# Contents

# 1 src/datastructure/fastmap.cpp

```cpp
// Implements map operations for keys known in construction
// Undefined behavior when key doesn't exist
// O(n log n) construction and O(log n) access
#include <bits/stdc++.h>
using namespace std;

template<typename keyT, typename valueT>
struct FastMap {
	vector<keyT> keys;
	vector<valueT> values;

	FastMap(const vector<keyT>&ks) : keys(ks), values(ks.size()) {
		sort(keys.begin(), keys.end());
	}

	valueT& operator[](keyT key) {
		auto it=lower_bound(keys.begin(), keys.end(), key);
		return values[it-keys.begin()];
	}
};
```

## 2 src/datastructure/orderedset.cpp

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/detail/standard_policies.hpp>
using namespace std;
using namespace __gnu_pbds;
//using namespace pb_ds;

typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;

int main() {
        ordered_set X;
        X.insert(1);
        X.insert(4);
        cout<<*X.find_by_order(1)<<endl; // 4
        cout<<X.order_of_key(3)<<endl;   // 1
}
```

## 3 src/datastructure/treap.cpp

```
// Treap implementation with pointers
// Expected running time of split and merge is O(log n)
#include <bits/stdc++.h>
using namespace std;

typedef struct node* pnode;
struct node {
        pnode l,r;
        int pr,c;
        node() {
                l=0;
                r=0;
                c=1;
                pr=rand();
        }
};

// Returns the size of the subtree t
int cnt(pnode t) {
        if (t) return t->c;
        return 0;
}

// Updates the size of the subtree t
void upd(pnode t) {
        if (t) t->c=cnt(t->l)+cnt(t->r)+1;
}

// Put lazy updates here
void push(pnode t) {
        if (t) {
                // Something
        }
}

// Merges trees l and r into tree t
void merg(pnode& t, pnode l, pnode r) {
        push(l);
        push(r);
        if (!l) t=r;
        else if(!r) t=l;
        else {
                if (l->pr>r->pr) {
                        merg(l->r, l->r, r);
                        t=l;
                }
                else {
                        merg(r->l, l, r->l);
                        t=r;
                }
```

```
        }
        upd(t);
}

// Splits tree t into trees l and r
// Size of tree l will be k
void split(pnode t, pnode& l, pnode& r, int k) {
        if (!t) {
                l=0;
                r=0;
                return;
        }
        else {
                push(t);
                if (cnt(t->l)>=k) {
                        split(t->l, l, t->l, k);
                        r=t;
                }
                else {
                        split(t->r, t->r, r, k-cnt(t->l)-1);
                        l=t;
                }
        }
        upd(t);
}
```

## 4 src/general.cpp

```cpp
// Standard
#include <bits/stdc++.h>
#define F first
#define S second
typedef long long ll;
typedef __int128 lll;
typedef long double ld;
using namespace std;

// GCC extension namespaces
using namespace __gnu_pbds;
using namespace __gnu_cxx;

// Data structures
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

// Numeric
#include <ext/numeric>

int main(){
        // Fast I/O:
        ios_base::sync_with_stdio(0);
        cin.tie(0);
}
```

## 5 src/geom/anglesort.cpp

```cpp
// cp is comparasion function for sorting points around origin
// points are sorted in clockwise order
/*
122
143
443*/
#include <bits/stdc++.h>
#define X real()
#define Y imag()
#define F first
#define S second
using namespace std;
typedef long double ld;
typedef long long ll;
// Coordinate type
typedef ld CT;
typedef complex<CT> co;

bool ccw(co a, co b, co c) {
        return ((c-a)*conj(b-a)).Y>0;
}

int ar(co x) {
        if (x.Y>=0&&x.X<0) return 1;
        if (x.X>=0&&x.Y>0) return 2;
        if (x.Y<=0&&x.X>0) return 3;
        return 4;
}

bool cp(co p1, co p2) {
        if (ar(p1)!=ar(p2)) {
                return ar(p1)<ar(p2);
        }
        return ccw({0, 0}, p2, p1)>0;
}
```

## 6 src/geom/basic.cpp

```
// Basic geometry functions using complex numbers
// Mostly copied from https://github.com/ttalvitie/libcontest/
/* Useful functions of std
        CT abs(co x): Length
        CT norm(co x): Square of length
        CT arg(co x): Angle
        co polar(CT length, CT angle): Complex from polar components
*/
#include <bits/stdc++.h>
#define X real()
#define Y imag()
using namespace std;
typedef long double ld;
typedef long long ll;
// Coordinate type
typedef ld CT;
typedef complex<CT> co;

// Return true iff points a, b, c are CCW oriented.
bool ccw(co a, co b, co c) {
        return ((c-a)*conj(b-a)).Y>0;
}

// Return true iff points a, b, c are collinear.
// NOTE: doesn't make much sense with non-integer CT.
bool collinear(co a, co b, co c) {
        return ((c-a)*conj(b-a)).Y==0;
}

// Rotate x with agle ang
co rotate(co x, CT ang) {
        return x*polar((CT)1, ang);
}

// Check whether segments [a, b] and [c, d] intersect.
// The segments must not be collinear. Doesn't handle edge cases (endpoint of
// a segment on the other segment) consistently.
bool intersects(co a, co b, co c, co d) {
        return ccw(a, d, b)!=ccw(a, c, b)&&ccw(c, a, d)!=ccw(c, b, d);
}

// Interpolate between points a and b with parameter t.
co interpolate(CT t, co a, co b) {
        return a+t*(b-a);
}

//  Return interpolation parameter between a and b of projection of v to the
//  line defined by a and b.
//  NOTE: no rounding behavior specified for integers.
CT projectionParam(co v, co a, co b) {
```

```
        return ((v-a)/(b-a)).X;
}

//  Compute the distance of point v from line a..b.
//  NOTE: Only for non-integers!
CT pointLineDistance(co p, co a, co b) {
        return abs(((p-a)/(b-a)).Y)*abs(b-a);
}

//  Compute the distance of point v from segment a..b.
//  NOTE: Only for non-integers!
CT pointSegmentDistance(co p, co a, co b) {
        co z=(p-a)/(b-a);
        if(z.X<0) return abs(p-a);
        if(z.X>1) return abs(p-b);
        return abs(z.Y)*abs(b-a);
}

//  Return interpolation parameter between a and b of the point that is also
//  on line c..d.
//  NOTE: Only for non-integers!
CT intersectionParam(co a, co b, co c, co d) {
        co u=(c-a)/(b-a);
        co v=(d-a)/(b-a);
        return (u.X*v.Y-u.Y*v.X)/(v.Y-u.Y);
}
```

## 7 src/geom/convexhull.cpp

```
// Computes the convex hull of given set of points in O(n log n)
// Uses Andrew's algorithm
// The points on the edges of the hull are not listed
// Change > to >= in ccw function to list the points on the edges
// Returns points in counterclockwise order
#include <bits/stdc++.h>
#define X real()
#define Y imag()
using namespace std;
typedef long double ld;
typedef long long ll;
// Coordinate type
typedef ll CT;
typedef complex<CT> co;

bool ccw(co a, co b, co c) {
        return ((c-a)*conj(b-a)).Y>0;
}

vector<co> convexHull(vector<co> ps) {
        auto cmp = [](co a, co b) {
                if (a.X==b.X) return a.Y<b.Y;
                else return a.X<b.X;
        };
        sort(ps.begin(), ps.end(), cmp);
        ps.erase(unique(ps.begin(), ps.end()), ps.end());

        int n=ps.size();
        if (n<=2) return ps;

        vector<co> hull;
        hull.push_back(ps[0]);
        for (int d=0;d<2;d++) {
                if (d) reverse(ps.begin(), ps.end());
                int s=hull.size();
                for (int i=1;i<n;i++) {
                        while ((int)hull.size()>s&&!ccw(hull[hull.size()-2], hull.back(), ps[i]))
{
                                hull.pop_back();
                        }
                        hull.push_back(ps[i]);
                }
        }
        hull.pop_back();
        return hull;
}
```

## 8 src/geom/minkowskisum.cpp

```
// Computes the Minkowski sum of 2 convex polygons in O(n+m log n+m)
// Returns convex polygon in counterclockwise order
// The points on the edges of the hull are listed
// The convex hulls must be in counterclockwise order
#include <bits/stdc++.h>
#define X real()
#define Y imag()
#define F first
#define S second
using namespace std;
typedef long double ld;
typedef long long ll;
// Coordinate type
typedef ll CT;
typedef complex<CT> co;

bool ccw(co a, co b, co c) {
        return ((c-a)*conj(b-a)).Y>0;
}

int ar(co x) {
        if (x.Y>=0&&x.X<0) return 1;
        if (x.X>=0&&x.Y>0) return 2;
        if (x.Y<=0&&x.X>0) return 3;
        return 4;
}

bool cp(pair<co, pair<int, int> > p1, pair<co, pair<int, int> > p2) {
        if (ar(p1.F)!=ar(p2.F)) {
                return ar(p1.F)<ar(p2.F);
        }
        return ccw({0, 0}, p2.F, p1.F)>0;
}

vector<co> minkowski(vector<co>&a, vector<co>&b) {
        int n=a.size();
        int m=b.size();
        if (n==0) return b;
        if (m==0) return a;
        if (n==1) {
                vector<co> ret(m);
                for (int i=0;i<m;i++) {
                        ret[i]=b[i]+a[0];
                }
                return ret;
        }
        if (m==1) {
                vector<co> ret(n);
                for (int i=0;i<n;i++) {
                        ret[i]=a[i]+b[0];
```

```
		}
		return ret;
	}
	vector<pair<co, pair<int, int> > > pp;
	for (int i=0;i<n;i++) {
		pp.push_back({a[(i+1)%n]-a[i], {1, i}});
	}
	for (int i=0;i<m;i++) {
		pp.push_back({b[(i+1)%m]-b[i], {2, i}});
	}
	sort(pp.rbegin(), pp.rend(), cp);
	co s={0, 0};
	co ad={0, 0};
	for (int i=0;i<(int)pp.size();i++) {
		s+=pp[i].F;
		if (pp[i].S.F!=pp[i+1].S.F) {
			if (pp[i].S.F==1) ad=a[(pp[i].S.S+1)%n]+b[(pp[i+1].S.S)%m];
			else ad=b[(pp[i].S.S+1)%m]+a[(pp[i+1].S.S)%n];
			ad-=s;
			break;
		}
	}
	s=ad;
	vector<co> ret(pp.size());
	for (int i=0;i<(int)pp.size();i++) {
		ret[i]=s;
		s+=pp[i].F;
	}
	return ret;
}
```

## 9 src/graph/dynamicconnectivity.cpp

```
// O(n log n) offline solution for dynamic connectivity problem
// ? count the number of connected components
// + A B add edge between A and B
// - A B remove edge between A and B
#include <bits/stdc++.h>
#define F first
#define S second
using namespace std;

struct e{
        int a,b,l,r;
};
int qqs[603030];
int qv[603030];
int is[603030];
int uf[603030];
int id[603030];

int getu(int a){
        if (uf[a]==a) return a;
        return uf[a]=getu(uf[a]);
}

void un(int a, int b){
        a=getu(a);
        b=getu(b);
        if (a!=b) uf[a]=b;
}

void go(int l, int r, int uc, int n, vector<e> es){
        for (int i=1;i<=n;i++){
                is[i]=0;
        }
        int i2=1;
        vector<pair<int, int> > te;
        vector<e> ce;
        for (e ee:es){
                if (ee.a!=ee.b&&(!(ee.l>r||ee.r<l))){
                        if (is[ee.a]==0){
                                is[ee.a]=i2;
                                ee.a=i2++;
                        }
                        else{
                                ee.a=is[ee.a];
                        }
                        if (is[ee.b]==0){
                                is[ee.b]=i2;
                                ee.b=i2++;
                        }
                        else{
```

```
                        ee.b=is[ee.b];
                }
                if (ee.l<=l&&r<=ee.r){
                        te.push_back({ee.a, ee.b});
                }
                else{
                        ce.push_back(ee);
                }
        }
    }
    for (int i=1;i<=n;i++){
        if (is[i]==0){
                uc++;
        }
    }
    for (int i=1;i<i2;i++){
        uf[i]=i;
        id[i]=0;
    }
    for (auto ee:te){
        un(ee.F, ee.S);
    }
    int i3=1;
    for (int i=1;i<i2;i++){
        if (id[getu(uf[i])]==0){
                id[getu(uf[i])]=i3++;
        }
    }
    for (e&ee:ce){
        ee.a=id[getu(ee.a)];
        ee.b=id[getu(ee.b)];
    }
    if (l==r){
        qv[l]=uc+i3-1;
    }
    else{
        int m=(l+r)/2;
        go(l, m, uc, i3-1, ce);
        go(m+1, r, uc, i3-1, ce);
    }
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    int n,k;
    cin>>n>>k;
    int qs=0;
    vector<e> es;
    map<pair<int, int>, int> ae;
    for (int i=1;i<=k;i++){
        char t;
```

```
                cin>>t;
                if (t=='?'){
                        qqs[qs++]=i;
                }
                else{
                        int a,b;
                        cin>>a>>b;
                        if (t=='+'){
                                pair<int, int> lol={min(a, b), max(a, b)};
                                ae[lol]=i;
                        }
                        else{
                                pair<int, int> lol={min(a, b), max(a, b)};
                                int s=ae[lol];
                                ae[lol]=0;
                                es.push_back({a, b, s, i});
                        }
                }
        }
        for (auto t:ae){
                if (t.S>0){
                        es.push_back({t.F.F, t.F.S, t.S, k});
                }
        }
        go(0, (1<<19)-1, 0, n, es);
        for (int i=0;i<qs;i++){
                cout<<qv[qqs[i]]<<'\n';
        }
}
```

## 10 src/graph/edmondskarp.cpp

```cpp
// Edmonds Karp algorithm for maxflow O(V E^2) or O(f E)
// f is the capacity network and the actual flow can be found in it
// If edges for both directions are used finding actual flow is harder
// Uses 1-indexing

#include <bits/stdc++.h>
#define F first
#define S second
using namespace std;
typedef long long ll;

const int inf=2e9;

struct maxFlow{

	vector<vector<int> > f;
	vector<vector<int> > g;
	vector<int> fr;
	vector<int> used;

	int flow(int so, int si, int n) {
		queue<pair<pair<int, int>, int> > bfs;
		bfs.push({{0, so}, inf});
		int fl=0;
		while(!bfs.empty()){
			auto x=bfs.front();
			bfs.pop();
			if (used[x.F.S]) continue;
			used[x.F.S]=1;
			fr[x.F.S]=x.F.F;
			if (x.F.S==si){
				fl=x.S;
				break;
			}
			for (int nx:g[x.F.S]){
				if (f[x.F.S][nx]>0){
					bfs.push({{x.F.S, nx}, min(x.S, f[x.F.S][nx])});
				}
			}
		}
		for (int i=1;i<=n;i++) used[i]=0;
		if (fl>0){
			int x=si;
			while (fr[x]>0){
				f[x][fr[x]]+=fl;
				f[fr[x]][x]-=fl;
				x=fr[x];
			}
			return fl;
		}
```

```
        return 0;
    }

    ll getMaxFlow(int source, int sink){
        int n=fr.size()-1;
        for (int i=1;i<=n;i++){
            g[i].clear();
            for (int ii=1;ii<=n;ii++){
                if (f[i][ii]!=0||f[ii][i]!=0){
                    g[i].push_back(ii);
                }
            }
        }
        ll r=0;
        while (1){
            int fl=flow(source, sink, n);
            if (fl==0) break;
            r+=(ll)fl;
        }
        return r;
    }

    void addEdge(int a, int b, int c){
        f[a][b]=c;
    }

    maxFlow(int n) : f(n+1), g(n+1), fr(n+1), used(n+1) {
        for (int i=1;i<=n;i++){
            f[i]=vector<int>(n+1);
        }
    }
};
```

## 11 src/graph/eulertour.cpp

```
// Finds Euler tour of graph in O(E) time

// Parameters are the adjacency list, number of nodes,
// return value vector, and d=1 if the graph is directed
// Return array contains E+1 elements, the first and last
// elements are same

// Undefined behavior if Euler tour doesn't exist

// Note that Eulerian path can be reduced to Euler tour
// by adding an edge from the last vertex to the first

// In bidirectional graph edges must be in both direction
// Be careful to not add loops twice in case of bidirectional graph
#include <bits/stdc++.h>
#define F first
#define S second
using namespace std;

struct EulerTour {
        int dir;
        vector<vector<pair<int, int> > > g;
        vector<int> used;

        void dfs(int x, vector<int>& ret) {
                int t=x;
                vector<int> c;
                while (1) {
                        while (used[g[t].back().S]) g[t].pop_back();
                        auto nx=g[t].back();
                        g[t].pop_back();
                        used[nx.S]=1;
                        t=nx.F;
                        c.push_back(t);
                        if (t==x) break;
                }
                for (int a:c) {
                        ret.push_back(a);
                        while (g[a].size()>0&&used[g[a].back().S]) g[a].pop_back();
                        if (g[a].size()>0) dfs(a, ret);
                }
        }

        EulerTour(vector<int>* og, int n, vector<int>& ret, int d=0) : dir(d), g(n+1) {
                int i2=0;
                for (int i=1;i<=n;i++) {
                        for (int nx:og[i]) {
                                if (d==1||nx<=i) {
                                        if (d==0&&nx<i) {
                                                g[nx].push_back({i, i2});
```

```
                                        }
                                        g[i].push_back({nx, i2++});
                                }
                        }
                }
                used.resize(i2);
                for (int i=1;i<=n;i++) {
                        if (g[i].size()>0) {
                                ret.push_back(i);
                                dfs(i, ret);
                                break;
                        }
                }
        }
};
```

## 12 src/graph/mincostflow.cpp

```
// Finds minimum-cost k-flow
// O(V E^2 log U), where U is maximum possible flow
// Finding augmenting path is O(V E), usually faster
// Uses scaling flow and finds augmenting path with SPFA
// Only 1-directional edges allowed
// Doesn't work if graph contains negative cost cycles
// Uses 1-indexing

#include <bits/stdc++.h>
#define F first
#define S second
using namespace std;
typedef long long ll;
typedef long double ld;

const ll inf=1e18;

struct minCostFlow {
	vector<vector<ll> > f;
	vector<vector<int> > g;
	vector<vector<ll> > c;
	vector<ll> d;
	vector<int> from;
	vector<int> inq;
	queue<int> spfa;

	void relax(int x, ll di, int p) {
		if (di>=d[x]) return;
		d[x]=di;
		from[x]=p;
		if (!inq[x]) {
			spfa.push(x);
			inq[x]=1;
		}
	}

	ll augment(ll x, ll s, ll fl) {
		if (x==s) return fl;
		ll r=augment(from[x], s, min(fl, f[from[x]][x]));
		f[from[x]][x]-=r;
		f[x][from[x]]+=r;
		return r;
	}

	pair<ll, ll> flow(int s, int t, ll miv, ll kf) {
		int n=g.size()-1;
		for (int i=1;i<=n;i++) {
			d[i]=inf;
			inq[i]=0;
		}
```

```
        relax(s, 0, 0);
        while (!spfa.empty()) {
                int x=spfa.front();
                spfa.pop();
                inq[x]=0;
                for (int nx:g[x]) {
                        if (f[x][nx]>=miv) relax(nx, d[x]+c[x][nx], x);
                }
        }
        if (d[t]<inf) {
                ll fl=augment(t, s, kf);
                return {fl, fl*d[t]};
        }
        return {0, 0};
}

// maxv is maximum possible flow on a single augmenting path
// kf is inteded flow, set infinite for maxflow
// returns {flow, cost}
pair<ll, ll> getKFlow(int source, int sink, ll maxv, ll kf) {
        int n=g.size()-1;
        for (int i=1;i<=n;i++) {
                g[i].clear();
                for (int ii=1;ii<=n;ii++) {
                        if (f[i][ii]!=0||f[ii][i]!=0) g[i].push_back(ii);
                }
        }
        ll r=0;
        ll k=1;
        ll co=0;
        while (k*2<=maxv) k*=2;
        for (;k>0&&kf>0;k/=2) {
                while (1) {
                        pair<ll, ll> t=flow(source, sink, k, kf);
                        r+=t.F;
                        kf-=t.F;
                        co+=t.S;
                        if (kf==0||t.F==0) break;
                }
        }
        return {r, co};
}

void addEdge(int a, int b, ll capa, ll cost) {
        f[a][b]=capa;
        c[a][b]=cost;
        c[b][a]=-cost;
}

minCostFlow(int n) : f(n+1), g(n+1), c(n+1), d(n+1), from(n+1), inq(n+1) {
        for (int i=1;i<=n;i++) {
                f[i]=vector<ll>(n+1);
```

```
                        c[i]=vector<ll>(n+1);
                }
        }
};
```

## 13 src/graph/rootedtree.cpp

```cpp
// Build parent array of tree using O(n log n) space
// Query i:th parent in O(log n) time
// Query lca in O(log n) time
// Query distance in O(log n) time
// Uses 1-indexing
#include <bits/stdc++.h>
using namespace std;

// This has to be at least ceil(log2(n))
const int logSize=22;

struct RootedTree {

	vector<int> d;
	vector<array<int, logSize> > p;

	// Dfs for building parent array
	void dfs(vector<int>* g, int x, int pp, int dd) {
		p[x][0]=pp;
		for(int i=1;i<logSize;i++) {
			p[x][i]=p[p[x][i-1]][i-1];
		}
		d[x]=dd;
		for (int nx:g[x]) {
			if (nx!=pp){
				dfs(g, nx, x, dd+1);
			}
		}
	}

	// Construct parent array data structure of tree of size n
	// g is the adjacency list of the tree
	RootedTree(vector<int>* g, int n, int root=1) : d(n+1), p(n+1) {
		dfs(g, root, 0, 0);
	}

	// Returns the node h edges above x.
	// Returns 0 if no such node exists
	int parent(int x, int h) {
		for (int i=logSize-1;i>=0;i--) {
			if ((1<<i)&h) {
				x=p[x][i];
			}
		}
		return x;
	}

	// Returns lca of nodes a and b
	int lca(int a, int b) {
		if (d[a]<d[b]) swap(a, b);
```

```
                a=parent(a, d[a]-d[b]);
                if (a==b) return a;
                for (int i=logSize-1;i>=0;i--) {
                        if (p[a][i]!=p[b][i]) {
                                a=p[a][i];
                                b=p[b][i];
                        }
                }
                return p[a][0];
        }

        // Returns distance from a to b
        int dist(int a, int b) {
                int l=lca(a, b);
                return d[a]+d[b]-2*d[l];
        }
};
```

## 14 src/graph/scalingflow.cpp

```cpp
// Scaling flow algorithm for maxflow
// O(E^2 log U), where U is maximum possible flow
// In practice O(E^2)
// Uses 1-indexing

#include <bits/stdc++.h>
#define F first
#define S second
using namespace std;
typedef long long ll;

struct maxFlow {
        vector<vector<ll> > f;
        vector<vector<int> > g;
        vector<int> used;
        int cc;

        ll flow(int x, int t, ll fl, ll miv) {
                if (x==t) return fl;
                used[x]=cc;
                for (int nx:g[x]) {
                        if (used[nx]!=cc&&f[x][nx]>=miv) {
                                ll r=flow(nx, t, min(fl, f[x][nx]), miv);
                                if (r>0) {
                                        f[x][nx]-=r;
                                        f[nx][x]+=r;
                                        return r;
                                }
                        }
                }
                return 0;
        }

        // maxv is maximum expected maxflow
        ll getMaxFlow(int source, int sink, ll maxv) {
                int n=g.size()-1;
                for (int i=1;i<=n;i++) {
                        g[i].clear();
                        for (int ii=1;ii<=n;ii++) {
                                if (f[i][ii]!=0||f[ii][i]!=0) g[i].push_back(ii);
                        }
                }
                cc=1;
                ll r=0;
                ll k=1;
                while (k*2<=maxv) k*=2;
                for (;k>0;k/=2) {
                        while (ll t=flow(source, sink, maxv, k)) {
                                r+=t;
                                cc++;
```

```
                }
                cc++;
            }
            return r;
        }

        void addEdge(int a, int b, ll c) {
            f[a][b]+=c;
        }

        maxFlow(int n) : f(n+1), g(n+1), used(n+1) {
            for (int i=1;i<=n;i++) {
                f[i]=vector<ll>(n+1);
            }
        }
};
```

## 15 src/graph/stronglyconnected.cpp

```cpp
// Uses Kosaraju's algorithm O(V+E)
// Components will be returned in topological order
// Uses 1-indexing
#include <bits/stdc++.h>
using namespace std;

struct SCC{
	vector<int> used;
	vector<vector<int> > g2;

	// First dfs
	void dfs1(vector<int>* g, int x, vector<int>& ns) {
		if (used[x]==1) return;
		used[x]=1;
		for (int nx:g[x]) {
			g2[nx].push_back(x);
			dfs1(g, nx, ns);
		}
		ns.push_back(x);
	}

	// Second dfs
	void dfs2(int x, vector<int>& co) {
		if (used[x]==2) return;
		used[x]=2;
		co.push_back(x);
		for (int nx:g2[x]) {
			dfs2(nx, co);
		}
	}

	// Returns strongly connected components of the graph in vector ret
	// n is the size of the graph, g is the adjacency list
	SCC(vector<int>* g, int n, vector<vector<int> >& ret) : used(n+1), g2(n+1) {
		vector<int> ns;
		for (int i=1;i<=n;i++) {
			dfs1(g, i, ns);
		}
		for (int i=n-1;i>=0;i--) {
			if (used[ns[i]]!=2) {
				ret.push_back(vector<int>());
				dfs2(ns[i], ret.back());
			}
		}
	}
};
```

## 16 src/graph/unionfind.cpp

```cpp
// Fast union find
// Uses 1-indexing
#include <bits/stdc++.h>
using namespace std;

struct unionFind {
	vector<int> u;
	vector<int> us;

	// Construct union find data structure of n vertices
	unionFind(int n) : u(n+1), us(n+1) {
		for (int i=1;i<=n;i++) {
			u[i]=i;
			us[i]=1;
		}
	}

	// Get the union of x
	int get(int x) {
		if (x==u[x]) return x;
		return u[x]=get(u[x]);
	}

	// Union a and b
	void un(int a, int b) {
		a=get(a);
		b=get(b);
		if (a!=b) {
			if (us[a]<us[b]) swap(a, b);
			us[a]+=us[b];
			u[b]=a;
		}
	}
};
```

## 17 src/math/crt.cpp

```cpp
// Solves x from system of equations x == a_i (mod p_i)
// Overflows only if p_1*...*p_n overflows
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

ll pot(ll x, ll p, ll mod) {
	if (p==0) return 1;
	if (p%2==0) {
		x=pot(x, p/2, mod);
		return (x*x)%mod;
	}
	return (x*pot(x, p-1, mod))%mod;
}

ll inv(ll x, ll mod) {
	return pot(x%mod, mod-2, mod);
}

ll solve(vector<ll> a, vector<ll> p) {
	vector<ll> x(a.size());
	ll r=0;
	ll k=1;
	for (int i=0;i<(int)a.size();i++) {
		x[i]=a[i];
		for (int j=0;j<i;j++) {
			x[i]=inv(p[j], p[i])*(x[i]-x[j]);
			x[i]=x[i]%p[i];
			if (x[i]<0) x[i]+=p[i];
		}
		r+=k*x[i];
		k*=p[i];
	}
	return r;
}
```

## 18 src/math/diophantine.cpp

```cpp
// solves ax+by=c in O(log a+b) time
// returns {is, {x, y}}, is=0 if there is no solution
// use __int128 for 64 bit numbers
#include <bits/stdc++.h>
#define F first
#define S second
using namespace std;
typedef long long ll;

ll ee(ll a, ll b, ll ca, ll cb, ll xa, ll xb, ll&x, ll&y) {
        if (cb==0) {
                x=xa;
                if (b==0) y=0;
                else y=(ca-a*xa)/b;
                return ca;
        }
        else return ee(a, b, cb, ca%cb, xb, xa-(ca/cb)*xb, x, y);
}

pair<int, pair<ll, ll> > solve(ll a, ll b, ll c) {
        if (c==0) return {1, {0, 0}};
        if (a==0&&b==0) return {0, {0, 0}};
        ll x,y;
        ll g=ee(a, b, a, b, 1, 0, x, y);
        if (abs(c)%g>0) return {0, {0, 0}};
        return {1, {x*(c/g), y*(c/g)}};
}
```

## 19 src/math/fft.cpp

```
// Fast Fourier transform and convolution using it
// O(n log n)
// Source: http://cses.fi/kkkk.pdf
#include <bits/stdc++.h>
using namespace std;
typedef long double ld;
typedef long long ll;
typedef complex<ld> co;
const ld PI=atan2(0, -1);

vector<co> fft(vector<co> x, int d) {
        int n=x.size();
        for (int i=0;i<n;i++) {
                int u=0;
                for (int j=1;j<n;j*=2) {
                        u*=2;
                        if (i&j) u++;
                }
                if (i<u) {
                        swap(x[i], x[u]);
                }
        }
        for (int m=2;m<=n;m*=2) {
                co wm=exp(co{0, d*2*PI/m});
                for (int k=0;k<n;k+=m) {
                        co w=1;
                        for (int j=0;j<m/2;j++) {
                                co t=w*x[k+j+m/2];
                                co u=x[k+j];
                                x[k+j]=u+t;
                                x[k+j+m/2]=u-t;
                                w*=wm;
                        }
                }
        }
        if (d==-1) {
                for (int i=0;i<n;i++) {
                        x[i]/=n;
                }
        }
        return x;
}

vector<ll> conv(vector<ll> a, vector<ll> b) {
        int as=a.size();
        int bs=b.size();
        vector<co> aa(as);
        vector<co> bb(bs);
        for (int i=0;i<as;i++) {
                aa[i]=a[i];
```

```
        }
        for (int i=0;i<bs;i++) {
                bb[i]=b[i];
        }
        int n=1;
        while (n<as+bs-1) n*=2;
        aa.resize(n*2);
        bb.resize(n*2);
        aa=fft(aa, 1);
        bb=fft(bb, 1);
        vector<co> c(2*n);
        for (int i=0;i<2*n;i++) {
                c[i]=aa[i]*bb[i];
        }
        c=fft(c, -1);
        c.resize(as+bs-1);
        vector<ll> r(as+bs-1);
        for (int i=0;i<as+bs-1;i++){
                r[i]=(ll)round(c[i].real());
        }
        return r;
}

int main(){
        // Shoud print 12 11 30 7
        vector<ll> a={3, 2, 7};
        vector<ll> b={4, 1};
        vector<ll> c=conv(a, b);
        for (ll t:c){
                cout<<t<<endl;
        }
}
```

## 20 src/math/gaussjordan.cpp

```cpp
// Solves system of linear equations in O(n^3)
// Using doubles or mod 2
// Using doubles might have large precision errors or overflow
// Returns 0 if no solution exists, 1 if there is one solution
// or 2 if infinite number of solutions exists
// If at least one solution exists, it is returned in ans
// You can modify the general algorithm to work mod p by using modular inverse
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef long double ld;
const ld eps=1e-12;

// Using doubles
int gaussD (vector<vector<ld> > a, vector<ld>& ans) {
        int n=(int)a.size();
        int m=(int)a[0].size()-1;

        vector<int> where(m,-1);
        for (int col=0,row=0;col<m&&row<n;col++) {
                int sel=row;
                for (int i=row;i<n;i++) {
                        if (abs(a[i][col])>abs(a[sel][col])) sel=i;
                }
                if (abs(a[sel][col])<eps) continue;
                for (int i=col;i<=m;i++) {
                        swap (a[sel][i], a[row][i]);
                }
                where[col]=row;

                for (int i=0;i<n;i++) {
                        if (i!=row) {
                                ld c=a[i][col]/a[row][col];
                                for (int j=col;j<=m;j++) {
                                        a[i][j]-=a[row][j]*c;
                                }
                        }
                }
                row++;
        }

        ans.assign(m, 0);
        for (int i=0;i<m;i++) {
                if (where[i]!=-1) ans[i]=a[where[i]][m]/a[where[i]][i];
        }
        for (int i=0;i<n;i++) {
                ld sum=0;
                for (int j=0;j<m;j++) {
                        sum+=ans[j]*a[i][j];
                }
```

```
                if (abs(sum-a[i][m])>eps) return 0;
        }

        for (int i=0;i<m;i++) {
                if (where[i]==-1) return 2;
        }
        return 1;
}

// mod 2
// n is number of rows m is number of variables
const int M=4;
int gaussM(vector<bitset<M> > a, int n, int m, bitset<M-1>& ans) {
        vector<int> where (m, -1);
        for (int col=0,row=0;col<m&&row<n;col++) {
                for (int i=row;i<n;i++) {
                        if (a[i][col]) {
                                swap (a[i], a[row]);
                                break;
                        }
                }
                if (!a[row][col]) continue;
                where[col]=row;

                for (int i=0;i<n;i++) {
                        if (i!=row&&a[i][col]) {
                                a[i]^=a[row];
                        }
                }
                row++;
        }
        ans=0;
        for (int i=0;i<m;i++) {
                if (where[i]!=-1) ans[i]=a[where[i]][m];
        }
        for (int i=0;i<n;i++) {
                int sum=0;
                for (int j=0;j<m;j++) {
                        sum^=ans[j]*a[i][j];
                }
                if (sum!=a[i][m]){
                        return 0;
                }
        }
        for (int i=0;i<m;i++){
                if (where[i]==-1) return 2;
        }
        return 1;
}

int main() {
        // Should output 2, 1 2 0
```

```
    vector<vector<ld> > d(3);
    d[0]={3, 3, -15, 9};
    d[1]={1, 0, -2, 1};
    d[2]={2, -1, -1, 0};
    vector<ld> da;
    cout<<gaussD(d, da)<<endl;
    cout<<da[0]<<" "<<da[1]<<" "<<da[2]<<endl;

    // Should output 1, 110
    // Note that bitsets are printed in reverse order
    bitset<M> r1("0110");
    bitset<M> r2("1101");
    bitset<M> r3("0111");
    vector<bitset<M> > m={r1, r2, r3};
    bitset<M-1> ma;
    cout<<gaussM(m, 3, 3, ma)<<endl;
    cout<<ma<<endl;
}
```

## 21 src/math/miller-rabin.cpp

```cpp
// Deterministic Miller-Rabin primality test
// Works for all 64 bit integers
// Support of 128 bit integers is required to test over 32 bit integers
// Source: http://qubit.pw/trophy.pdf
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef __int128 lll;

lll powmod(lll a, lll p, lll mod){
        if (p==0) return 1;
        if (p==2) return (a*a)%mod;
        if (p%2==1) return (a*powmod(a, p-1, mod))%mod;
        lll t=powmod(a, p/2, mod);
        return (t*t)%mod;
}

bool is_w(ll a, ll even, ll odd, ll p){
        lll u = powmod(a, odd, p);
        if (u==1) return 0;
        for (ll j=1;j<even;j*=2) {
                if (u==p-1) return 0;
                u*=u;
                u%=p;
        }
        return 1;
}

bool isPrime(ll p) {
        if (p==2) return 1;
        if (p<=1||p%2==0) return 0;
        ll odd=p-1;
        ll even=1;
        while (odd%2==0) {
                even*=2;
                odd/=2;
        }
        ll b[7]={2, 325, 9375, 28178, 450775, 9780504, 1795265022};
        for (ll i=0;i<7;i++) {
                ll a=b[i]%p;
                if (a==0) return 1;
                if (is_w(a, even, odd, p)) return 0;
        }
        return 1;
}
```

## 22 src/string/lcparray.cpp

```cpp
// Constructs LCP array from suffix array in O(n) time
// You can change vector<int> s to string s
#include <bits/stdc++.h>
using namespace std;

vector<int> lcpArray(vector<int> s, vector<int> sa) {
        int n=s.size();
        int k=0;
        vector<int> ra(n), lcp(n);
        for (int i=0;i<n;i++) ra[sa[i]]=i;
        for (int i=0;i<n;i++) {
                if (k) k--;
                if (ra[i]==n-1) {
                        k=0;
                        continue;
                }
                int j=sa[ra[i]+1];
                while (k<n&&s[(i+k)%n]==s[(j+k)%n]) k++;
                lcp[ra[i]]=k;
                if (ra[(sa[ra[i]]+1)%n]>ra[(sa[ra[j]]+1)%n]) k=0;
        }
        return lcp;
}
```

## 23 src/string/suffixarray.cpp

```cpp
// Suffix array in O(n log^2 n)
// ~300ms runtime for 10^5 character string, ~2000ms for 5*10^5
// You can change vector<int> s to string s
#include <bits/stdc++.h>
#define F first
#define S second
using namespace std;

vector<int> suffixArray(vector<int> s) {
	int n=s.size();
	vector<int> k(n);
	for (int i=0;i<n;i++) {
		k[i]=s[i];
	}
	vector<pair<pair<int, int>, int> > v(n);
	for (int t=1;t<=n;t*=2) {
		for (int i=0;i<n;i++) {
			int u=-1;
			if (i+t<n) u=k[i+t];
			v[i]={{k[i], u}, i};
		}
		sort(v.begin(), v.end());
		int c=0;
		for (int i=0;i<n;i++) {
			if (i>0&&v[i-1].F!=v[i].F) c++;
			k[v[i].S]=c;
		}
		if (c==n-1) break;
	}
	vector<int> sa(n);
	for (int i=0;i<n;i++) sa[k[i]]=i;
	return sa;
}
```

## 24 src/string/suffixautomaton.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

struct suffixAutomaton{
        vector<map<char, int> > g;
        vector<int> link;
        vector<int> len;
        int last;
        void addC(char c){
                int p=last;
                int t=link.size();
                link.push_back(0);
                len.push_back(len[last]+1);
                g.push_back(map<char, int>());
                while (p!=-1&&g[p].count(c)==0){
                        g[p][c]=t;
                        p=link[p];
                }
                if (p!=-1){
                        int q=g[p][c];
                        if (len[p]+1==len[q]){
                                link[t]=q;
                        }
                        else{
                                int qq=link.size();
                                link.push_back(link[q]);
                                len.push_back(len[p]+1);
                                g.push_back(g[q]);
                                while (p!=-1&&g[p][c]==q){
                                        g[p][c]=qq;
                                        p=link[p];
                                }
                                link[q]=qq;
                                link[t]=qq;
                        }
                }
                last=t;
        }
        suffixAutomaton() : suffixAutomaton(""){}
        suffixAutomaton(string s){
                last=0;
                g.push_back(map<char, int>());
                link.push_back(-1);
                len.push_back(0);
                for (int i=0;i<(int)s.size();i++){
                        addC(s[i]);
                }
        }
};
```

## 25 src/string/z.cpp

```cpp
// Computes the Z array in linear time
// z[i] is the length of the longest common prefix of substring
// starting at i and the string
// You can use string s instead of vector<int> s
// z[0]=0
#include <bits/stdc++.h>
using namespace std;

vector<int> zAlgo(vector<int> s) {
        int n=s.size();
        vector<int> z(n);
        int l=0;
        int r=0;
        for (int i=1;i<n;i++) {
                z[i]=max(0, min(z[i-l], r-i));

                while (i+z[i]<n&&s[z[i]]==s[i+z[i]]) z[i]++;

                if (i+z[i]>r) {
                        l=i;
                        r=i+z[i];
                }
        }
        return z;
}
```