Contents

1 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;
		}
	}
};
```

2 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());
			}
		}
	}
};
```

3 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;

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

	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]) {
			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;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];
    }
};
```

4 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);
        }
    }
};
```

5 src/graph/eulertour.cpp

```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;
                        }
                }
        }
};
```

6 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);
}
```

7 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;
}
```

8 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;
}
```

9 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;
}
```

10 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
#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;
```

```
}
```

11 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);
}
```

12 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()];
        }
};
```

13 src/datastructure/treap.cpp

```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);
}
```