

# LU ICPC kladīte ;)

## Contents

1. C++ programming language .....	1
1.1. Input/Output disable sync .....	1
1.2. Optimization pragmas .....	1
1.3. Printing structs .....	1
1.4. Lambda func for sorting .....	1
2. Algebra .....	1
2.1. Binary exponentiation .....	1
2.2. Extended euclidean .....	1
2.3. Modular inversion & division .....	1
2.4. Linear Diophantine equation .....	1
2.5. Linear sieve .....	2
2.6. Matrix multiplication .....	2
2.7. FFT .....	2
3. Geometry .....	2
3.1. Dot product .....	2
3.2. Cross product .....	2
3.3. Line-point distance .....	2
3.4. Shoelace formula .....	2
3.5. Segment to line .....	3
3.6. Three point orientation .....	3
3.7. Line-line intersection .....	3
3.8. Check if two segments intersect .....	3
3.9. Graham's scan .....	3
4. Data structures .....	4
4.1. Treap .....	4
4.2. Lazy segment tree .....	4
4.3. Sparse table .....	4
4.4. Fenwick tree .....	4
4.5. Trie .....	4
4.6. Aho-Corasick .....	5
4.7. Disjoint Set Union .....	5
4.8. Merge sort tree .....	5
5. Graph algorithms .....	6
5.1. Bellman-Ford .....	6
5.2. Dijkstra .....	6
5.3. Floyd-Warshall .....	6
5.4. Bridges & articulations .....	6
5.5. Dinic's max flow / matching .....	6
5.6. Flow with demands .....	7
5.7. Kosaraju's algorithm .....	7
5.8. Lowest Common Ancestor .....	7
5.9. General matching in a graph .....	7
6. String Processing .....	7
6.1. Knuth-Morris-Pratt (KMP) .....	7
6.2. Suffix Array .....	8
6.3. Longest common prefix with SA .....	8

6.4. Rabin-Karp .....	8
6.5. Z-function .....	8
6.6. Manacher's .....	8
7. Dynamic programming .....	8
7.1. Convex hull trick .....	8
7.2. Longest Increasing Subsequence .....	9
7.3. SOS DP (Sum over Subsets) .....	9

## C++ programming language

### 1.1. Input/Output disable sync

```
ios_base::sync_with_stdio(false);
cin.tie(NULL); cout.tie(NULL);
```

### 1.2. Optimization pragmas

```
// change to O3 to disable fast-math for geometry problems
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt,tune=native")
```

### 1.3. Printing structs

```
ostream& operator<<(ostream& os, const pair<int, int>& p) {
    return os << "(" << p.first << ", " << p.second << ")";
}
```

### 1.4. Lambda func for sorting

```
using ii = pair<int,int>;
vector<ii> fracs = {{1, 2}, {3, 4}, {1, 3}};
// sort positive rational numbers
sort(fracs.begin(), fracs.end(),
    [](const ii& a, const ii& b) {
        return a.fi*b.se < b.fi*a.se;
    });
```

## Algebra

### 2.1. Binary exponentiation

```
ll m_pow(ll base, ll exp, ll mod) {
    base %= mod;
    ll result = 1;
    while (exp > 0) {
        if (exp & 1) result = ((ll)result * base) % mod;
        base = ((ll)base * base) % mod;
        exp >>= 1;
    }
    return result;
}
```

### 2.2. Extended euclidean

$$a \cdot x + b \cdot y = \gcd(a, b)$$

```
int gcd_ext(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1; y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
```

### 2.3. Modular inversion & division

gcd\_ext defined in Section 2.2.

$$\exists x(a \cdot x \equiv 1(\bmod m)) \Leftrightarrow \gcd(a, m) = 1$$

```
int mod_inv(int b, int m) {
    int x, y;
    int g = gcd_ext(b, m, &x, &y);
    if (g != 1) return -1;
    return (x%m + m) % m;
}
int m_divide(ll a, ll b, ll m) {
    int inv = mod_inv(b, m);
    assert(inv != -1);
    return (inv * (a % m)) % m;
}
```

### 2.4. Linear Diophantine equation

gcd\_ext defined in Section 2.2.

$$a \cdot x + b \cdot y = c$$

$$\left\{ x = x_0 + k \cdot \frac{b}{g}; y = y_0 - k \cdot \frac{a}{g} \right\}$$

```
bool find_x0_y0(int a, int b, int c, int &x0, int &y0, int &g) {
    g = gcd_ext(abs(a), abs(b), x0, y0);
    if (c % g) return false;
    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
```

## 2.5. Linear sieve

```
const int N = 10000000;
vector<int> lp(N+1);
vector<int> pr;

for (int i=2; i <= N; ++i) {
    if (lp[i] == 0) {
        lp[i] = i;
        pr.push_back(i);
    }
    for (int j = 0; i * pr[j] <= N; ++j) {
        lp[i * pr[j]] = pr[j];
        if (pr[j] == lp[i]) break;
    }
}

// "inherit" vector's constructor
using vector::vector;

Matrix operator *(Matrix other)
{
    int rows = size();
    int cols = other[0].size();
    Matrix res(rows, vector<int>(cols));
    for(int i=0; i<rows; i++)
        for(int j=0; j<other.size(); j++)
            for(int k=0; k<cols; k++)
                res[i][k] += at(i).at(j)*other[j][k];
    return res;
}
};
```

## 2.6. Matrix multiplication

## 2.7. FFT

```
using ld = long double;
const int N = 1<<18;
const ld PI = acos(-1.0);
struct T {
    ld x, y;
    T() : x(0), y(0) {}
    T(ld a, ld b=0) : x(a), y(b) {}

    T operator/=(ld k) { x/=k; y/=k; return (*this); }
    T operator*(T a) const { return T(x*a.x - y*a.y, x*a.y + y*a.x); }
    T operator+(T a) const { return T(x+a.x, y+a.y); }
    T operator-(T a) const { return T(x-a.x, y-a.y); }
};

void fft(T* a, int n, int s) {
    for (int i=0, j=0; i<n; i++) {
        if (i>j) swap(a[i], a[j]);
        for (int l=n/2; (j^=l) < l; l>>=1);
    }

    for(int i = 1; (1<<i) <= n; i++){
        int M = 1 << i;
        int K = M >> 1;
        T wn = T(cos(s*2*PI/M), sin(s*2*PI/M));
        for(int j = 0; j < n; j += M) {
            T w = T(1, 0);
            for(int l = j; l < K + j; ++l){
                T t = w*a[l + K];
                a[l + K] = a[l] - t;
                a[l] = a[l] + t;
                w = wn*w;
            }
        }
    }
}

void multiply(T* a, T* b, int n) {
    while (n&(n-1)) n++; // ensure n is a power of two
    fft(a,n,1);
    fft(b,n,1);
    for (int i = 0; i < n; i++) a[i] = a[i]*b[i];
    fft(a,n,-1);
    for (int i = 0; i < n; i++) a[i] /= n;
}

int main() {
    // Example polynomials: (2 + 3x) and (1 - x)
    T a[10] = {T(2), T(3)};
    T b[10] = {T(1), T(-1)};
    multiply(a, b, 4);
    for (int i = 0; i < 10; i++)
        std::cout << int(a[i].x) << " ";
}
```

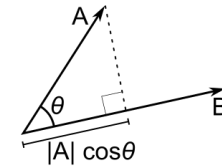
## Geometry

### 3.1. Dot product

$$a \cdot b = |a| |b| \cos(\theta)$$

$$a \cdot b = a_x b_x + a_y b_y$$

$$\theta = \arccos\left(\frac{a_x b_x + a_y b_y}{|a| |b|}\right)$$



Projection of a onto b:

$$\frac{a \cdot b}{|b|}$$

### 3.2. Cross product

$$a \times b = |a| |b| \sin(\theta)$$

$$a \times b = a_x b_y - a_y b_x$$

$\theta$  is positive if a is clockwise from b

### 3.3. Line-point distance

Line given by  $ax + by + c = 0$  and point  $(x_0, y_0)$ .

$$\text{distance} = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

The coordinates of this point are:

$$x = \frac{b(bx_0 - ay_0) - ac}{a^2 + b^2}$$

$$y = \frac{a(-bx_0 + ay_0) - bc}{a^2 + b^2}$$

### 3.4. Shoelace formula

$$2A = \sum_{i=1}^n \begin{vmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{vmatrix}$$

### 3.5. Segment to line

$$((P_x, P_y), (Q_x, Q_y)) \rightarrow Ax + By + C = 0$$

$$A = P_y - Q_y$$

$$B = Q_x - P_x$$

$$C = -AP_x - BP_y$$

Rationing the obtained line equation:

1. divide  $A, B, C$  by their GCD
2. if  $A < 0$  or  $A = 0 \wedge B < 0$  then multiply all by  $-1$

### 3.6. Three point orientation

```
// 0 --> p, q and r are collinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p1, Point p2, Point p3)
{
    // See 10th slides from following link for derivation
    // of the formula
    int val = (p2.y - p1.y) * (p3.x - p2.x)
              - (p2.x - p1.x) * (p3.y - p2.y);

    if (val == 0)
        return 0; // collinear

    return (val > 0) ? 1 : 2; // clock or counterclock wise
}
```

### 3.7. Line-line intersection

From system of linear equations derived Cramer's rule:

$$x = -\frac{c_1 b_2 - c_2 b_1}{a_1 b_2 - a_2 b_1}$$

$$y = -\frac{a_1 c_2 - a_2 c_1}{a_1 b_2 - a_2 b_1}$$

If the denominator equals zero, the lines are parallel or coincident.

### 3.8. Check if two segments intersect

```
bool on_segment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;

    return false;
}

bool do_intersect(Point p1, Point q1, Point p2, Point q2)
{
    // Find the four orientations needed for general and
    // special cases
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    if (o1 != o2 && o3 != o4)
        return true;

    if (o1 == 0 && on_segment(p1, p2, q1)) return true;
    if (o2 == 0 && on_segment(p1, q2, q1)) return true;
    if (o3 == 0 && on_segment(p2, p1, q2)) return true;
    if (o4 == 0 && on_segment(p2, q1, q2)) return true;
}
```

### 3.9. Graham's scan

```
struct pt {double x, y;};
int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}

bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(),
            a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    a = st;
}
```

## Data structures

### 4.1. Treap

// Implicit segment tree implementation

```
struct Node{
    int value, cnt, priority;
    Node *left, *right;
    Node(int p) : value(p), cnt(1), priority(gen()), left(NULL),
right(NULL) {};
};

typedef Node* pnode;

int get(pnode q){
    if(!q) return 0;
    return q->cnt;
}

void update_cnt(pnode &q){
    if(!q) return;
    q->cnt = get(q->left) + get(q->right) + 1;
}

void merge(pnode &T, pnode lef, pnode rig){
    if(!lef) {T=rig;return;}
    if(!rig){T=lef;return;}
    if(lef->priority > rig->priority){
        merge(lef->right, lef->right, rig);
        T = lef;
    }
    else{
        merge(rig->left, lef, rig->left);
        T = rig;
    }
    update_cnt(T);
}

void split(pnode cur, pnode &lef, pnode &rig, int key){
    if(!cur){
        lef = rig = NULL;
        return;
    }
    int id = get(cur->left) + 1;
    if(id <= key){
        split(cur->right, cur->right, rig, key - id);
        lef = cur;
    }
    else{
        split(cur->left, lef, cur->left, key);
        rig = cur;
    }
    update_cnt(cur);
}
```

### 4.2. Lazy segment tree

```
struct SumSegmentTree{
    vector<ll> S, 0, L;
    void build(ll ti, ll tl, ll tr){
        if(tl==tr){S[ti]=0[tl]; return;}
        build(ti*2, tl, (tl+tr)/2);
        build((ti*2)+1, ((tl+tr)/2)+1, tr);
        S[ti]=S[ti*2]+S[(ti*2)+1];
    }
    void push(ll ti, ll tl, ll tr){
        S[ti] += L[ti]*(tr-tl+1);
        if(tl==tr){L[ti]=0;return;}
        L[ti+ti] += L[ti], L[ti+ti+1] += L[ti];
        L[ti] = 0;
    }
    ll query(ll ti, ll tl, ll tr, ll i, ll j){
        push(ti, tl, tr);
        if(i<=tl&&tr<=j) return S[ti];
        if(tr<i||tl>j) return 0;
        ll a = query(ti*2, tl, (tl+tr)/2, i, j);
        ll b = query((ti*2)+1, ((tl+tr)/2)+1, tr, i, j);
        return a+b;
    }
    void update(ll ti, ll tl, ll tr, ll i, ll j, ll v){
        if(i<=tl&&tr<=j){L[ti]+=v;return;}
        if(tr<i||tl>j) return;
        S[ti]+=v*(i-j+1);
        update(ti*2, tl, (tl+tr)/2, i, j, v);
        update((ti*2)+1, ((tl+tr)/2)+1, tr, i, j, v);
    }
};

ST(vector<ll> &V){
    0 = V;
    S.resize(0.size()*4, 0);
    L.resize(0.size()*4, 0);
    build(1, 0, 0.size()-1);
}

};

4.3. Sparse table

const int N;
const int M; //log2(N)
int sparse[N][M];

void build() {
    for(int i = 0; i < n; i++)
        sparse[i][0] = v[i];

    for(int j = 1; j < M; j++)
        for(int i = 0; i < n; i++)
            sparse[i][j] =
                i + (1 << j - 1) < n
                ? min(sparse[i][j - 1], sparse[i + (1 << j - 1)][j - 1])
                : sparse[i][j - 1];
}

int query(int a, int b){
    int pot = 32 - __builtin_clz(b - a) - 1;
    return min(sparse[a][pot], sparse[b - (1 << pot) + 1][pot]);
}
```

### 4.4. Fenwick tree

```
struct FenwickTree {
    vector<ll> bit; // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    ll sum(int r) {
        ll ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    ll sum(int l, int r) { // l to r of the og array INCLUSIVE
        return sum(r) - sum(l - 1);
    }

    void add(int idx, ll delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};

4.5. Trie

const int K = 26;

struct Vertex {
    int next[K];
    bool output = false;
    Vertex() {fill(begin(next), end(next), -1);}
};

vector<Vertex> t(1); // trie nodes

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back();
        }
        v = t[v].next[c];
    }
    t[v].output = true;
}
```

## 4.6. Aho-Corasick

```
const int K = 26;

struct Vertex {
    int next[K];
    bool output = false;
    int p = -1; // parent node
    char pch; // "transition" character from parent to this node
    int link = -1; // fail link
    int go[K]; // if need more memory can delete this and use "next"

    // additional potentially useful things
    int depth = -1;
    // longest string that has an output from this vertex
    int exitlen = -1;

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);
void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch); // !!!!! ch not c
        }
        v = t[v].next[c];
    }
    t[v].output = true;
}

int go(int v, char ch);
int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            // !!!!! ch not c
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
}
```

```
// int go(int v, char ch) { // go without the go[K] variable
//     int c = ch - 'a';
//     if (t[v].next[c] == -1) {
//         // !!!!! ch not c
//         t[v].next[c] = v == 0 ? 0 : go(get_link(v), ch);
//     }
//     return t[v].next[c];
// }

// helper function
int get_depth(int v){
    if (t[v].depth == -1){
        if (v == 0) {
            t[v].depth = 0;
        } else {
            t[v].depth = get_depth(t[v].p)+1;
        }
    }
    return t[v].depth;
}

// helper function
int get_exitlen(int v){
    if (t[v].exitlen == -1){
        if (v == 0){
            t[v].exitlen = 0;
        } else if (t[v].output) {
            t[v].exitlen = get_depth(v);
        } else {
            t[v].exitlen = get_exitlen(get_link(v));
        }
    }
    return t[v].exitlen;
}

4.7. Disjoint Set Union
struct DSU {
    vector<int> parent, rank;
    DSU(int n) {
        parent.resize(n); rank.resize(n);
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }
    int root(int a) {
        if (parent[a] == a) return a;
        return parent[a] = find(parent[a]);
    }
    void unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return;
        if (rank[a] < rank[b]) {
            parent[a] = b;
        } else if (rank[a] > rank[b]) {
            parent[b] = a;
        } else {
            parent[b] = a;
            rank[a] = rank[a] + 1;
        }
    }
};
```

## 4.8. Merge sort tree

```
struct MergeSortTree {

    int size;
    vector<vector<ll>> values;

    void init(int n){
        size = 1;
        while (size < n){
            size *= 2;
        }
        values.resize(size*2, vl(0));
    }

    void build(vl &arr, int x, int lx, int rx){
        if (rx - lx == 1){
            if (lx < arr.size()){
                values[x].pb(arr[lx]);
            } else {
                values[x].pb(-1);
            }
            return;
        }
        int m = (lx+rx)/2;
        build(arr, 2 * x + 1, lx, m);
        build(arr, 2 * x + 2, m, rx);

        int i = 0;
        int j = 0;
        int asize = values[2*x+1].size();
        while (i < asize && j < asize){
            if (values[2*x+1][i] < values[2*x+2][j]){
                values[x].pb(values[2*x+1][i]);
                i++;
            } else {
                values[x].pb(values[2*x+2][j]);
                j++;
            }
        }
        while (i < asize) {
            values[x].pb(values[2*x+1][i]);
            i++;
        }
        while (j < asize){
            values[x].pb(values[2*x+2][j]);
            j++;
        }
    }

    void build(vl &arr){
        build(arr, 0, 0, size);
    }
};
```

```

int calc(int l, int r, int x, int lx, int rx, int k){
    if (lx >= r || rx <= l) return 0;

    // (elements strictly less than k currently)
    if (lx >= l && rx <= r) { // CHANGE HEURISTIC HERE
        int lft = -1;
        int right = values[x].size();
        while (right - lft > 1){
            int mid = (lft+right)/2;
            if (values[x][mid] < k){
                lft = mid;
            } else {
                right = mid;
            }
        }
        return lft+1;
    }

    int m = (lx+rx)/2;
    int values1 = calc(l, r, 2*x+1, lx, m, k);
    int values2 = calc(l, r, 2*x+2, m, rx, k);
    return values1 + values2;
}

int calc(int l, int r, int k){
    return calc(l, r, 0, 0, size, k);
}
};

```

## Graph algorithms

### 5.1. Bellman-Ford

```

void solve()
{
    vector<int> d(n, INF);
    d[v] = 0;
    for (;;) {
        bool any = false;

        for (Edge e : edges)
            if (d[e.a] < INF)
                if (d[e.b] > d[e.a] + e.cost) {
                    d[e.b] = d[e.a] + e.cost;
                    any = true;
                }

        if (!any)
            break;
    }
    // display d, for example, on the screen
}

```

### 5.2. Dijkstra

```

vector<int> adj[N], adjw[N];
int dist[N];

memset(dist, 63, sizeof(dist));
priority_queue<pii> pq;

```

```

pq.push(mp(0,0));

while (!pq.empty()) {
    int u = pq.top().nd;
    int d = -pq.top().st;
    pq.pop();

    if (d > dist[u]) continue;
    for (int i = 0; i < adj[u].size(); ++i) {
        int v = adj[u][i];
        int w = adjw[u][i];
        if (dist[u] + w < dist[v])
            dist[v] = dist[u]+w, pq.push(mp(-dist[v], v));
    }
}

```

### 5.3. Floyd-Warshall

```

int adj[N][N]; // no-edge = INF

for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);

```

### 5.4. Bridges & articulations

```

// Articulation points and Bridges O(V+E)
int par[N], art[N], low[N], num[N], ch[N], cnt;

void articulation(int u) {
    low[u] = num[u] = ++cnt;
    for (int v : adj[u]) {
        if (!num[v]) {
            par[v] = u; ch[u]++;
            articulation(v);
            if (low[v] >= num[u]) art[u] = 1;
            if (low[v] > num[u]) { /* u-v bridge */ }
            low[u] = min(low[u], low[v]);
        }
        else if (v != par[u]) low[u] = min(low[u], num[v]);
    }
}

```

```

for (int i = 0; i < n; ++i) if (!num[i])
    articulation(i), art[i] = ch[i]>1;

```

### 5.5. Dinic's max flow / matching

Time complexity:

- generally:  $O(EV^2)$
- small flow:  $O(F(V+E))$
- bipartite graph or unit flow:  $O(E\sqrt{V})$

Usage:

- dinic()
- add\_edge(from, to, capacity)
- recover() (optional)

```

const ll N=1e5+5, INF=1e9;
struct edge{ll v, c, f;};

ll src=0, snk=N-1, h[N], ptr[N];
vector<edge> edgs;

vector<ll> g[N];

void add_edge(ll u, ll v, ll c) {
    edgs.push_back({v,c,0}), edgs.push_back({u,0,0});
    ll k=edgs.size();
    g[u].push_back(k), g[v].push_back(k+1);
}

bool bfs() {
    memset(h, 0, sizeof(h));
    queue<ll> q;
    h[src]=1;
    q.push(src);
    while(!q.empty()){
        ll u=q.front();q.pop();
        for(ll i:g[u]){
            ll v=edgs[i].v;
            if(!h[v]&&edgs[i].f<edgs[i].c)
                q.push(v),h[v]=h[u]+1;
        }
    }
    return h[snk];
}

ll dfs(ll u, ll flow){
    if(!flow or u==snk) return flow;
    for(ll &i=ptr[u];i<g[u].size();i++){
        edge &dir=edgs[g[u][i]],&rev=edgs[g[u][i]^1];
        if(h[dir.v]!=h[u]+1) continue;
        ll inc=min(flow,dir.c-dir.f);
        inc=dfs(dir.v,inc);
        if(inc){ dir.f+=inc,rev.f-=inc; return inc;}
    }
    return 0;
}

ll dinic(){
    ll flow=0;
    while(bfs()){
        memset(ptr,0,sizeof(ptr));
        while(ll inc=dfs(src,INF)) flow += inc;
    }
    return flow;
}

vector<pair<ii,ll>> recover() {
    vector<pair<ii,ll>> res;
    for(ll i=0;i<edgs.size();i+=2){
        if(edgs[i].f>0){
            ll v=edgs[i].v, u=edgs[i^1].v;
            res.push_back({u,v,edgs[i].f});
        }
    }
    return res;
}

```

## 5.6. Flow with demands

Finding an arbitrary flow

- Assume a network with  $[L; R]$  on edges (some may have  $L = 0$ ), let's call it old network.
- Create a New Source and New Sink (this will be the src and snk for Dinic).
- Modelling network:
  1. Every edge from the old network will have cost  $R - L$
  2. Add an edge from New Source to every vertex  $v$  with cost:
    - $S(L)$  for every  $(u, v)$ . (sum all  $L$  that LEAVES  $v$ )
  3. Add an edge from every vertex  $v$  to New Sink with cost:
    - $S(L)$  for every  $(v, w)$ . (sum all  $L$  that ARRIVES  $v$ )
  4. Add an edge from Old Source to Old Sink with cost INF (circulation problem)
- The Network will be valid if and only if the flow saturates the network (max flow ==  $S(L)$ )

Finding Min Flow

- To find min flow that satisfies just do a binary search in the (Old Sink -> Old Source) edge
- The cost of this edge represents all the flow from old network
- Min flow =  $S(L)$  that arrives in Old Sink + flow that leaves (Old Sink -> Old Source)

## 5.7. Kosaraju's algorithm

```
const int N = 2e5 + 5;
```

```
vector<int> adj[N], adjt[N];
int n, ordn, scc_cnt, vis[N], ord[N], scc[N];
```

```
//Directed Version
```

```
void dfs(int u) {
    vis[u] = 1;
    for (auto v : adj[u]) if (!vis[v]) dfs(v);
    ord[ordn++] = u;
}
```

```
void dfst(int u) {
    scc[u] = scc_cnt, vis[u] = 0;
    for (auto v : adjt[u]) if (vis[v]) dfst(v);
}
```

```
// add edge: u -> v
void add_edge(int u, int v){
    adj[u].push_back(v);
    adjt[v].push_back(u);
}
```

```
// run kosaraju
void kosaraju(){
    for (int i = 1; i <= n; ++i) if (!vis[i]) dfs(i);
    for (int i = ordn - 1; i >= 0; --i) if (vis[ord[i]]) scc_cnt++, dfst(ord[i]);
}
```

## 5.8. Lowest Common Ancestor

```
const int N = 1e6, M = 25;
int anc[M][N], h[N], rt;
```

```
// TODO: Calculate h[u] and set anc[0][u] = parent of node u for each u
```

```
// build (sparse table)
anc[0][rt] = rt; // set parent of the root to itself
for (int i = 1; i < M; ++i)
    for (int j = 1; j <= n; ++j)
        anc[i][j] = anc[i-1][anc[i-1][j]];
```

```
// query
int lca(int u, int v) {
    if (h[u] < h[v]) swap(u, v);
    for (int i = M-1; i >= 0; --i) if (h[u] - (1<<i) >= h[v])
        u = anc[i][u];
```

```
    if (u == v) return u;
```

```
    for (int i = M-1; i >= 0; --i) if (anc[i][u] != anc[i][v])
        u = anc[i][u], v = anc[i][v];
    return anc[0][u];
}
```

## 5.9. General matching in a graph

```
vector<int> Blossom(vector<vector<int>> graph){
    int n = graph.size();
    int timer = -1;
    vector<int> mate(n, -1), label(n), parent(n),
                orig(n), aux(n, -1), q;
    auto lca = [&](int x, int y) {
        for (timer++; ; swap(x, y)) {
            if (x == -1) continue;
            if (aux[x] == timer) return x;
            aux[x] = timer;
            x = (mate[x] == -1 ? -1 : orig[parent[mate[x]]]);
        }
    };
    auto blossom = [&](int v, int w, int a) {
        while (orig[v] != a) {
            parent[v] = w; w = mate[v];
            if (label[w] == 1) label[w] = 0, q.push_back(w);
            orig[v] = orig[w] = a; v = parent[w];
        }
    };
    auto augment = [&](int v) {
        while (v != -1) {
            int pv = parent[v], nv = mate[pv];
            mate[v] = pv; mate[pv] = v; v = nv;
        }
    };
    auto bfs = [&](int root) {
        fill(label.begin(), label.end(), -1);
        iota(orig.begin(), orig.end(), 0);
        q.clear();
        label[root] = 0; q.push_back(root);
        for (int i = 0; i < (int)q.size(); ++i) {
            int v = q[i];
            for (auto x : graph[v]) {
                if (label[x] == -1) {
                    label[x] = 1; parent[x] = v;
                    if (mate[x] == -1)
                        return augment(x), 1;
                    label[mate[x]] = 0; q.push_back(mate[x]);
                } else if (label[x] == 0 && orig[v] != orig[x]) {
                    int a = lca(orig[v], orig[x]);
                    blossom(x, v, a); blossom(v, x, a);
                }
            }
        }
        return 0;
    };
    for (int i = 0; i < n; i++)
        if (mate[i] == -1)
            bfs(i);
    return mate;
}
```

## String Processing

### 6.1. Knuth-Morris-Pratt (KMP)

```
// Knuth-Morris-Pratt - String Matching O(n+m)
char s[N], p[N];
int b[N], n, m; // n = strlen(s), m = strlen(p);

void kmppre() {
    b[0] = -1;
    for (int i = 0, j = -1; i < m; b[++i] = ++j)
        while (j >= 0 and p[i] != p[j])
            j = b[j];
}
```

```
void kmp() {
    for (int i = 0, j = 0; i < n; i++) {
        while (j >= 0 and s[i] != p[j]) j = b[j];
        i++, j++;
        if (j == m) {
            // match position i-j
            j = b[j];
        }
    }
}
```

## 6.2. Suffix Array

```
// s.push('$');
vector<int> suffix_array(string &s){
    int n = s.size(), alph = 256;
    vector<int> cnt(max(n, alph)), p(n), c(n);
```

```
    for(auto c : s) cnt[c]++;
    for(int i = 1; i < alph; 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++)
        c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
```

```
    vector<int> c2(n), p2(n);
```

```
    for(int k = 0; (1 << k) < n; k++){
        int classes = c[p[n - 1]] + 1;
        fill(cnt.begin(), cnt.begin() + classes, 0);
```

```
        for(int i = 0; i < n; i++) p2[i] = (p[i] - (1 << k) + n)%n;
        for(int i = 0; i < n; i++) cnt[c[i]]++;
        for(int i = 1; i < classes; i++) cnt[i] += cnt[i - 1];
        for(int i = n - 1; i >= 0; i--) p[--cnt[c[p2[i]]]] = p2[i];
```

```
        c2[p[0]] = 0;
        for(int i = 1; i < n; i++){
            pair<int, int> b1 = {c[p[i]], c[(p[i] + (1 << k))%n]};
            pair<int, int> b2 = {c[p[i - 1]], c[(p[i - 1] + (1 << k))%n]};
            c2[p[i]] = c2[p[i - 1]] + (b1 != b2);
        }

        c.swap(c2);
    }
    return p;
}
```

## 6.3. Longest common prefix with SA

```
vector<int> lcp(string &s, vector<int> &p){
    int n = s.size();
    vector<int> ans(n - 1), pi(n);
    for(int i = 0; i < n; i++) pi[p[i]] = i;

    int lst = 0;
    for(int i = 0; i < n - 1; i++){
        if(pi[i] == n - 1) continue;
        while(s[i + lst] == s[p[pi[i] + 1] + lst]) lst++;

        ans[pi[i]] = lst;
        lst = max(0, lst - 1);
    }

    return ans;
}
```

## 6.4. Rabin-Karp

```
// Rabin-Karp - String Matching + Hashing O(n+m)
const int B = 31;
char s[N], p[N];
int n, m; // n = strlen(s), m = strlen(p)
```

```
void rabin() {
    if (n < m) return;
```

```
    ull hp = 0, hs = 0, E = 1;
    for (int i = 0; i < m; ++i)
        hp = ((hp*B)%MOD + p[i])%MOD,
        hs = ((hs*B)%MOD + s[i])%MOD,
        E = (E*B)%MOD;
```

```
    if (hs == hp) { /* matching position 0 */ }
    for (int i = m; i < n; ++i) {
        hs = ((hs*B)%MOD + s[i])%MOD;
        hhs = (hs - s[i-m]*E%MOD + MOD)%MOD;
        if (hs == hp) { /* matching position i-m+1 */ }
    }
}
```

## 6.5. Z-function

The Z-function of a string  $s$  is an array  $z$  where  $z_i$  is the length of the longest substring starting from  $s_i$  which is also a prefix of  $s$ .

Examples:

- “aaaaa”: [0, 4, 3, 2, 1]
- “aaabaab”: [0, 2, 1, 0, 2, 1, 0]
- “abacaba”: [0, 0, 1, 0, 3, 0, 1]

```
vector<int> zfunction(const string& s){
    vector<int> z (s.size());
    for (int i = 1, l = 0, r = 0, n = s.size(); i < n; i++){
        if (i <= r) z[i] = min(z[i-l], r - i + 1);
        while (i + z[i] < n and s[z[i]] == s[i + z[i]]) z[i]++;
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    return z;
}
```

## 6.6. Manacher’s

```
// Manacher (Longest Palindromic String) - O(n)
int lps[2*N+5];
char s[N];

int manacher() {
    int n = strlen(s);

    string p (2*n+3, '#');
    p[0] = '^';
    for (int i = 0; i < n; i++) p[2*(i+1)] = s[i];
    p[2*n+2] = '$';

    int k = 0, r = 0, m = 0;
    int l = p.length();
    for (int i = 1; i < l; i++) {
        int o = 2*k - i;
        lps[i] = (r > i) ? min(r-i, lps[o]) : 0;
        while (p[i + 1 + lps[i]] == p[i - 1 - lps[i]]) lps[i]++;
        if (i + lps[i] > r) k = i, r = i + lps[i];
        m = max(m, lps[i]);
    }
    return m;
}
```

# Dynamic programming

## 7.1. Convex hull trick

// Convex Hull Trick

// ATTENTION: This is the maximum convex hull. If you need the minimum  
// CHT use {-b, -m} and modify the query function.

// In case of floating point parameters swap long long with long double  
typedef long long type;  
struct line { type b, m; };

```
line v[N]; // lines from input
int n; // number of lines
// Sort slopes in ascending order (in main):
sort(v, v+n, [](line s, line t){
    return (s.m == t.m) ? (s.b < t.b) : (s.m < t.m); });
```

```
// nh: number of lines on convex hull
// pos: position for linear time search
// hull: lines in the convex hull
int nh, pos;
line hull[N];
```

```
bool check(line s, line t, line u) {
    // verify if it can overflow. If it can just divide using long double
    return (s.b - t.b)*(u.m - s.m) < (s.b - u.b)*(t.m - s.m);
}
```



```

// Add new line to convex hull, if possible
// Must receive lines in the correct order, otherwise it won't work
void update(line s) {
    // 1. if first lines have the same b, get the one with bigger m
    // 2. if line is parallel to the one at the top, ignore
    // 3. pop lines that are worse
    // 3.1 if you can do a linear time search, use
    // 4. add new line

    if (nh == 1 and hull[nh-1].b == s.b) nh--;
    if (nh > 0 and hull[nh-1].m >= s.m) return;
    while (nh >= 2 and !check(hull[nh-2], hull[nh-1], s)) nh--;
    pos = min(pos, nh);
    hull[nh++] = s;
}

type eval(int id, type x) { return hull[id].b + hull[id].m * x; }

// Linear search query - O(n) for all queries
// Only possible if the queries always move to the right
type query(type x) {
    while (pos+1 < nh and eval(pos, x) < eval(pos+1, x)) pos++;
    return eval(pos, x);
    // return -eval(pos, x);    ATTENTION: Uncomment for minimum CHT
}

```

## 7.2. Longest Increasing Subsequence

```

memset(dp, 63, sizeof dp);
for (int i = 0; i < n; ++i) {
    // increasing: lower_bound
    // non-decreasing: upper_bound
    int j = lower_bound(dp, dp + lis, v[i]) - dp;
    dp[j] = min(dp[j], v[i]);
    lis = max(lis, j + 1);
}

```

## 7.3. SOS DP (Sum over Subsets)

// O(bits\*(2^bits))

```
const int bits = 20;
```

```
vector<int> a(1<<bits); // initial value of each subset
vector<int> f(1<<bits); // sum over all subsets
// (at f[011] = a[011]+a[001]+a[010]+a[000])
```

```

for (int i = 0; i<(1<<bits); i++){
    f[i] = a[i];
}
for (int i = 0; i < bits; i++) {
    for(int mask = 0; mask < (1<<bits); mask++){
        if(mask & (1<<i)){
            f[mask] += f[mask^(1<<i)];
        }
    }
}
}

```