1 Init

1.1 Debug

```
g++ -Wall -Wextra -pedantic -g -std=c++11 -O2 -Wshadow -Wformat=2 -Wfloat-equal -Wconversion -Wlogical-op
-Wcast-qual -Wcast-align -D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC -fsanitize=address -fsanitize=undefined
-fstack-protector
```

1.2 Grader

```python
from subprocess import check_output as co
import sys, os
_, e, d, i, o = sys.argv # input <exe> <dir> <inp-ext> <out-ext>
for (_, _, fs) in os.walk(d):
    for f in fs:
        if f[-len(i):] != i: continue
        out = [s.strip().decode() for s in co(e, stdin=open(os.path.join(d, f))).strip().splitlines()]
        exp = [s.strip() for s in open(os.path.join(d, f[:-len(i)]+o)).read().strip().splitlines()]

        print(f[:-len(i)], (exp == out))
    break
```

1.3 Complexity

Value	Complexity	Algorithms
$n < 10^{18}$	O(1), O(polylog(N))	Binary search, Functions (math)
$n < 10^{9}$	O(sqrt(N))	Prime check, factorization
$n < 10^{6}$	O(N), O(N log(N))	Greedy, Sorting, Binary search + Greedy, Divide and conquer, 1D Dynamic programming
$n < 10^{3}$	O(N^2), O(N^2 log(N))	2D Dynamic programming, All-pair shortest path
$n < 100$	O(N^3)	Max-flow, Various unoptimized traversals
$n < 20$	O(2^N)	Combinations
$n < 16$	O(N 2^N), O($N^2 2^N$)	Bitmask dynamic programming
$n < 10$	O($N!$)	Permutations

1.4 Common errors

- Loop bounds (especially with DP)
- Array bounds (make slightly bigger)
- Initialization (init and reset)
- Incorrect output format
- Wrong nesting
- Precision (use epsilon)
- Overflow (use 64-bit when in doubt)
- Invalid expressions (divide by zero, segfault)
- Index offset
- Rounding (floor, ceil or normal)
- Read complete input
- Boundary cases
- Wrong variable, copying mistakes, etc

2 STL

2.1 I/O

```
std::ios::sync_with_stdio(false); //speedup IO (dont combine with printf/scanf)
std::cin.ignore(n); //ignore n characters before continue
std::getline(std::cin, s); //reads whole line into string
std::cin >> std::noskipws; //dont skip whitespace
std::cout << std::fixed; //use fixed-point notation
std::cout << std::setprecision(n); //set the precision to n decimals
std::cout << std::setw(n); //set the length each output will contain
std::cout << std::setfill(c); //set the char to fill the remaining chars from above
```

2.2 String

```
//string split
std::vector<std::string> split(const std::string &s, char delim) {
    std::vector<std::string> elems;
    std::stringstream stream(s);
    std::string item;
    while (std::getline(stream, item, delim)) {
        elems.push_back(item);
    }
    return elems;
}
//substring
str.substr(pos, LENGTH); //till end if LENGTH is empty
//string find
str.find("substring") //return std::string::npos if not found, else position first letter
```

2.3 Containers

```
std::vector<int> vec;
//sort from small to large
bool cmp(int a, int b){
    return a < b;
}
std::sort(vec.begin(), vec.end(), cmp);
std::sort(vec.begin(), vec.end(), std::less<int>());
 ↪ //use std::greater<int>() for large to small and std::less<int>() for small to large
std::find(vec.begin(), vec.end(), NEEDLE); //use .find() if avaible (map/set)
std::count(vec.begin(), vec.end(), NEEDLE); //use .find() if available (map/set)
//find union/intersection/difference -- both need to be SORTED already
std::vector<int> tmp(vec.size()+vec2.size());
std::set_intersection(vec.begin(), vec.end(), vec2.begin(), vec2.end(), tmp.begin());
std::set_union(vec.begin(), vec.end(), vec2.begin(), vec2.end(), tmp.begin());
std::set_difference(vec.begin(), vec.end(), vec2.begin(), vec2.end(). tmp.begin());
//filter unique elements
auto last = std::unique(vec.begin(), vec.end());
vec.erase(last, v.end());
//get median (or other nth element in O(N))
std::nth_element(v.begin(), v.begin() + v.size()/2, v.end());
```

2.4 String streams

```
//c-style for complex parsing
int in;
sscanf(str.c_str(),"%d",&in)
//convert number to string
```

```
std::to_string(str)
//convert string to number(s)
std::stoi(in) //stoll for long long
```

2.5 Constants

```
INT_MIN
INT_MAX
LLONG_MIN
LLONG_MAX
PI //defined in header
EPS //defined in header
```

2.6 Math

```
ceil(a); //round up
floor(a); //round down
round(a); //round nearest
atan2(a, b); //arctan with two parameters
frac_part = modf(d, intpart); //split up in parts (int_part is a double!)
std::__gcd(a, b); //greatest common divisor a, b
```

2.7 Permutations

O($N!$)

```
std::vector arr;
std::sort(arr.begin(), arr.end());
do {
   //do something with permutation
} while (next_permutation(arr.begin(),arr.end());
```

2.8 Binary search trees

O(log(N))

```
std::unordered_set<int, int> us; //hash table allow retrieval in O(1) -- also unordered_map
std::multimap<int, int> m; //multimap can save values multiple times
m.lower_bound(i); //first element equal or higher then i
m.higher_bound(i); //first element higher then i
std::pair<auto iter, auto iter> p = m.equal_range(i); //get all with equal value
```

2.9 Bit twiddling hacks

```
in |= (1 << a); //enable bit at position a
in ^= (1 << a); //toggle bit at position a
if(in & (1 << a)); //check bit at position a
//get last bit set (least signifcant)
c = (b & -b)
//ll for the long long versions
__builtin_popcount(in); //count the amount of bits set
__builtin_ffs(in); //give the least signifcant index + 1 in binary representation
__builtin_clz(in); //returns the number of leading zeros
__builtin_ctz; //give number of trailing zeros
```

3 Datastructures

3.1 BIT

O(log(N))

```
//use a std::map to save very large tables or for simple 2D (faster below)
int lst[MAXN];
```

```cpp

int sum(int b){
    int sum = 0; b+=1;
    for(; b; b-=(b&(-b))){
        sum += lst[b];
    }
    return sum;
}

int update(int b, int v){
    b+=1;
    for(; b<MAXN; b+=(b&(-b))){
        lst[b] += v;
    }
}
```

3.2 2D-BIT

O(N log(N))

```cpp
//online O(N (log(N)^2) is also possible using std::map<int, int>
std::pair<int, int> rs[MAXN]; //list of points (y,x)
std::vector<int> lst2[MAXN];
std::vector<int> order[MAXN];

int sum(int b, int c){
    int sum = 0; b+=1; c+=1;
    for(; b; b-=(b&(-b))){
        int h = upper_bound(order[b].begin(), order[b].end(), c) - order[b].begin();
        for(int d = h; d; d-=(d&(-d))){
            sum += lst2[b][d];
        }
    }
    return sum;
}

void update(int b, int c, int v){
    b+=1; c+=1;
    int unt = 0;
    for(; b<MAXN; b+=(b&(-b))){
        int h = upper_bound(order[b].begin(), order[b].end(), c) - order[b].begin();
        for(int d = h; d<lst2[b].size(); d+=(d&(-d))){
            lst2[b][d] += v;
        }
    }
}

void init(){
    std::sort(rs, rs+MAXN);
    for(int i=0; i<MAXN; ++i){
        for(int j=rs[i].second+1; j<MAXN; j+=(j&(-j))){
            order[j].push_back(rs[i].first+1);
            lst2[j].push_back(0);
        }
    }
    for(int i=0; i<MAXN; ++i) lst2[i].push_back(0);
}
```

3.3 Segment tree

O(log(N))

```cpp
// inclusion segment tree (a node includes both it endpoints!)
struct Node{
```

```
Node(): li(0), ri(0), l(0), r(0) {}
    int li;
    int ri;
    Node *l;
    Node *r;

    int val;

    bool hupd;
    int upd;
};

Node *r;

Node *build(int li, int ri) {
    Node *n = new Node;
    n->li = li;
    n->ri = ri;
    n->upd = 0;
    if(li == ri){
        n->val = 0; //init
    }else{
        int mi = (li+ri)/2;
        n->r = build(li, mi);
        n->l = build(mi+1, ri);

        n->val = std::min(n->r->val, n->l->val); // merge
    }
    return n;
}

void pushd(Node *n){
    if(!n->hupd) return;
    n->l->val = n->upd; //update
    n->r->val = n->upd; //update
    n->l->upd = n->upd; //split (move old updates)
    n->r->upd = n->upd; //split (move old updates)
    n->l->hupd = true;
    n->r->hupd = true;
    n->hupd = false;
    n->upd = 0;
}

int query(Node *n, int li, int ri){
    if(ri < n->li || n->ri < li){
        //outside
        return INT_MAX;
    }else if(li <= n->li && n->ri <= ri){
        //inside
        return n->val;
    }
    pushd(n);

    int la = query(n->l, li, ri);
    int ra = query(n->r, li, ri);
    return std::min(la, ra); // merge
}

void update(Node *n, int li, int ri, int v){
    if(ri < n->li || n->ri < li){
        //outside
        return;
    }else if(li <= n->li && n->ri <= ri){
        //inside
        if(n->ri != n->li){
            pushd(n);
            n->upd = v; //split
            n->hupd = true;
        }
        n->val = v; //update
        return;
    }
    pushd(n);

    update(n->l, li, ri, v);
    update(n->r, li, ri, v);
    n->val = std::min(n->r->val, n->l->val); // merge
}
```

3.4 Union-Find

O(α)

```
int pr[MAXN];
int sz[MAXN];

int find(int k){
    if(pr[k] == k) return k;
    else return pr[k] = find(pr[k]);
}
void merge(int a, int b){
    a = find(a); b = find(b);
    if(a == b) return;
    if(sz[a] > sz[b]) std::swap(a, b);

    pr[a] = b;
    sz[b] += sz[a];
    sz[a] = 0;
}
```

4 Math

4.1 GCD/LCM

O(log(K))

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

int lcm(int a, int b){
    return a*b/gcd(a,b);
}
```

4.2 Extended Euclidean

O(log(K))

```
//determines a and b satisfiying s*a+t*b == gcd(a, b)
std::pair<int, int> extgcd(int a, int b){
    int s = 0, old_s = 1;
    int t = 1, old_t = 0;
    int r = b, old_r = a;
    while(r){
        int q = old_r/r;
        int sv;
        sv = r;
        r = old_r - q*r; old_r = sv;
        sv = s;
        s = old_s - q*s; old_s = sv;
        sv = t;
        t = old_t - q*t; old_t = sv;
    }
    return std::make_pair(old_s, old_t);
    //return s, t if you want with result = zero
}
```

4.3 Modular multiplicative inverse

O(log(K))

```
//needs extended euclidean
int mod_inverse(int a, int m){
    if(gcd(a, m) != 1) return -1; //inverse does not exist
    int inv = extgcd(a, m).first;
    if(inv < 0) {
        int mlt = inv/m;
        if(inv % m) mlt--;
        inv -= mlt*m;
    }
    return inv;
}
```

4.4 Modular exponentiation

O(log(K))

```
//trick also works with matrices
long long expmod(long long a, long long b, long long m){
    long long res = 1;
    a = a%m;
    while(b > 0){
        if((b%2) == 1) res = (res*a)%m;
        b >>= 1;
        a = (a*a) % m;
    }
    return res;
}
```

4.5 Fibonacci

O(N)

```
int fib(int n){
    int a = 0, b = 1;
    for(int i=0; i<n; ++i){
        int t = b;
        b = a+b;
        a = t;
    }
    return a;
}
```

4.6 Combinations

O(N^2)

```
int cmb[MAXN][MAXN];
int comb(int i, int j){
    if(i < 0) return 0;
    else if(j == 0) return 1;
    else if(cmb[i][j] != -1) return cmb[i][j];
    else return cmb[i][j] = comb(i-1,j-1)+comb(i-1,j);
}
```

4.7 Gaussian elimination

O(N^3)

```
int N; //amount of rows (columms = rows + 1)
double aug[MAXN][MAXN];
double ans[MAXN];
void gaussian_elimination() {
    // the forward elimination phase
    for (int i = 0; i < N - 1; i++) {
        int l = i;

        // which row has larest column value
        for (int j = i + 1; j < N; j++) if (fabs(aug[j][i]) > fabs(aug[l][i])) l = j;
        // swap this pivot row, reason: minimize floating point error
        for (int k = i; k <= N; k++){
                double t = aug[i][k];
                aug[i][k] = aug[l][k];
                aug[l][k] = t;
        }
        // the actual forward elimination phase
        for (int j = i + 1; j < N; j++)
            for (int k = N; k >= i; k--) aug[j][k] -= aug[i][k] * aug[j][i] / aug[i][i];

    }

    // the back substitution phase
    for (int j = N - 1; j >= 0; j--) {
        double t = 0.0;
        for (int k = j + 1; k < N; k++) t += aug[j][k] * ans[k];
        ans[j] = (aug[j][N] - t) / aug[j][j];
    }
}
```

4.8 Prime generation

O(N log(log(N)))

```
std::vector<long long> prms;
long long is_prm[MAXN];

//generate primes needed for functions below
void init_prms(){
    for(long long i=0; i<MAXN; ++i) is_prm[i] = true;
    for(long long i=2; i<MAXN; ++i){
        if(!is_prm[i]) continue;
        prms.push_back(i);
        for(long long j=i*i; j<MAXN; j+=i) is_prm[j] = false;
    }
}
```

4.9 Is prime

O(sqrt(N))

```
bool is_prime(long long n){
    for(size_t i=0; i<prms.size(); ++i){
        if(prms[i]*prms[i] > n) return true;
        if((n % prms[i]) == 0) return false;
    }
    return true;
}
```

4.10 Factorize

O(N log(log(N)))

```
//returns list of factors and how many times they occur
std::vector<std::pair<int, int> > factors(long long n){
    std::vector<std::pair<int, int> > v;
    for(size_t i=0; i<prms.size(); ++i){
        long long k = 0;
        while((n % prms[i]) == 0){
                ++k;
                n/=prms[i];
        }
        if(k) v.push_back(std::make_pair(prms[i], k));
    }
    if(n!=1) v.push_back(std::make_pair(n, 1));
    return v;
}
```

4.11 Is prime (fast)

O(polylog(N))

```
//deterministic until 2^64 -- needs long long expmod probably!
bool miller_rabin(long long n){
    long long prms[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

    long long s = 0, d = n-1;
    while((d % 2) == 0){
        d/=2;
        s++;
    }
    for(int i=0; i<12; ++i){
        if(prms[i] >= n) break;
        if(expmod(prms[i], d, n) == 1) continue;
        long long c = 1;
        int j = 0;
        for(j=0; j<s; ++j){
            if(expmod(prms[i], c*d, n) == n-1) break;
            c *= 2;
        }
        if(j == s) return false;
    }
    return true;
}
```

5 Graph

5.1 Header

```
struct Edge{
    //add constructor if convenient
    int f; //from
    int t; //to
    int d; //distance

    int cap; //capacity (max flow)
    int flw; //flow (max flow)
    Edge *rev; //reverse edge (in case undirected)

    bool use; //edge is used (euler-tour)
};

struct Node{
    int n; //index
    std::vector<Edge*> ed; //adjadency list
};

Node nd[MAXN]; //nodes
int dis[MAXN]; //distance to node

int dpt[MAXN]; //depth node (SCC/bridge)
int low[MAXN]; //low-link node (SCC/bridge)
//next edge of a node, used to keep track of next path in a non-recursive dfs
std::vector<Edge*>::iterator eit[MAXN];

Edge *frm[MAXN]; //edge used to node
bool vis[MAXN]; //visited

int incnt[MAXN]; //active indegree count (topo-sort)

void init(){
    for(int i=0; i<MAXN; ++i){
        nd[i].n = i;
        for(size_t j=0; j<nd[i].ed.size(); ++j) delete nd[i].ed[j];
        nd[i].ed.clear();
        dis[i] = INT_MAX;
        dpt[i] = -1;
        low[i] = INT_MAX;
        frm[i] = 0;
        vis[i] = false;
        incnt[i] = 0;
    }
}
```

5.2 BFS

O(E+V)

```
void bfs(int F){
    for(int i=0; i<MAXN; ++i) dis[i] = INT_MAX;
    std::queue<int> q; //replace by stack for dfs
    dis[F] = 0;
    q.push(F);
    while(!q.empty()){
        int p = q.front();
        q.pop();
        for(auto iter = nd[p].ed.begin(); iter != nd[p].ed.end(); ++iter){
            Edge *e = *iter;
            if(dis[e->t] == INT_MAX){
                dis[e->t] = dis[p] + 1;
                q.push(e->t);
            }
        }
    }
}
```

5.3 Dijkstra

O((E+V)log(V))

```
int phi[MAXN];
#define POT(u,v) (phi[u] - phi[v])

```

```
void dijkstra(int F){
    for(int i=0; i<MAXN; ++i) dis[i] = INT_MAX, phi[i] = 0;

    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int > >, std::greater<std::pair<int, int > > > pq;
    dis[F] = 0;
    pq.push(std::make_pair(0, F));
    while(!pq.empty()){
        std::pair<int, int> p = pq.top();
        pq.pop();
        if(dis[p.second] != p.first) continue;
        for(auto iter = nd[p.second].ed.begin(); iter != nd[p.second].ed.end(); ++iter){
            Edge *e = *iter;
            if(p.first + e->d + POT(p.first, e->t) < dis[e->t]){
                dis[e->t] = p.first + e->d + POT(p.first, e->t);
                pq.push(std::make_pair(dis[e->t], e->t));
            }
        }
    }

    //addition if we want to work with negative-weight paths (min-cost max-flow)
    //for(int i=0; i<MAXN; ++i) if(phi[i] < INT_MAX) phi[i] += dis[i];
}
```

5.4 Floyd-Warshall

$O(V^3)$

```
int dism[MAXN][MAXN];

void floyd_warshall(){
    for(int k=0; k<MAXN; ++k){
        for(int i=0; i<MAXN; ++i){
            for(int j=0; j<MAXN; ++j){
                //line depends on exercise (this is maximum edge on minimum path)
                dism[i][j] = std::min(dism[i][j], std::max(dism[i][k], dism[k][j]));
            }
        }
    }
}
```

5.5 Bellman-Ford

O(E V)

```
bool bellman_ford(int F){
    //compute shortest path
    for(int i=0; i<MAXN; ++i) dis[i] = INT_MAX;
    dis[F] = 0;
    for(int k=0; k<MAXN-1; ++k){
        for(int i=0; i<MAXN; ++i){
            for(auto iter = nd[i].ed.begin(); iter != nd[i].ed.end(); ++iter){
                Edge *e = *iter;
                if(dis[i] + e->d < dis[e->t]){
                    dis[e->t] = dis[i] + e->d;
                }
            }
        }
    }

    //check for negative weight path
    for(int i=0; i<MAXN; ++i){
        for(auto iter = nd[i].ed.begin(); iter != nd[i].ed.end(); ++iter){
            Edge *e = *iter;
            if(dis[i] + e->d < dis[e->t]) return false;
        }
    }
    return true;
}
```

5.6 Minimum spanning tree

O(E log(V))

```
//needs union-find
bool cmp(Edge *e1, Edge *e2) { return e1->d < e2->d; }

std::vector<Edge*> kruskal(std::vector<Edge*> &edg){
    std::vector<Edge*> vec;

    std::sort(edg.begin(), edg.end(), cmp);
    for(size_t i=0; i<edg.size(); ++i){
        int a = find(edg[i]->f);
        int b = find(edg[i]->t);
        if(a != b){
            vec.push_back(edg[i]);
            merge(a, b);
        }
    }

    return vec;
}
```

5.7 Biconnected components

O(E+V)

```
std::vector<int> biconnected(int r){
    for(int i=0; i<MAXN; ++i) eit[i] = nd[i].ed.begin();
    std::vector<int> res; //result nodes

    std::stack<int> st; //stack
    st.push(r);
    dpt[r] = 0;
    while(!st.empty()){
        int c = st.top();
        st.pop();

        //add to result
        if(eit[c] == nd[c].ed.end()){
            int cnt = 0;
            for(auto iter = nd[c].ed.begin(); iter != nd[c].ed.end(); ++iter){
                int n = (*iter)->t;
                if(dpt[c] != 0 && low[n] >= dpt[c] && dpt[n] == dpt[c]+1){ //other edges
                    res.push_back(c);
                    break;
                }else if(dpt[c] == 0 && dpt[n] == dpt[c]+1){ //root
                    dpt[n] = 0;
                    ++cnt;
                }
            }
            if(dpt[c] == 0 && cnt >= 2) res.push_back(c); //root
            continue;
        }

        //init
        if(low[c] == INT_MAX) low[c] = dpt[c];

        //loop through children (non-recursive so we should come back to this node)
        st.push(c);
        auto iter = eit[c];
        for(; iter!=nd[c].ed.end(); ++iter){
            int n = (*iter)->t;
            if(dpt[n] != -1){
                if(dpt[n] < dpt[c]-1) low[c] = std::min(low[c], dpt[n]); // back edge
                else if(dpt[n] > dpt[c]) low[c] = std::min(low[c], low[n]); // forward edge
            }else{
                dpt[n] = dpt[c]+1;
                st.push(n);
                break;
            }
        }
        eit[c] = iter;
    }
    return res;
}
```

5.8 Strongly connected components

O(E+V)

```
int tarjan_ind = 0;
std::vector<std::vector<int> > tarjan(int r){
    for(int i=0; i<MAXN; ++i) eit[i] = nd[i].ed.begin();
    std::vector<std::vector<int> > res; //result components

    std::stack<int> st; //stack
    std::stack<int> cp; //current component
    st.push(r);
    dpt[r] = ++tarjan_ind;
    while(!st.empty()){
        int c = st.top();
        st.pop();

        //init
        if(low[c] == INT_MAX){
            cp.push(c);
            low[c] = dpt[c];
        }

        //add to result
        if(eit[c] == nd[c].ed.end()){
            if(low[c] == dpt[c]){
                res.push_back(std::vector<int>());
                while(true){
                    int n = cp.top();
                    dpt[n] = INT_MAX; low[n] = INT_MAX;
                    res.back().push_back(n);
                    cp.pop();
                    if(n == c) break;
                }
            }
            continue;
        }

        //loop through children (non-recursive so we should come back to this node)
        st.push(c);
        auto iter = eit[c];
        for(; iter!=nd[c].ed.end(); ++iter){
            int n = (*iter)->t;
            if(dpt[n] != -1){
                if(dpt[n] < dpt[c]) low[c] = std::min(low[c], dpt[n]); // back edge
                else if(dpt[n] > dpt[c]) low[c] = std::min(low[c], low[n]); // forward edge
            }else{
                dpt[n] = ++tarjan_ind;
                st.push(n);
                break;
            }
        }
        eit[c] = iter;
    }
    return res;
}
```

5.9 2-SAT

O(E+V)

```
//needs tarjan (MAXN = 2x the amount of variables)
int asgn[MAXN]; //start from 2 all even indexes contain normal var

//give terms with negative as negotiation (make sure no zero used!)
int comp[MAXN];
int casg[MAXN];
bool two_sat(std::vector<std::pair<int, int> > terms){
    for(int i=0; i<MAXN; ++i) comp[i] = casg[i] = -1;

    //build graph
    for(size_t i=0; i<terms.size(); ++i){
        std::pair<int, int> p = terms[i];
        Edge *e = new Edge();
        if(p.first < 0) e->f = -2*p.first;
        else e->f = 2*p.first+1;
        if(p.second < 0) e->t = -2*p.second+1;
        else e->t = 2*p.second;
        nd[e->f].ed.push_back(e);
        e = new Edge();
        if(p.second < 0) e->f = -2*p.second;
        else e->f = 2*p.second+1;
        if(p.first < 0) e->t = -2*p.first+1;
        else e->t = 2*p.first;
        nd[e->f].ed.push_back(e);
    }

    //apply tarjan
    std::vector<std::vector<int> > all;
    for(int k=0; k<MAXN; ++k){
        if(dpt[k] != -1) continue;
        std::vector<std::vector<int> > vec = tarjan(k);
        for(size_t i=0;
         ↪  i<vec.size(); ++i) for(size_t j=0; j<vec[i].size(); ++j) comp[vec[i][j]] = i+all.size();
        all.insert(all.end(), vec.begin(), vec.end());
    }

    //reverse topological traverse
    for(size_t i=0; i<all.size(); ++i){
        if(casg[i] == -1) casg[i] = true;

        for(size_t j=0; j<all[i].size(); ++j){
            int chk = comp[all[i][j]]/2;
            if(comp[chk*2] == comp[chk*2+1]) return false;

            if(all[i][j] % 2) casg[comp[all[i][j]-1]] = !casg[i];
            else casg[comp[all[i][j]+1]] = !casg[i];
        }
    }

    //set assignment (2 contains assignment var 1, 4 assignment var 2 etc...)
    for(size_t i=0; i<all.size(); ++i) for(size_t j=0; j<all[i].size(); ++j) asgn[all[i][j]/2] = casg[i];
    return true;
}
```

5.10 Max Flow

O(V^2E)

```
//bfs is safer (unless path length is limited and integer)
//for min-cost replace by Bellman-Ford or Dijkstra with potentials
int dfs(int a, int b){
    if(a == b) return INT_MAX;
    if(vis[a]) return 0;
    vis[a] = true;

    for(size_t i=0; i<nd[a].ed.size(); ++i){
        Edge *e = nd[a].ed[i];

        int cap = (e->cap-e->flw)+e->rev->flw;
        if(cap == 0) continue;
        else{
            int k = dfs(e->t, b);
            if(k == 0) continue;
            frm[e->t] = e;
            return std::min(cap, k);
        }
    }
    return 0;
}

int max_flow(int a, int b){
    int mf = 0;

    frm[a] = 0;
    while(true){
        for(int i=0; i<MAXN; ++i) vis[i] = false;
        int f = dfs(a, b);
        if(f == 0) break;
        mf += f;

        int lst = b;
        while(frm[lst]){
            Edge *nr = frm[lst];

            int rf = std::min(nr->cap-nr->flw, f);
            nr->flw += rf;
            nr->rev->flw -= f-rf;

            lst = frm[lst]->f;
        }
    }
    return mf;
}
```

5.11 Bipartite matching

O(VE)

```
int mtch[MAXN]; //only the size of one side
bool aug(int n) {
    if (vis[n]) return false;
    vis[n] = true;

    for (size_t i = 0; i < nd[n].ed.size(); i++) {
        Edge *e = nd[n].ed[i];
        //try match with edge (if available or previous can be rematched)
        if (mtch[e->t] == -1 || aug(mtch[e->t])) {
            mtch[e->t] = n;
            return true;
        }
    }
    return false;
}

void bipartite_matching() {
    for (int i = 0; i < MAXN; i++) mtch[i] = -1;

    int M = 0; //contains the maximum matching
    for (int i = 0; i < MAXN; i++) { //try to start match from i
        for (int j = 0; j<MAXN; j++) vis[j] = false;
        if (aug(i)) M++;
    }
}
```

5.12 Euler tour

O(V^2E)

```
//start from one of the two non-even edges (if trail)
std::list<int> euler_tour(int s){
    std::list<int> ans;
    for(int i=0; i<MAXN; ++i) eit[i] = nd[i].ed.begin();

    std::stack<int> st;
    st.push(s);
    while(!st.empty()){
        int c = st.top();
        st.pop();

        auto iter = eit[c];
        if(iter == nd[c].ed.end()){
            ans.push_front(c);
            continue;
        }

        st.push(c);
        for(; iter != nd[c].ed.end(); ++iter){
            Edge *e = *iter;
            if(e->use) continue;
            e->use = e->rev->use = true;
            st.push(e->t);
            break;
        }
        eit[c] = iter;
    }

    return ans;
}
```

5.13 Topological sort

O(E+V)

```
int ecnt[MAXN];

std::vector<int> toposort(){
    std::vector<int> res;
    std::queue<int> q;
    for(int i=0; i<MAXN; ++i){
        if(incnt[i] == 0) q.push(i);
    }

    while(!q.empty()){
        int c = q.front();
        q.pop();
        res.push_back(c);
        for(auto iter = nd[c].ed.begin(); iter != nd[c].ed.end(); ++iter){
            Edge *e = (*iter);
            --incnt[e->t];
            if(incnt[e->t] == 0) q.push(e->t);
        }
    }
    return res;
}
```

5.14 LCA

O(E+V)

```
//set MAXK to the log2 of MAXN
int par[MAXN][MAXK]; //example of sparse table idea
void construct(int k, int p = -1){
    if(dpt[k] != -1) return;

    if(p == -1) dpt[k] = 0;
    else dpt[k] = dpt[p]+1;

    //compute parents
    par[k][0] = p;
    for(int i=1; i<MAXK; ++i){
        if(par[k][i-1] == -1) break;
        par[k][i] = par[par[k][i-1]][i-1];
    }

    //dfs children
    for(size_t i=0; i<nd[k].ed.size(); ++i){
        int t = nd[k].ed[i]->t;

        construct(t, k);
    }
}

int query(int a, int b){
    if(dpt[a] < dpt[b]) std::swap(a, b);

    //level out
    int k = 0;
    for(; k<MAXK; ++k){
        if(par[a][k] == -1 || dpt[par[a][k]] < dpt[b]) break;
    }
    --k;
    while(dpt[a] != dpt[b]){
        a = par[a][k];
        while(k>=0 && (par[a][k] == -1 || dpt[par[a][k]] < dpt[b])) --k;
    }

    //go up
    for(k=0; k<MAXK; ++k){
        if(par[a][k] == par[b][k]) break;
    }
    --k;
    while(k >= 0){
        a = par[a][k];
        b = par[b][k];
        while(k >= 0 && par[a][k] == par[b][k]) --k;
    }
    //do last jump if necessary
    if(a != b){
        k++;

        a = par[a][k];
        b = par[b][k];
    }

    return a;
}
```

6 Geometry

6.1 Headers

```
#define PI (2*std::acos(0.0))
#define EPS 1e-9

struct Vec{
    Vec(): x(0), y(0), z(0) {}
    Vec(Vec p1, Vec p2): x(p2.x-p1.x), y(p2.y-p1.y), z(p2.z-p1.z) {}
    Vec(double i, double j, double k = 0): x(i), y(j), z(k) {}
    double x;
    double y;
    double z;

    bool operator<(const Vec &o){
        if(fabs(x - o.x) > EPS) return x < o.x;
        else if(fabs(y - o.y) > EPS) return y < o.y;
        else return z < o.z;
    }
    bool operator==(const Vec &o){
        return (fabs(x-o.x) < EPS) && (fabs(y-o.y) < EPS) && (fabs(z-o.z) < EPS);
    }
};
Vec operator+(const Vec &a, const Vec &b){
    return Vec(a.x+b.x, a.y+b.y, a.z+b.z);
}
Vec operator-(const Vec &a, const Vec &b){
    return Vec(a.x-b.x, a.y-b.y, a.z-b.z);
}
Vec operator*(const double d, Vec b){
    return Vec(d*b.x, d*b.y, d*b.z);
}

struct Line{
    Line(Vec i, Vec j): b(i), d(j) {}
    Vec b; //base
    Vec d; //direction
};
Line fromPoints(Vec i, Vec j){
    return Line(i, Vec(i, j));
}

double dot(Vec a, Vec b){ //dot product
    return a.x*b.x+a.y*b.y+a.z*b.z;
}
double cross(Vec a, Vec b){ //cross product (2D)
    return a.x*b.y-a.y*b.x;
}
Vec cross_vec(Vec a, Vec b){
    return Vec(a.y*b.z-b.y*a.z, a.z*b.x-b.z*a.x, a.x*b.y-a.y*b.x);
}
double len_sq(Vec a){ //give the squared length of a vector
    return dot(a, a);
}
double len(Vec a){ //squares the squared length
    return std::sqrt(len_sq(a));
}

double angle(Vec p, Line l){ //return the angle between the line and the point
    Vec c(l.b, p);
    return acos(dot(c, l.d)/std::sqrt(len_sq(c)*len_sq(l.d)));
}
double ccw(Vec p, Line l){ //return true if p is left of l
    return cross(Vec(l.b, p), l.d) < 0;
}

bool colinear(Line l, Vec p){ //check if a point is on a line (NOT segment)
    return fabs(cross(Vec(l.b, p), l.d)) < EPS;
}
bool in_segment(Line l, Vec p){ //check if in segment (NOT if on line)
    Vec b = l.b;
    Vec e = l.b+l.d;
    return (std::min(b.x,e.x)-EPS <= p.x && p.x <= std::max(b.x,e.x)+EPS) &&
           (std::min(b.y,e.y)-EPS <= p.y && p.y <= std::max(b.y,e.y)+EPS);
}
```

6.2 Distances

O(1)

```
double dist(Vec p1, Vec p2){
    return hypot(p1.x-p2.x, p1.y-p2.y);
}

double dist(Vec p, Line l){
    Vec c(l.b, p);
    double u = dot(c, l.d)/len_sq(l.d);
    if(u < 0.0) return dist(l.b, p);
    else if(u > 1.0) return dist(l.b+l.d, p);
    return dist(p, l.b+u*l.d);
}
```

6.3 Intersect

O(1)

```
std::pair<int, Vec> intersect(Line l1, Line l2){ //0 = non intersecting, 1 = intersecting, 2 = overlapping
    //check if single point on line
    if(l1.d == Vec(0, 0)) return std::make_pair(in_segment(l2, l1.b) && colinear(l2, l1.b), l1.b);
    if(l2.d == Vec(0, 0)) return std::make_pair(in_segment(l1, l2.b) && colinear(l1, l2.b), l2.b);

    Vec v = Vec(l1.b, l2.b);
    double c = cross(l1.d, l2.d);
    if(fabs(c) < EPS){
        if (fabs(cross(v, l1.d)) > EPS) return std::make_pair(0, Vec()); //parallel
        else{
            if(in_segment(l2, l1.b)) return std::make_pair(2, l1.b); //colinear and overlapping
            if(in_segment(l2, l1.b+l1.d)) return std::make_pair(2, l1.b+l1.d); //colinear and overlapping
            if(in_segment(l1, l2.b)) return std::make_pair(2, l2.b); //colinear and overlapping
            if(in_segment(l1, l2.b+l2.d)) return std::make_pair(2, l2.b+l2.d); //colinear and overlapping
            return std::make_pair(0, Vec()); //colinear but not overlapping
        }
    }else{
        double t = cross(v, l1.d)/c;
        double u = cross(v, l2.d)/c;
        if(-EPS <= t && t <= 1+EPS && -EPS <= u && u <= 1+EPS) return std::make_pair(1, l1.b+u*l1.d); //intersects
        else return std::make_pair(0, Vec()); //not intersecting
    }
}
```

6.4 Area

O(N)

```
//returns signed area
double areaPolygon(const std::vector<Vec> &v){ //ADD FIRST POINT AGAIN!
    double res = 0.0;
    for(size_t i=0; i<v.size()-1; ++i){
        res += v[i].x*v[i+1].y-v[i+1].x*v[i].y;
    }
    return res/2.0;
}

double areaCircle(double ab, double bc, double ca){
    double pm = ab+bc+ca;
    double sp = pm/2;
    return std::sqrt(sp*(sp-ab)*(sp-bc)*(sp-ca));
}
```

6.5 Miscellaneous

O(N)

```
/* TEST THESE */

bool inPolygon(Vec pt, const std::vector<Vec> &v){ //ADD FIRST POINT AGAIN!
    double sum = 0.0;
    for(size_t i=0; i<v.size()-1; ++i){
        if(ccw(v[i+1], fromPoints(pt, v[i]))){ //extend this to handle border
            sum += angle(v[i+1], fromPoints(pt, v[i]));
        }else sum -= angle(v[i+1], fromPoints(pt, v[i]));
    }
    return fabs(fabs(sum) - 2*PI) < EPS;
}

Vec centroid(const std::vector<Vec> &v){ //ADD FIRST POINT AGAIN!
    Vec ans;
    for(size_t i=0; i<v.size()-1; ++i){
        double spc = v[i].x*v[i+1].y-v[i+1].x*v[i].y;
        ans.x += (v[i].x+v[i+1].x)*spc;
        ans.y += (v[i].y+v[i+1].y)*spc;
    }
    ans = (1/(6.0*areaPolygon(v)))*ans;
    return ans;
}

double radiusInCircle(double ab, double bc, double ca){
    double sp = 0.5*(ab+bc+ca);
    return areaCircle(ab, bc, ca)/sp;
}

double radiusCircumCircle(double ab, double bc, double ca){
    return ab*bc*ca/(4.0*areaCircle(ab, bc, ca));
}

std::pair<int, Vec> inCircleTriangle(Vec pa, Vec pb, Vec pc){ //returns radius and center
    double r = radiusInCircle(len(Vec(pa, pb)), len(Vec(pb, pc)), len(Vec(pa, pc)));
    if(fabs(r) < EPS) return std::make_pair(r, Vec());

    double ratio = len(Vec(pa, pb))/len(Vec(pa, pc));
    Vec pt = pb + (ratio/(1+ratio))*Vec(pb, pc);
    Line l1 = fromPoints(pa, pt); //check this line

    ratio = len(Vec(pb, pa))/len(Vec(pb, pc));
    pt = pa + (ratio/(1+ratio))*Vec(pa, pc);
    Line l2 = fromPoints(pb, pt); //check this line

    return std::make_pair(r, intersect(l1, l2).second);
}
```

6.6 Convex-hull

O(N log(N))

```
Vec pivot(0, 0); //will contain the point that is used as pivot

bool angle_cmp(Vec p1, Vec p2){
    Line l = Line(pivot, Vec(pivot, p2));
    if(colinear(l, p1)) return dist(pivot, p1) < dist(pivot, p2);
    return !ccw(p1, l);
}

std::vector<Vec> convex_hull(std::vector<Vec> v){ //DONT ADD FIRST POINT AGAIN
    std::vector<Vec> ans;
    if(v.size() <= 3) return v;

    int s = 0;
    for(size_t i=1; i<v.size(); ++i){
        if(v[i].y+EPS < v[s].y || (fabs(v[i].y - v[s].y) < EPS && v[i].x < v[s].x)) s = i;
    }

    std::swap(v[0], v[s]);
    pivot = v[0];
    sort(++v.begin(), v.end(), angle_cmp);

    ans.push_back(v.back()); ans.push_back(v[0]); ans.push_back(v[1]);
    size_t i = 2;
    while(i < v.size()){
        size_t j = ans.size()-1;
        if(ccw(ans[j-1], Line(ans[j], v[i]))) ans.push_back(v[i++]);
        else ans.pop_back();
    }
    return ans;
}
```

7 String

7.1 Edit distance

O(N^2)

```
int dp[MAXN][MAXN]; //edit distance dp table
inline int edit_distance(std::string a, std::string b){
    dp[0][0] = 0;
    for(size_t i=1; i<=a.size(); ++i) dp[i][0] = i;
    for(size_t j=1; j<=b.size(); ++j) dp[0][j] = j;
    for(size_t i=1; i<=a.size(); ++i){
        for(size_t j=1; j<=b.size(); ++j){
            dp[i][j] = std::min(dp[i][j-1], dp[i-1][j])+1; //add character: +1
            if(a[i-1] == b[j-1]) dp[i][j] = std::min(dp[i][j], dp[i-1][j-1]); //same character: no cost
            else dp[i][j] = std::min(dp[i][j], dp[i-1][j-1]+1); //replace character +1
        }
    }
    return dp[a.size()][b.size()];
}
```

7.2 KMP

O(N)

```
int bt[MAXN]; //back table created preprocessing pattern
void kmpPreprocess(std::string P) {
    size_t i = 0, j = -1; bt[0] = -1;
    while (i < P.size()) {
        while (j >= 0 && P[i] != P[j]) j = bt[j];
        i++; j++;
        bt[i] = j;
    }
}

void kmpSearch(std::string T, std::string P) {
    kmpPreprocess(P); //preprocess first

    size_t i = 0, j = 0;
    while (i < T.size()) {
        while (j >= 0 && T[i] != P[j]) j = bt[j];
        i++; j++;
        if (j == P.size()) {
            //pattern is found in text at index i-j -- DO STUFF HERE
            j = bt[j];
        }
    }
}
```

7.3 Suffix array

O(N log(N)^2)

```
//MAXN should be 2x normal
int SA[MAXN];

int RA[MAXN];
int tempRA[MAXN];
int SK;
bool cmp(int a, int b){
    if(RA[a] == RA[b]) return RA[a+SK] < RA[b+SK];
    else return RA[a] < RA[b];
}

void constructSA(std::string T){
    T += '$';
    int N = T.size();
    for(int i=0; i<2*N; ++i) {
        if(i < N) RA[i] = T[i];
        else RA[i] = 0;
        SA[i] = i;
    }
    int r = 0;
    for(SK=1; SK<N; SK<<=1){
        std::sort(SA, SA+N, cmp); //can also use 2x counting sort

        tempRA[SA[0]] = r = 0;
        for(int i=0; i<N; ++i){
            if(RA[SA[i]] != RA[SA[i-1]] || RA[SA[i]+SK] != RA[SA[i-1]+SK]) ++r;
            tempRA[SA[i]] = r;
        }

        for(int i=0; i<N; ++i) RA[i] = tempRA[i];
        if(RA[SA[N-1]] == N-1) break;
    }
}
```

7.4 Longest common subsequence

O(N)

```
int LCP[MAXN]; //longest common prefix for suffix at position i

int PLCP[MAXN];
int PHI[MAXN];
void constructLCP(std::string T){
    T += '$';
    int N = T.size();

    PHI[SA[0]] = -1;
    for(int i=1; i<N; ++i) PHI[SA[i]] = SA[i-1];

    int L = 0;
    for(int i=0; i<N; ++i){
        if(PHI[i] == -1) {
            PLCP[i] = 0;
            continue;
        }
        while(T[i+L] == T[PHI[i] + L]) L++;
        PLCP[i] = L;
        L = std::max(L-1, 0);
    }
    for(int i=0; i<N; ++i) LCP[i] = PLCP[SA[i]];
}
```

8 Other

8.1 Binary search (integer)

O(log(N))

```
bool can(double f){
    //add implementation here
    return true;
}

```

```
int binsearch(int lo, int hi){
    while(lo < hi){
        int mid = (lo+hi)/2;

        if(can(mid)) lo = mid+1;
        else hi = mid;
    }
    return lo;
}
```

8.2 Binary search (floating-point)

O(log(N))

```
double binsearch(double lo, double hi){
    double mid, ans;
    while(std::fabs(hi-lo) > EPS){
        mid = (lo+hi)/2.0;
        if(can(mid)) {
            hi = mid;
            ans = mid;
        }else lo = mid;
    }
    return ans;
}
```

8.3 Counting sort

O(N)

```
const int MAXNUM = 100;
int ind[MAXNUM]; //contains new index after sort
int tmp[MAXN]; //temporary sort array

void counting_sort(std::vector<int> arr) { //stable counting sort
    for(int i=0; i<MAXNUM; ++i) ind[i] = 0;

    for(size_t i=0; i<arr.size(); ++i){
        ind[arr[i]]++;
    }
    int sum = 0;
    for(int i=0; i<MAXNUM; ++i){
        int t = ind[i];
        ind[i] = sum;
        sum += t;
    }
    for(size_t i=0; i<arr.size(); ++i){
        tmp[ind[arr[i]]++] = arr[i];
    }
    for(size_t i=0; i<arr.size(); ++i){
        arr[i] = tmp[i];
    }
}
```

8.4 Generate permutations

O(N^K)

```
std::list<std::vector<int> > gen_permutation(std::vector<int> &opts, int sz){
    std::list<std::vector<int> > all;
    if(sz == 0){
        for(size_t i=0; i<opts.size(); ++i) all.push_back({opts[i]});
        return all;
    }
    for(size_t i=0; i<opts.size(); ++i){
        std::list<std::vector<int> > lst = gen_permutation(opts, sz-1);
        for(auto iter = lst.begin(); iter != lst.end(); ++iter) iter->push_back(opts[i]);
        all.insert(all.end(), lst.begin(), lst.end());
    }
    return all;
}
```

8.5 Generate combinations

O($N\binom{N}{K}$)

```
std::list<std::vector<int> > gen_combinations(std::vector<int> &opts, int k){
    std::vector<bool> has(opts.size());
    std::fill(has.begin() + k, has.end(), true);

    std::list<std::vector<int> > all;
    do {
        std::vector<int> comb;
        for (size_t i = 0; i < has.size(); ++i) {
            if (!has[i]) comb.push_back(opts[i]);
        }
        all.push_back(comb);
    } while (std::next_permutation(has.begin(), has.end()));

    return all;
}
```

8.6 LIS

O(N log(N))

```
int ldp[MAXN]; //lis dp array
int lin[MAXN]; //lin[inp.size()-1] contains last index
int frm[MAXN]; //contains index number came from
int lis(std::vector<int> inp){ //non-decreasing lis
    for(int i=0; i<MAXN; ++i) ldp[i] = INT_MAX;

    for(size_t i=0; i<inp.size(); ++i){
        int k = std::upper_bound(ldp, ldp+MAXN, inp[i])-ldp;
        ldp[k] = inp[i];
        lin[k] = i;
        if(k == 0) frm[i] = -1;
        else frm[i] = lin[k-1];
    }

    return std::upper_bound(ldp, ldp+MAXN, INT_MAX-1)-ldp;
}
```