O(m \log n)$

```
int d[N] = INT_MAX; // N - amount of vertices
std::set<std::pair<int, int>> g;

int main() {
    int s; // start vertex
    int sum = 0;
    g.insert({d[s], s});
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if ((v == -1 || d[j] < d[v]) && d[j] != -1) {
                v = q.begin().second;
                q.erase(q.begin());
            }
        }
        sum += d[v];
        d[v] = -1;
        for (auto [u, w] : g[v]) {
            if (d[u] > w) {
                q.erase({d[u], u});
                d[u] = w;
                q.insert({d[u], u});
            }
        }
    }
}
```

## Алгоритм Флойда

Работает с матрицей смежности.

```
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            // check for negative weight cycles
            if (d[i][j] != inf && d[k][j] != inf) {
                d[i][j] = std::min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }
}
```

Пусть cor[i][j] = true, если расстояние от i до j может быть определено корректно (на пути нет циклов отрицательного веса). Для добавления этого ммассива нужно будет написать следующее:

```
std::vector<std::vector<bool>> cor(i, std::vector<bool>(j, true));
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int t = 0; t < n; t++) {
            if (d[i][t] < inf && d[t][j] < inf && d[t][t] < 0) {
                cor[i][j] = false;
            }
        }
    }
}
```

## Алгоритм Форда-Беллмана

Работает со списком рёбер.

```
struct Edge {
    int from;
    int to;
    int weight;
    Edge() {}
};

std::vector<Edge> graph;
// fill it somehow
for (int phase_count; phase_count < n - 1; phase_count++) {
    for (const auto& edge : graph) {
        if (d[edge.to] == inf) {
            continue;
        }
        d[edge.to] = std::min(d[edge.to], d[edge.from] + edge.weight);
    }
}

// adding a flag to check if we can end cycle earlier
for (int phase_count; phase_count < n - 1; phase_count++) {
    bool flag = false;
    for (const auto& edge : graph) {
        if (d[edge.from] != inf && d[edge.to] > d[edge.from] + edge.weight) {
            flag = true;
        }
    }
}
```

```
for (int ph = 0; ph < n - 1; ph++) {
    a) d[ph + 1] = d[ph];
    b) d[ph + 1].assign(n, inf);
    for (const auto& edge: graph) {
        if (d[ph][edge.from] == inf) {
            continue;
        }
        d[ph + 1][edge.to] = std::min(d[ph + 1][edge.to], d[ph][e.from] + edge.weight);
    }
}
```

Теперь в d[k][v] хранится вес кратчайшего пути от s до v, состоящего из:
Со строчкой a): не более чем k рёбер.
Со строчкой b): ровно из k рёбер.

## Алгоритм Левита

Работает со списком смежности.

```
std::vector<int> id(n, 0); // all vertices in M0 from the beginning
std::vector<int> d(n, inf);
d[s] = 0;
std::deque<int> q;
q.push_back(s);
while (!q.empty()) {
    int v = q.front();
    q.pop_front();
    for (auto& [w, len] : g[v]) {
        // w stands for second vertex
        // len stands for weight of v -- w edge
        if (d[w] <= d[v] + len) {
            continue;
        }
        if (id[w] == 0) {
            q.push_back(w);
        }
        if (id[w] == 2) {
            q.push_front(w);
        }
        id[vertex] = 1;
    }
}
```