ALGORITHMS AND DATA STRUCTURES FOR COMPETITIVE PROGRAMMING

Contents

1. Code Templates 2
1.1. Basic Configuration 2
1.2. C++ Header 2
1.3. Java Template 2
2. Data Structures 2
2.1. Union-Find 2
2.2. Segment Tree 3
2.3. Fenwick Tree 3
2.4. Matrix 3
2.5. Trie 4
2.6. AVL Tree 5
2.7. Heap 6
2.8. Skiplist 7
3. Graphs 8
3.1. Breadth-First Search 8
3.2. Single-Source Shortest Paths 9
3.3. All-Pairs Shortest Paths 9
3.4. Strongly Connected Components 10
3.5. Minimum Spanning Tree 10
3.6. Topological Sort 10
3.7. Bipartite Matching 11
3.8. Maximum Flow 11
3.9. Minimum Cost Maximum Flow 12
4. Strings 13
4.1. The Z algorithm 13
5. Mathematics 14
5.1. Fraction 14
5.2. Binomial Coefficients 14
5.3. Euclidean algorithm 14
5.4. Trial Division Primality Testing 14
5.5. Sieve of Eratosthenes 15
5.6. Modular Multiplicative Inverse 15
5.7. Modular Exponentiation 15
5.8. Chinese Remainder Theorem 15
5.9. Formulas 15
6. Geometry 16
6.1. Primitives 16
6.2. Convex Hull 16
6.3. Formulas 17
7. Other Algorithms 17
7.1. Binary Search 17
7.2. 2SAT 17
7.3. nth Permutation 18
7.4. Cycle-Finding 18
7.5. Dates 18
8. Useful Information 18
8.1. Tips & Tricks 18
8.2. Fast Input Reading 19
8.3. Worst Time Complexity 19
8.4. Bit Hacks 19

1. Code Templates

1.1. **Basic Configuration.** Vim and (Caps Lock = Escape) configuration.

```
xset r rate 150 100
xmodmap -e "remove Lock = Caps_Lock" -e "keysym Caps_Lock = Escape" -e "add Lock = Caps_Lock"
echo "set nocp et sw=4 ts=4 sts=4 si cindent ru noeb showcmd showmode | syn on | colorscheme slate" > ~/.vimrc
```

1.2. **C++ Header.** A C++ header.

```cpp
#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <iomanip>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <string>
#include <utility>
#include <vector>
using namespace std;

#define all(o) (o).begin(), (o).end()
#define allr(o) (o).rbegin(), (o).rend()
#define pb push_back
const int INF = 2147483647;
const double EPS = 1e-9;
const double pi = acos(-1);
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<vi> vvi;
typedef vector<vii> vvii;
template <class T> T mod(T a, T b) { return (a % b + b) % b; }
template <class T> int size(T &x) { return x.size(); }
```

1.3. **Java Template.** A Java template.

```java
import java.util.*;
import java.math.*;
import java.io.*;

public class Main {
    public static void main(String[] args) throws Exception {
        Scanner in = new Scanner(System.in);
        PrintWriter out = new PrintWriter(System.out, false);
        // code
        out.flush();
    }
}
```

2. Data Structures

2.1. **Union-Find.** An implementation of the Union-Find disjoint sets data structure.

```cpp
struct union_find {
    vi parent;
    int cnt;
    union_find(int n) { parent.resize(cnt = n); for (int i = 0; i < cnt; i++) parent[i] = i; }
    int find(int i) { return parent[i] == i ? i : (parent[i] = find(parent[i])); }
    bool unite(int i, int j) {
        int ip = find(i), jp = find(j);
        parent[ip] = jp; return ip != jp; } };
```

2.2. **Segment Tree.** An implementation of a Segment Tree.

```
// const int ID = INF;
// int f(int a, int b) { return min(a, b); }
const int ID = 0;
int f(int a, int b) { return a + b; }
struct segment_tree {
    int n; vi data;
    segment_tree(const vi &arr) : n(size(arr)), data(4*n) { mk(arr, 0, n-1, 0); }
    int mk(const vi &arr, int l, int r, int i) {
        if (l == r) return data[i] = arr[l];
        int m = (l + r) / 2;
        return data[i] = f(mk(arr, l, m, 2*i+1), mk(arr, m+1, r, 2*i+2)); }
    int query(int a, int b) { return q(a, b, 0, n-1, 0); }
    int q(int a, int b, int l, int r, int i) {
        if (r < a || b < l) return ID;
        if (a <= l && r <= b) return data[i];
        int m = (l + r) / 2;
        return f(q(a, b, l, m, 2*i+1), q(a, b, m+1, r, 2*i+2)); }
    int update(int i, int v) { u(i, v, 0, n-1, 0); }
    int u(int i, int v, int l, int r, int j) {
        if (r < i || i < l) return data[j];
        if (l == i && r == i) return data[j] = v;
        int m = (l + r) / 2;
        return data[j] = f(u(i, v, l, m, 2*j+1), u(i, v, m+1, r, 2*j+2)); } };
```

2.3. **Fenwick Tree.** A Fenwick Tree is a data structure that represents an array of n numbers. It supports adjusting the i-th element in $O(\log n)$ time, and computing the sum of numbers in the range $i..j$ in $O(\log n)$ time. It only needs $O(n)$ space.

```
struct fenwick_tree {
    int n; vi data;
    fenwick_tree(int _n) : n(_n), data(vi(n)) { }
    void update(int at, int by) { while (at < n) data[at] += by, at |= at + 1; }
    int query(int at) {
        int res = 0;
        while (at >= 0) res += data[at], at = (at & (at + 1)) - 1;
        return res; }
    int rsq(int a, int b) { return query(b) - query(a - 1); }
};
struct fenwick_tree_sq {
    int n; fenwick_tree x1, x0;
    fenwick_tree_sq(int _n) : n(_n), x1(fenwick_tree(n)), x0(fenwick_tree(n)) { }
    // insert f(y) = my + c if x <= y
    void update(int x, int m, int c) { x1.update(x, m); x0.update(x, c); }
    int query(int x) { return x*x1.query(x) + x0.query(x); }
};
void range_update(fenwick_tree_sq &s, int a, int b, int k) {
    s.update(a, k, k * (1 - a)); s.update(b+1, -k, k * b); }
int range_query(fenwick_tree_sq &s, int a, int b) {
    return s.query(b) - s.query(a-1); }
```

2.4. **Matrix.** A Matrix class.

```
template <class K> bool eq(K a, K b) { return a == b; }
template <> bool eq<double>(double a, double b) { return abs(a - b) < EPS; }
template <class T>
class matrix {
private:
    vector<T> data;
    int cnt;
    inline T& at(int i, int j) { return data[i * cols + j]; }
public:
    int rows, cols;
    matrix(int r, int c) : rows(r), cols(c), cnt(r * c) { data.assign(cnt, T(0)); }
    matrix(const matrix& other) : rows(other.rows), cols(other.cols), cnt(other.cnt), data(other.data) { }
    T& operator()(int i, int j) { return at(i, j); }
    void operator +=(const matrix& other) { for (int i = 0; i < cnt; i++) data[i] += other.data[i]; }
    void operator -=(const matrix& other) { for (int i = 0; i < cnt; i++) data[i] -= other.data[i]; }
    void operator *=(T other) { for (int i = 0; i < cnt; i++) data[i] *= other; }
    matrix<T> operator +(const matrix& other) { matrix<T> res(*this); res += other; return res; }
    matrix<T> operator -(const matrix& other) { matrix<T> res(*this); res -= other; return res; }
    matrix<T> operator *(T other) { matrix<T> res(*this); res *= other; return res; }
    matrix<T> operator *(const matrix& other) {
        matrix<T> res(rows, other.cols);
        for (int i = 0; i < rows; i++) for (int j = 0; j < other.cols; j++)
            for (int k = 0; k < cols; k++) res(i, j) += at(i, k) * other.data[k * other.cols + j];
        return res; }
    matrix<T> transpose() {
        matrix<T> res(cols, rows);
        for (int i = 0; i < rows; i++) for (int j = 0; j < cols; j++) res(j, i) = at(i, j);
        return res; }
    matrix<T> pow(int p) {
        matrix<T> res(rows, cols), sq(*this);
        for (int i = 0; i < rows; i++) res(i, i) = T(1);
        while (p) {
            if (p & 1) res = res * sq;
            p >>= 1;
            if (p) sq = sq * sq;
        } return res; }
    matrix<T> rref() {
        matrix<T> mat(*this);
        for (int r = 0, c = 0; c < cols; c++) {
            int k = r;
            while (k < rows && eq<T>(mat(k, c), T(0))) k++;
            if (k >= rows) continue;
            if (k != r) for (int i = 0; i < cols; i++) swap(mat.at(k, i), mat.at(r, i));
            if (!eq<T>(mat(r, c), T(1))) for (int i = cols-1; i >= c; i--) mat(r, i) /= mat(r, c);
            for (int i = 0; i < rows; i++) {
                T m = mat(i, c);
                if (i != r && !eq<T>(m, T(0))) for (int j = 0; j < cols; j++) mat(i, j) -= m * mat(r, j);
            } r++;
        } return mat; }
};
```

2.5. **Trie.** A Trie class.

```
template <class T>
class trie {
private:
    struct node {
        map<T, node*> children;
        int prefixes, words;
        node() { prefixes = words = 0; } };
public:
    node* root;
    trie() : root(new node()) {  }
    template <class I>
    void insert(I begin, I end) {
        node* cur = root;
        while (true) {
            cur->prefixes++;
            if (begin == end) { cur->words++; break; }
            else {
                T head = *begin;
                typename map<T, node*>::const_iterator it = cur->children.find(head);
                if (it == cur->children.end()) it = cur->children.insert(pair<T, node*>(head, new node())).first;
                begin++, cur = it->second; } } }
    template<class I>
    int countMatches(I begin, I end) {
        node* cur = root;
        while (true) {
            if (begin == end) return cur->words;
            else {
                T head = *begin;
                typename map<T, node*>::const_iterator it = cur->children.find(head);
                if (it == cur->children.end()) return 0;
                begin++, cur = it->second; } } }
    template<class I>
    int countPrefixes(I begin, I end) {
        node* cur = root;
        while (true) {
            if (begin == end) return cur->prefixes;
            else {
                T head = *begin;
                typename map<T, node*>::const_iterator it = cur->children.find(head);
                if (it == cur->children.end()) return 0;
                begin++, cur = it->second; } } } };
```

2.6. **AVL Tree.** A fast, easily augmentable, balanced binary search tree.

```
#define AVL_MULTISET 0

template <class T>
class avl_tree {
public:
    struct node {
        T item; node *p, *l, *r;
        int size, height;
        node(const T &item, node *p = NULL) : item(item), p(p), l(NULL), r(NULL), size(1), height(0) { } };
    avl_tree() : root(NULL) { }
    node *root;
    node* find(const T &item) const {
        node *cur = root;
        while (cur) {
            if (cur->item < item) cur = cur->r;
            else if (item < cur->item) cur = cur->l;
            else break; }
        return cur; }
    node* insert(const T &item) {
        node *prev = NULL, **cur = &root;
        while (*cur) {
            prev = *cur;
            if ((*cur)->item < item) cur = &((*cur)->r);
#if AVL_MULTISET
            else cur = &((*cur)->l);
#else
            else if (item < (*cur)->item) cur = &((*cur)->l);
            else return *cur;
#endif
        }
        node *n = new node(item, prev);
        *cur = n, fix(n); return n; }
    void erase(const T &item) { erase(find(item)); }
    void erase(node *n, bool free = true) {
        if (!n) return;
        if (!n->l && n->r) parent_leg(n) = n->r, n->r->p = n->p;
        else if (n->l && !n->r) parent_leg(n) = n->l, n->l->p = n->p;
        else if (n->l && n->r) {
            node *s = successor(n);
            erase(s, false);
            s->p = n->p, s->l = n->l, s->r = n->r;
            if (n->l) n->l->p = s;
            if (n->r) n->r->p = s;
            parent_leg(n) = s, fix(s);
            return;
        } else parent_leg(n) = NULL;
        fix(n->p), n->p = n->l = n->r = NULL;
        if (free) delete n; }
    node* successor(node *n) const {
        if (!n) return NULL;
        if (n->r) return nth(0, n->r);
        node *p = n->p;
        while (p && p->r == n) n = p, p = p->p;
        return p; }
    node* predecessor(node *n) const {
        if (!n) return NULL;
        if (n->l) return nth(n->l->size-1, n->l);
        node *p = n->p;
        while (p && p->l == n) n = p, p = p->p;
        return p; }
    inline int size() const { return sz(root); }
    void clear() { delete_tree(root), root = NULL; }
    node* nth(int n, node *cur = NULL) const {
        if (!cur) cur = root;
        while (cur) {
            if (n < sz(cur->l)) cur = cur->l;
            else if (n > sz(cur->l)) n -= sz(cur->l) + 1, cur = cur->r;
            else break;
        } return cur; }
private:
    inline int sz(node *n) const { return n ? n->size : 0; }
    inline int height(node *n) const { return n ? n->height : -1; }
    inline bool left_heavy(node *n) const { return n && height(n->l) > height(n->r); }
    inline bool right_heavy(node *n) const { return n && height(n->r) > height(n->l); }
    inline bool too_heavy(node *n) const { return n && abs(height(n->l) - height(n->r)) > 1; }
    void delete_tree(node *n) { if (n) { delete_tree(n->l), delete_tree(n->r); delete n; } }
    node*& parent_leg(node *n) {
        if (!n->p) return root;
        if (n->p->l == n) return n->p->l;
        if (n->p->r == n) return n->p->r;
        assert(false); }
    void augment(node *n) {
        if (!n) return;
        n->size = 1 + sz(n->l) + sz(n->r);
        n->height = 1 + max(height(n->l), height(n->r)); }
    #define rotate(l, r) \
        node *l = n->l; \
        l->p = n->p; \
        parent_leg(n) = l; \
        n->l = l->r; \
        if (l->r) l->r->p = n; \
        l->r = n, n->p = l; \
        augment(n), augment(l)
    void left_rotate(node *n) { rotate(r, l); }
    void right_rotate(node *n) { rotate(l, r); }
    void fix(node *n) {
        while (n) { augment(n);
            if (too_heavy(n)) {
                if (left_heavy(n) && right_heavy(n->l)) left_rotate(n->l);
                else if (right_heavy(n) && left_heavy(n->r)) right_rotate(n->r);
                if (left_heavy(n)) right_rotate(n);
                else left_rotate(n);
                n = n->p; }
            n = n->p; } } };
```

Also a very simple wrapper over the AVL tree that implements a map interface.

```
#include "avl_tree.cpp"

template <class K, class V>
class avl_map {
public:
    struct node {
        K key; V value;
        node(K k, V v) : key(k), value(v) { }
        bool operator <(const node &other) const { return key < other.key; } };
    avl_tree<node> tree;
    V& operator [](K key) {
        typename avl_tree<node>::node *n = tree.find(node(key, V(0)));
        if (!n) n = tree.insert(node(key, V(0)));
        return n->item.value;
    }
};
```

2.7. **Heap.** An implementation of a binary heap.

```
#define RESIZE
#define SWP(x,y) tmp = x, x = y, y = tmp
struct default_int_cmp {
    default_int_cmp() { }
    bool operator ()(const int &a, const int &b) { return a < b; } };
template <class Compare = default_int_cmp>
class heap {
```

```cpp
private:
    int len, count, *q, *loc, tmp;
    Compare _cmp;
    inline bool cmp(int i, int j) { return _cmp(q[i], q[j]); }
    inline void swp(int i, int j) { SWP(q[i], q[j]), SWP(loc[q[i]], loc[q[j]]); }
    void swim(int i) {
        while (i > 0) {
            int p = (i - 1) / 2;
            if (!cmp(i, p)) break;
            swp(i, p), i = p; } }
    void sink(int i) {
        while (true) {
            int l = 2*i + 1, r = l + 1;
            if (l >= count) break;
            int m = r >= count || cmp(l, r) ? l : r;
            if (!cmp(m, i)) break;
            swp(m, i), i = m; } }
public:
    heap(int init_len = 128) : count(0), len(init_len), _cmp(Compare()) {
        q = new int[len], loc = new int[len];
        memset(loc, 255, len << 2); }
    ~heap() { delete[] q; delete[] loc; }
    void push(int n, bool fix = true) {
        if (len == count || n >= len) {
#ifdef RESIZE
            int newlen = 2 * len;
            while (n >= newlen) newlen *= 2;
            int *newq = new int[newlen], *newloc = new int[newlen];
            for (int i = 0; i < len; i++) newq[i] = q[i], newloc[i] = loc[i];
            memset(newloc + len, 255, (newlen - len) << 2);
            delete[] q, delete[] loc;
            loc = newloc, q = newq, len = newlen;
#else
            assert(false);
#endif
        }
        assert(loc[n] == -1);
        loc[n] = count, q[count++] = n;
        if (fix) swim(count-1); }
    void pop(bool fix = true) {
        assert(count > 0);
        loc[q[0]] = -1, q[0] = q[--count], loc[q[0]] = 0;
        if (fix) sink(0);
    }
    int top() { assert(count > 0); return q[0]; }
    void heapify() { for (int i = count - 1; i > 0; i--) if (cmp(i, (i - 1) / 2)) swp(i, (i - 1) / 2); }
    void update_key(int n) { assert(loc[n] != -1), swim(loc[n]), sink(loc[n]); }
    bool empty() { return count == 0; }
    int size() { return count; }
    void clear() { count = 0, memset(loc, 255, len << 2); } };
```

2.8. **Skiplist.** An implementation of a skiplist.

```cpp
#define BP 0.20
#define MAX_LEVEL 10
unsigned int bernoulli(unsigned int MAX) {
    unsigned int cnt = 0;
    while(((float) rand() / RAND_MAX) < BP && cnt < MAX) cnt++;
    return cnt; }
template<class T> class skiplist {
public:
    struct node {
        T item;
        int *lens;
        node **next;
        #define CA(v, t) v((t*)calloc(level+1, sizeof(t)))
        node(int level, T i) : item(i), CA(lens, int), CA(next, node*) {}
        ~node() { free(lens); free(next); }; };
    int current_level, _size;
    node *head;
    skiplist() : current_level(0), _size(0), head(new node(MAX_LEVEL, 0)) { };
    ~skiplist() { clear(); delete head; head = NULL; }
    #define FIND_UPDATE(cmp, target) \
        int pos[MAX_LEVEL + 2]; \
        memset(pos, 0, sizeof(pos)); \
        node *x = head; \
        node *update[MAX_LEVEL + 1]; \
        memset(update, 0, MAX_LEVEL + 1); \
        for(int i = MAX_LEVEL; i >= 0; i--) { \
            pos[i] = pos[i + 1]; \
            while(x->next[i] != NULL && cmp < target) { pos[i] += x->lens[i]; x = x->next[i]; } \
            update[i] = x; \
        } x = x->next[0];
    int size() { return _size; }
    void clear() { while(head->next && head->next[0]) erase(head->next[0]->item); }
    node *find(T target) { FIND_UPDATE(x->next[i]->item, target); return x && x->item == target ? x : NULL; }
    node *nth(int k) { FIND_UPDATE(pos[i] + x->lens[i], k+1); return x; }
    int count_less(T target) { FIND_UPDATE(x->next[i]->item, target); return pos[0]; }
    node* insert(T target) {
        FIND_UPDATE(x->next[i]->item, target);
        if(x && x->item == target) return x; // SET
        int lvl = bernoulli(MAX_LEVEL);
        if(lvl > current_level) current_level = lvl;
        x = new node(lvl, target);
        for(int i = 0; i <= lvl; i++) {
            x->next[i] = update[i]->next[i];
            x->lens[i] = pos[i] + update[i]->lens[i] - pos[0];
            update[i]->next[i] = x;
            update[i]->lens[i] = pos[0] + 1 - pos[i];
        }
        for(int i = lvl + 1; i <= MAX_LEVEL; i++) update[i]->lens[i]++;
        _size++;
        return x; }
    void erase(T target) {
        FIND_UPDATE(x->next[i]->item, target);
        if(x && x->item == target) {
            for(int i = 0; i <= current_level; i++) {
                if(update[i]->next[i] == x) {
                    update[i]->next[i] = x->next[i];
                    update[i]->lens[i] = update[i]->lens[i] + x->lens[i] - 1;
                } else update[i]->lens[i] = update[i]->lens[i] - 1;
            }
            delete x; _size--;
            while(current_level > 0 && head->next[current_level] == NULL) current_level--; } } };
```

3. Graphs

3.1. Breadth-First Search. An implementation of a breadth-first search that counts the number of edges on the shortest path from the starting vertex to the ending vertex in the specified unweighted graph (which is represented with an adjacency list). Note that it assumes that the two vertices are connected. It runs in $O(|V| + |E|)$ time.

```
int bfs(int start, int end, vvi& adj_list) {
    queue<ii> Q;
    Q.push(ii(start, 0));

    while (true) {
        ii cur = Q.front(); Q.pop();

        if (cur.first == end)
            return cur.second;

        vi& adj = adj_list[cur.first];
        for (vi::iterator it = adj.begin(); it != adj.end(); it++)
            Q.push(ii(*it, cur.second + 1));
    }
}
```

Another implementation that doesn't assume the two vertices are connected. If there is no path from the starting vertex to the ending vertex, a -1 is returned.

```
int bfs(int start, int end, vvi& adj_list) {
    set<int> visited;
    queue<ii> Q;
    Q.push(ii(start, 0));
    visited.insert(start);

    while (!Q.empty()) {
        ii cur = Q.front(); Q.pop();

        if (cur.first == end)
            return cur.second;

        vi& adj = adj_list[cur.first];
        for (vi::iterator it = adj.begin(); it != adj.end(); it++)
            if (visited.find(*it) == visited.end()) {
                Q.push(ii(*it, cur.second + 1));
                visited.insert(*it);
            }
    }

    return -1;
}
```

3.2. **Single-Source Shortest Paths.**

3.2.1. *Dijkstra's algorithm.* An implementation of Dijkstra's algorithm. It runs in $\Theta(|E| \log |V|)$ time.

```
int *dist, *dad;
struct cmp {
    bool operator()(int a, int b) { return dist[a] != dist[b] ? dist[a] < dist[b] : a < b; }
};

pair<int*, int*> dijkstra(int n, int s, vii *adj) {
    dist = new int[n];
    dad = new int[n];
    for (int i = 0; i < n; i++) dist[i] = INF, dad[i] = -1;
    set<int, cmp> pq;
    dist[s] = 0, pq.insert(s);
    while (!pq.empty()) {
        int cur = *pq.begin(); pq.erase(pq.begin());
        for (int i = 0, cnt = size(adj[cur]); i < cnt; i++) {
            int nxt = adj[cur][i].first, ndist = dist[cur] + adj[cur][i].second;
            if (ndist < dist[nxt]) pq.erase(nxt), dist[nxt] = ndist, dad[nxt] = cur, pq.insert(nxt);
        }
    }

    return pair<int*, int*>(dist, dad);
}
```

3.2.2. *Bellman-Ford algorithm.* The Bellman-Ford algorithm solves the single-source shortest paths problem in $O(|V||E|)$ time. It is slower than Dijkstra's algorithm, but it works on graphs with negative edges and has the ability to detect negative cycles, neither of which Dijkstra's algorithm can do.

```
int* bellman_ford(int n, int s, vii* adj, bool& has_negative_cycle) {
    has_negative_cycle = false;
    int* dist = new int[n];
    for (int i = 0; i < n; i++) dist[i] = i == s ? 0 : INF;
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n; j++)
            if (dist[j] != INF)
                for (int k = 0, len = size(adj[j]); k < len; k++)
                    dist[adj[j][k].first] = min(dist[adj[j][k].first], dist[j] + adj[j][k].second);
    for (int j = 0; j < n; j++)
        for (int k = 0, len = size(adj[j]); k < len; k++)
            if (dist[j] + adj[j][k].second < dist[adj[j][k].first])
                has_negative_cycle = true;
    return dist;
}
```

3.3. **All-Pairs Shortest Paths.**

3.3.1. *Floyd-Warshall algorithm.* The Floyd-Warshall algorithm solves the all-pairs shortest paths problem in $O(|V|^3)$ time.

```
void floyd_warshall(int** arr, int n) {
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (arr[i][k] != INF && arr[k][j] != INF)
                    arr[i][j] = min(arr[i][j], arr[i][k] + arr[k][j]);
}
```

3.4. Strongly Connected Components.

3.4.1. *Kosaraju's algorithm.* Kosarajus's algorithm finds strongly connected components of a directed graph in $O(|V|+|E|)$ time.

```
#include "../data-structures/union_find.cpp"

vector<bool> visited;
vi order;

void scc_dfs(const vvi &adj, int u) {
    int v; visited[u] = true;
    for (int i = 0; i < size(adj[u]); i++) if (!visited[v = adj[u][i]]) scc_dfs(adj, v);
    order.push_back(u);
}

pair<union_find, vi> scc(const vvi &adj) {
    int n = size(adj), u, v;
    order.clear();
    union_find uf(n);
    vi dag;
    vvi rev(n);
    for (int i = 0; i < n; i++) for (int j = 0; j < size(adj[i]); j++) rev[adj[i][j]].push_back(i);
    visited.resize(n), fill(all(visited), false);
    for (int i = 0; i < n; i++) if (!visited[i]) scc_dfs(rev, i);
    fill(all(visited), false);
    stack<int> S;
    for (int i = n-1; i >= 0; i--) {
        if (visited[order[i]]) continue;
        S.push(order[i]), dag.push_back(order[i]);
        while (!S.empty()) {
            visited[u = S.top()] = true, S.pop(), uf.unite(u, order[i]);
            for (int i = 0; i < size(adj[u]); i++) if (!visited[v = adj[u][i]]) S.push(v);
        }
    }
    return pair<union_find, vi>(uf, dag);
}
```

3.5. Minimum Spanning Tree.

3.5.1. *Kruskal's algorithm.*

```
#include "../data-structures/union_find.cpp"

// n is the number of vertices
// edges is a list of edges of the form (weight, (a, b))
// the edges in the minimum spanning tree are returned on the same form
vector<pair<int, ii> > mst(int n, vector<pair<int, ii> > edges) {
    union_find uf(n);
    sort(all(edges));
    vector<pair<int, ii> > res;
    for (int i = 0, cnt = size(edges); i < cnt; i++)
        if (uf.find(edges[i].second.first) != uf.find(edges[i].second.second)) {
            res.push_back(edges[i]);
            uf.unite(edges[i].second.first, edges[i].second.second);
        }
    return res;
}
```

3.6. Topological Sort.

3.6.1. *Modified Depth-First Search.*

```
void tsort_dfs(int cur, char* color, const vvi& adj, stack<int>& res, bool& has_cycle) {
    color[cur] = 1;
    for (int i = 0, cnt = size(adj[cur]); i < cnt; i++) {
        int nxt = adj[cur][i];
        if (color[nxt] == 0)
            tsort_dfs(nxt, color, adj, res, has_cycle);
        else if (color[nxt] == 1)
            has_cycle = true;
        if (has_cycle) return;
    }
    color[cur] = 2;
    res.push(cur);
}

vi tsort(int n, vvi adj, bool& has_cycle) {
    has_cycle = false;
    stack<int> S;
    vi res;
    char* color = new char[n];
    memset(color, 0, n);
    for (int i = 0; i < n; i++) {
        if (!color[i]) {
            tsort_dfs(i, color, adj, S, has_cycle);
            if (has_cycle) return res;
        }
    }
    while (!S.empty()) res.push_back(S.top()), S.pop();
    return res;
}
```

3.7. **Bipartite Matching.** The alternating paths algorithm solves bipartite matching in $O(mn^2)$ time, where m, n are the number of vertices on the left and right side of the bipartite graph, respectively.

```
vi* adj;
bool* done;
int* owner;
int alternating_path(int left) {
    if (done[left]) return 0;
    done[left] = true;
    for (int i = 0; i < size(adj[left]); i++) {
        int right = adj[left][i];
        if (owner[right] == -1 || alternating_path(owner[right])) {
            owner[right] = left; return 1;
        } }
    return 0; }
```

3.8. **Maximum Flow.**

3.8.1. *Edmonds Karp's algorithm.* An implementation of Edmonds Karp's algorithm that runs in $O(|V||E|^2)$. It computes the maximum flow of a flow network.

```
struct mf_edge {
    int u, v, w;
    mf_edge* rev;
    mf_edge(int _u, int _v, int _w, mf_edge* _rev = NULL) {
        u = _u; v = _v; w = _w; rev = _rev;
    }
};
int max_flow(int n, int s, int t, vii* adj) {
    vector<mf_edge*>* g = new vector<mf_edge*>[n];
    vector<mf_edge*> t_prev;
    for (int i = 0; i < n; i++) {
        for (int j = 0, len = size(adj[i]); j < len; j++) {
            mf_edge *cur = new mf_edge(i, adj[i][j].first, adj[i][j].second),
                    *rev = new mf_edge(adj[i][j].first, i, 0, cur);
            cur->rev = rev;
            g[i].push_back(cur);
            g[cur->v].push_back(rev);
            if(cur->v == t && cur->w > 0) t_prev.push_back(cur);
        }
    }
    int flow = 0;
    mf_edge** back = new mf_edge*[n];
    while (true) {
        for (int i = 0; i < n; i++) back[i] = NULL;
        queue<int> Q; Q.push(s);
        while (!Q.empty()) {
            int cur = Q.front(); Q.pop();
            if (cur == t) break;
            for (int i = 0, len = size(g[cur]); i < len; i++) {
                mf_edge* nxt = g[cur][i];
                if (nxt->v != s && nxt->w > 0 && back[nxt->v] == NULL) {
                    back[nxt->v] = nxt;
                    Q.push(nxt->v);
                }
            }
        }
        if(back[t] == NULL || back[t]->w == 0) break;
        for(int i = 0; i < t_prev.size(); ++i) {
            mf_edge *z = t_prev[i];
            if(!z || z->w == 0 || back[z->u] == NULL) continue;
            int cap = z->w;
            mf_edge* cure = back[z->u];
            if (cure == NULL) break;
            while (true) {
                cap = min(cap, cure->w);
                if (cure->u == s) break;
                cure = back[cure->u];
            }
            if(cap == 0) continue;
            assert(cap < INF);
            z->w -= cap;
            z->rev->w += cap;
            cure = back[z->u];
            while (true) {
                cure->w -= cap;
                cure->rev->w += cap;
                if (cure->u == s) break;
                cure = back[cure->u];
            }
            flow += cap;
        }
    }
    // instead of deleting g, we could also
    // use it to get info about the actual flow
    for (int i = 0; i < n; i++)
        for (int j = 0, len = size(g[i]); j < len; j++)
            delete g[i][j];
    delete[] g;
    delete[] back;
    return flow;
}
```

3.9. **Minimum Cost Maximum Flow.** An implementation of Edmonds Karp's algorithm, modified to find shortest path to augment each time (instead of just any path). It computes the maximum flow of a flow network, and when there are multiple maximum flows, finds the maximum flow with minimum cost.

```
struct mcmf_edge {
    int u, v, w, c;
    mcmf_edge* rev;
    mcmf_edge(int _u, int _v, int _w, int _c, mcmf_edge* _rev = NULL) {
        u = _u; v = _v; w = _w; c = _c; rev = _rev;
    }
};

ii min_cost_max_flow(int n, int s, int t, vector<pair<int, ii> >* adj) {
    vector<mcmf_edge*>* g = new vector<mcmf_edge*>[n];
    for (int i = 0; i < n; i++) {
        for (int j = 0, len = size(adj[i]); j < len; j++) {
            mcmf_edge *cur = new mcmf_edge(i, adj[i][j].first, adj[i][j].second.first, adj[i][j].second.second),
                      *rev = new mcmf_edge(adj[i][j].first, i, 0, -adj[i][j].second.second, cur);
            cur->rev = rev;
            g[i].push_back(cur);
            g[adj[i][j].first].push_back(rev);
        }
    }
    int flow = 0, cost = 0;
    mcmf_edge** back = new mcmf_edge*[n];
    int* dist = new int[n];
    while (true) {
        for (int i = 0; i < n; i++) back[i] = NULL, dist[i] = INF;
        dist[s] = 0;
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n; j++)
                if (dist[j] != INF)
                    for (int k = 0, len = size(g[j]); k < len; k++)
                        if (g[j][k]->w > 0 && dist[j] + g[j][k]->c < dist[g[j][k]->v]) {
                            dist[g[j][k]->v] = dist[j] + g[j][k]->c;
                            back[g[j][k]->v] = g[j][k];
                        }
        mcmf_edge* cure = back[t];
        if (cure == NULL) break;
        int cap = INF;
        while (true) {
            cap = min(cap, cure->w);
            if (cure->u == s) break;
            cure = back[cure->u];
        }
        assert(cap > 0 && cap < INF);
        cure = back[t];
        while (true) {
            cost += cap * cure->c;
            cure->w -= cap;
            cure->rev->w += cap;
            if (cure->u == s) break;
            cure = back[cure->u];
        }
        flow += cap;
    }
    // instead of deleting g, we could also
    // use it to get info about the actual flow
    for (int i = 0; i < n; i++)
        for (int j = 0, len = size(g[i]); j < len; j++)
            delete g[i][j];
    delete[] g;
    delete[] back;
    delete[] dist;
    return ii(flow, cost);
}
```

4. Strings

4.1. **The Z algorithm.** Given a string S, $Z_i(S)$ is the longest substring of S starting at i that is also a prefix of S. The Z algorithm computes these Z values in $O(n)$ time, where $n = |S|$. Z values can, for example, be used to find all occurrences of a pattern P in a string T in linear time. This is accomplished by computing Z values of $S = TP$, and looking for all i such that $Z_i \geq |T|$.

```
int* z_values(string s) {
    int n = size(s);
    int* z = new int[n];
    int l = 0, r = 0;
    z[0] = n;
    for (int i = 1; i < n; i++) {
        z[i] = 0;
        if (i > r) {
            l = r = i;
            while (r < n && s[r - l] == s[r]) r++;
            z[i] = r - l; r--;
        } else if (z[i - l] < r - i + 1) z[i] = z[i - l];
        else {
            l = i;
            while (r < n && s[r - l] == s[r]) r++;
            z[i] = r - l; r--; } }
    return z;
}
```

5. Mathematics

5.1. Fraction.

A fraction (rational number) class. Note that numbers are stored in lowest common terms.

```
template <class T>
class fraction {
private:
    T gcd(T a, T b) { return b == T(0) ? a : gcd(b, a % b); }
public:
    T n, d;
    fraction(T n_, T d_) {
        assert(d_ != 0);
        n = n_, d = d_;
        if (d < T(0)) n = -n, d = -d;
        T g = gcd(abs(n), abs(d));
        n /= g, d /= g; }
    fraction(T n_) : n(n_), d(1) { }
    fraction(const fraction<T>& other) : n(other.n), d(other.d) { }
    fraction<T> operator +(const fraction<T>& other) const { return fraction<T>(n * other.d + other.n * d, d * other.d);}
    fraction<T> operator -(const fraction<T>& other) const { return fraction<T>(n * other.d - other.n * d, d * other.d);}
    fraction<T> operator *(const fraction<T>& other) const { return fraction<T>(n * other.n, d * other.d); }
    fraction<T> operator /(const fraction<T>& other) const { return fraction<T>(n * other.d, d * other.n); }
    bool operator <(const fraction<T>& other) const { return n * other.d < other.n * d; }
    bool operator <=(const fraction<T>& other) const { return !(other < *this); }
    bool operator >(const fraction<T>& other) const { return other < *this; }
    bool operator >=(const fraction<T>& other) const { return !(*this < other); }
    bool operator ==(const fraction<T>& other) const { return n == other.n && d == other.d; }
    bool operator !=(const fraction<T>& other) const { return !(*this == other); }
};
```

5.2. Binomial Coefficients.

The binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the number of ways to choose k items out of a total of n items.

```
int nck(int n, int k) {
    if (n - k < k) k = n - k;
    int res = 1;
    for (int i = 1; i <= k; i++) res = res * (n - (k - i)) / i;
    return res;
}
```

5.3. Euclidean algorithm.

The Euclidean algorithm computes the greatest common divisor of two integers a, b.

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
```

The extended Euclidean algorithm computes the greatest common divisor d of two integers a, b and also finds two integers x, y such that $a \times x + b \times y = d$.

```
int egcd(int a, int b, int& x, int& y) {
    if (b == 0) { x = 1; y = 0; return a; }
    else {
        int d = egcd(b, a % b, x, y);
        x -= a / b * y;
        swap(x, y);
        return d;
    }
}
```

5.4. Trial Division Primality Testing.

An optimized trial division to check whether an integer is prime.

```
bool is_prime(int n) {
    if (n < 2) return false;
    if (n < 4) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;
    if (n < 25) return true;
    int s = static_cast<int>(sqrt(static_cast<double>(n)));
    for (int i = 5; i <= s; i += 6) if (n % i == 0 || n % (i + 2) == 0) return false;
    return true; }
```

5.5. **Sieve of Eratosthenes.** An optimized implementation of Eratosthenes' Sieve.

```cpp
vi prime_sieve(int n) {
    int mx = (n - 3) >> 1, sq, v, i = -1;
    vi primes;
    bool* prime = new bool[mx + 1];
    memset(prime, 1, mx + 1);
    if (n >= 2) primes.push_back(2);
    while (++i <= mx) if (prime[i]) {
        primes.push_back(v = (i << 1) + 3);
        if ((sq = i * ((i << 1) + 6) + 3) > mx) break;
        for (int j = sq; j <= mx; j += v) prime[j] = false; }
    while (++i <= mx) if (prime[i]) primes.push_back((i << 1) + 3);
    delete[] prime; // can be used for O(1) lookup
    return primes; }
```

5.6. **Modular Multiplicative Inverse.** A function to find a modular multiplicative inverse.

```cpp
#include "egcd.cpp"

int mod_inv(int a, int m) {
    int x, y, d = egcd(a, m, x, y);
    if (d != 1) return -1;
    return x < 0 ? x + m : x;
}
```

5.7. **Modular Exponentiation.** A function to perform fast modular exponentiation.

```cpp
template <class T>
T mod_pow(T b, T e, T m) {
    T res = T(1);
    while (e) {
        if (e & T(1)) res = mod(res * b, m);
        b = mod(b * b, m), e >>= T(1); }
    return res;
}
```

5.8. **Chinese Remainder Theorem.** An implementation of the Chinese Remainder Theorem.

```cpp
#include "egcd.cpp"
int crt(const vi& as, const vi& ns) {
    int cnt = size(as), N = 1, x = 0, r, s, l;
    for (int i = 0; i < cnt; i++) N *= ns[i];
    for (int i = 0; i < cnt; i++) egcd(ns[i], l = N/ns[i], r, s), x += as[i] * s * l;
    return mod(x, N); }
```

5.9. **Formulas.**

- Number of ways to choose k objects from a total of n objects where order matters and each item can only be chosen once: $P_k^n = \frac{n!}{(n-k)!}$
- Number of ways to choose k objects from a total of n objects where order matters and each item can be chosen multiple times: n^k
- Number of permutations of n objects, where there are n_1 objects of type 1, n_2 objects of type 2, $\dots$, n_k objects of type k: $\binom{n}{n_1, n_2, \dots, n_k} = \frac{n!}{n_1! \times n_2! \times \cdots \times n_k!}$
- Number of ways to choose k objects from a total of n objects where order does not matter and each item can only be chosen once:
  $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{n-k} = \prod_{i=1}^{k} \frac{n-(k-i)}{i} = \frac{n!}{k!(n-k)!}, \binom{n}{0} = 1, \binom{0}{k} = 0$
- Number of ways to choose k objects from a total of n objects where order does not matter and each item can be chosen multiple times: $f_k^n = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!}$
- Number of integer solutions to $x_1 + x_2 + \cdots + x_n = k$ where $x_i \geq 0$: f_k^n
- Number of subsets of a set with n elements: 2^n
- $|A \cup B| = |A| + |B| - |A \cap B|$
- $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$
- Number of ways to walk from the lower-left corner to the upper-right corner of an $n \times m$ grid by walking only up and to the right: $\binom{n+m}{m}$
- Number of strings with n sets of brackets such that the brackets are balanced:
  $C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1}\binom{2n}{n}$
- Number of triangulations of a convex polygon with n points, number of rooted binary trees with $n+1$ vertices, number of paths across an $n \times n$ lattice which do not rise above the main diagonal: C_n
- Number of permutations of n objects with exactly k ascending sequences or *runs*:
  $\left\langle {n \atop k} \right\rangle = \left\langle {n \atop n-k-1} \right\rangle = k \left\langle {n-1 \atop k} \right\rangle + (n-k+1) \left\langle {n-1 \atop k-1} \right\rangle = \sum_{i=0}^{k} (-1)^i \binom{n+1}{i} (k+1-i)^n, \left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1$
- Number of permutations of n objects with exactly k cycles: $\left[{n \atop k} \right] = \left[{n-1 \atop k-1} \right] + (n-1) \left[{n-1 \atop k} \right]$
- Number of ways to partition n objects into k sets: $\left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}, \left\{ {n \atop 0} \right\} = \left\{ {n \atop n} \right\} = 1$

6. Geometry

6.1. **Primitives.** Geometry primitives.

```cpp
#include <complex>
#define P(p) const point &p
#define L(p0, p1) P(p0), P(p1)

typedef complex<double> point;
typedef vector<point> polygon;
double dot(P(a), P(b)) { return real(conj(a) * b); }
double cross(P(a), P(b)) { return imag(conj(a) * b); }
point rotate(P(p), P(about), double radians) { return (p - about) * exp(point(0, radians)) + about; }
point reflect(P(p), L(about1, about2)) {
    point z = p - about1, w = about2 - about1;
    return conj(z / w) * w + about1; }
bool parallel(L(a, b), L(p, q)) { return abs(cross(b - a, q - p)) < EPS; }
double ccw(P(a), P(b), P(c)) { return cross(b - a, c - b); }
bool collinear(P(a), P(b), P(c)) { return abs(ccw(a, b, c)) < EPS; }
bool collinear(L(a, b), L(p, q)) { return abs(ccw(a, b, p)) < EPS && abs(ccw(a, b, q)) < EPS;  }
double angle(P(a), P(b), P(c)) { return acos(dot(b - a, c - b) / abs(b - a) / abs(c - b)); }
double signed_angle(P(a), P(b), P(c)) { return asin(cross(b - a, c - b) / abs(b - a) / abs(c - b)); }
bool intersect(L(a, b), L(p, q), point &res, bool segment = false) {
    // NOTE: check for parallel/collinear lines before calling this function
    point r = b - a, s = q - p;
    double c = cross(r, s), t = cross(p - a, s) / c, u = cross(p - a, r) / c;
    if (segment && (t < 0-EPS || t > 1+EPS || u < 0-EPS || u > 1+EPS)) return false;
    res = a + t * r;
    return true;
}
point closest_point(L(a, b), P(c), bool segment = false) {
    if (segment) {
        if (dot(b - a, c - b) > 0) return b;
        if (dot(a - b, c - a) > 0) return a;
    }
    double t = dot(c - a, b - a) / norm(b - a);
    return a + t * (b - a);
}
double polygon_area_signed(polygon p) {
    double area = 0; int cnt = size(p);
    for (int i = 1; i + 1 < cnt; i++) area += cross(p[i] - p[0], p[i + 1] - p[0]);
    return area / 2;
}
double polygon_area(polygon p) { return abs(polygon_area_signed(p)); }
// pair<polygon, polygon> cut_polygon(const polygon &poly, point a, point b) {
//     polygon left, right;
//     point it(-100, -100);
//     for (int i = 0, cnt = poly.size(); i < cnt; i++) {
//         int j = i == cnt-1 ? 0 : i + 1;
//         point p = poly[i], q = poly[j];
//         if (ccw(a, b, p) <= 0) left.push_back(p);
//         if (ccw(a, b, p) >= 0) right.push_back(p);
//         // myintersect = intersect where (a,b) is a line, (p,q) is a line segment
//         if (myintersect(a, b, p, q, it)) left.push_back(it), right.push_back(it);
//     }
//     return pair<polygon, polygon>(left, right);
// }
```

6.2. **Convex Hull.** An algorithm that finds the Convex Hull of a set of points.

```cpp
#include "primitives.cpp"
point ch_main;
bool ch_cmp(P(a), P(b)) {
    if (collinear(ch_main, a, b)) return abs(a - ch_main) < abs(b - ch_main);
    return atan2(imag(a) - imag(ch_main), real(a) - real(ch_main)) < atan2(imag(b) - imag(ch_main),
            real(b) - real(ch_main)); }
polygon convex_hull(polygon pts, bool add_collinear = false) {
    int cnt = size(pts), main = 0, i = 1;
    if (cnt <= 3) return pts;
    for (int i = 1; i < cnt; i++)
        if (imag(pts[i]) < imag(pts[main]) || abs(imag(pts[i]) - imag(pts[main]) < EPS && imag(pts[i]) > imag(pts[main]))
            main = i;
    swap(pts[0], pts[main]);
    ch_main = pts[0];
    sort(++pts.begin(), pts.end(), ch_cmp);
    point prev, now;
    stack<point> S; S.push(pts[cnt - 1]); S.push(pts[0]);
    while (i < cnt) {
        now = S.top(); S.pop();
        if (S.empty()) {
            S.push(now);
            S.push(pts[i++]);
        } else {
            prev = S.top();
            S.push(now);
            if (ccw(prev, now, pts[i]) > 0 || (add_collinear && abs(ccw(prev, now, pts[i])) < EPS)) S.push(pts[i++]);
            else S.pop();
        } }
    vector<point> res;
    while (!S.empty()) res.push_back(S.top()), S.pop();
    return res;
}
```

6.3. **Formulas.** Let $a = (a_x, a_y)$ and $b = (b_x, b_y)$ be two-dimensional vectors.

- $a \cdot b = |a||b| \cos\theta$, where θ is the angle between a and b.
- $a \times b = |a||b| \sin\theta$, where θ is the signed angle between a and b.
- $a \times b$ is equal to the area of the parallelogram with two of its sides formed by a and b. Half of that is the area of the triangle formed by a and b.

7. Other Algorithms

7.1. **Binary Search.** An implementation of binary search that finds a real valued root of the continous function f on the interval $[a, b]$, with a maximum error of ε.

```
double binary_search_continuous(double low, double high, double eps, double (*f)(double)) {
    while (true) {
        double mid = (low + high) / 2, cur = f(mid);
        if (abs(cur) < eps) return mid;
        else if (0 < cur) high = mid;
        else low = mid;
    }
}
```

Another implementation that takes a binary predicate f, and finds an integer value x on the integer interval $[a, b]$ such that $f(x) \wedge \neg f(x-1)$.

```
int binary_search_discrete(int low, int high, bool (*f)(int)) {
    assert(low <= high);
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (f(mid)) high = mid;
        else low = mid + 1;
    }
    assert(f(low));
    return low;
}
```

7.2. **2SAT.** A fast 2SAT solver.

```
#include "../graph/scc.cpp"

bool two_sat(int n, const vii& clauses, vi& all_truthy) {
    all_truthy.clear();
    vvi adj(2*n+1);
    for (int i = 0, cnt = size(clauses); i < cnt; i++) {
        adj[-clauses[i].first + n].push_back(clauses[i].second + n);
        if (clauses[i].first != clauses[i].second)
            adj[-clauses[i].second + n].push_back(clauses[i].first + n);
    }
    pair<union_find, vi> res = scc(adj);
    union_find scc = res.first;
    vi dag = res.second;
    vi truth(2*n+1, -1);
    for (int i = 2*n; i >= 0; i--) {
        int cur = order[i] - n, p = scc.find(cur + n), o = scc.find(-cur + n);
        if (cur == 0) continue;
        if (p == o) return false;
        if (truth[p] == -1) truth[p] = 1;
        truth[cur + n] = truth[p];
        truth[o] = 1 - truth[p];
        if (truth[p] == 1) all_truthy.push_back(cur);
    }
    return true;
}
```

7.3. ***n*th Permutation.** A very fast algorithm for computing the nth permutation of the list $\{0, 1, \ldots, k-1\}$.

```
vector<int> nth_permutation(int cnt, int n) {
    vector<int> idx(cnt), per(cnt), fac(cnt);
    for (int i = 0; i < cnt; i++) idx[i] = i;
    for (int i = 1; i <= cnt; i++) fac[i - 1] = n % i, n /= i;
    for (int i = cnt - 1; i >= 0; i--) per[cnt - i - 1] = idx[fac[i]], idx.erase(idx.begin() + fac[i]);
    return per;
}
```

7.4. **Cycle-Finding.** An implementation of Floyd's Cycle-Finding algorithm.

```
ii find_cycle(int x0, int (*f)(int)) {
    int t = f(x0), h = f(t), mu = 0, lam = 1;
    while (t != h) t = f(t), h = f(f(h));
    h = x0;
    while (t != h) t = f(t), h = f(h), mu++;
    h = f(t);
    while (t != h) h = f(h), lam++;
    return ii(mu, lam);
}
```

7.5. **Dates.** Functions to simplify date calculations.

```
int intToDay(int jd) { return jd % 7; }
int dateToInt(int y, int m, int d) {
    return 1461 * (y + 4800 + (m - 14) / 12) / 4 +
        367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
        3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
        d - 32075;
}
void intToDate(int jd, int &y, int &m, int &d) {
    int x, n, i, j;
    x = jd + 68569;
    n = 4 * x / 146097;
    x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001;
    x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447;
    d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x;
    y = 100 * (n - 49) + i + x;
}
```

8. Useful Information

8.1. **Tips & Tricks.**

- How fast does our algorithm have to be? Can we use brute-force?
- Does order matter?
- Is it better to look at the problem in another way? Maybe backwards?
- Are there subproblems that are recomputed? Can we cache them?
- Do we need to remember everything we compute, or just the last few iterations of computation?
- Does it help to sort the data?
- Can we speed up lookup by using a map (tree or hash) or an array?
- Can we binary search the answer?
- Can we add vertices/edges to the graph to make the problem easier? Can we turn the graph into some other kind of a graph (perhaps a DAG, or a flow network)?
- Make sure integers are not overflowing.
- Is it better to compute the answer modulo n? Perhaps we can compute the answer modulo $m_1, m_2, \ldots, m_k$, where $m_1, m_2, \ldots, m_k$ are pairwise coprime integers, and find the real answer using CRT?

- Are there any edge cases? When $n = 0, n = -1, n = 1, n = 2^{31} - 1$ or $n = -2^{31}$? When the list is empty, or contains a single element? When the graph is empty, or contains a single vertex? When the graph contains self-loops? When the polygon is concave or non-simple?
- Can we use exponentiation by squaring?

8.2. **Fast Input Reading.** If input or output is huge, sometimes it is beneficial to optimize the input reading/output writing. This can be achieved by reading all input in at once (using fread), and then parsing it manually. Output can also be stored in an output buffer and then dumped once in the end (using fwrite). A simpler, but still effective, way to achieve speed is to use the following input reading method.

```
void readn(register int *n) {
    int sign = 1;
    register char c;
    *n = 0;
    while((c = getc_unlocked(stdin)) != '\n') {
        switch(c) {
            case '-': sign = -1; break;
            case ' ': goto hell;
            case '\n': goto hell;
            default: *n *= 10; *n += c - '0'; break;
        }
    }
hell:
    *n *= sign;
}
```

8.3. **Worst Time Complexity.**

n	Worst AC Algorithm	Comment
≤ 10	$O(n!), O(n^6)$	e.g. Enumerating a permutation
≤ 15	$O(2^n \times n^2)$	e.g. DP TSP
≤ 20	$O(2^n), O(n^5)$	e.g. DP + bitmask technique
≤ 50	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, choosing ${}_nC_k = 4$
$\leq 10^2$	$O(n^3)$	e.g. Floyd Warshall's
$\leq 10^3$	$O(n^2)$	e.g. Bubble/Selection/Insertion sort
$\leq 10^5$	$O(n \log_2 n)$	e.g. Merge sort, building a Segment tree
$\leq 10^6$	$O(n), O(\log_2 n), O(1)$	Usually, contest problems have $n \leq 10^6$ (e.g. to read input)

8.4. **Bit Hacks.**

- `n & -n` returns the first set bit in n.
- `n & (n - 1)` is 0 only if n is a power of two.
- `snoob(x)` returns the next integer that has the same amount of bits set as `x`. Useful for iterating through subsets of some specified size.

```
int snoob(int x) {
    int y = x & -x, z = x + y;
    return z | ((x ^ z) >> 2) / y;
}
```