Template

```
#include<bits/stdc++.h>
#define pb push_back
#define mp make_pair
#define fs first
#define sd second
using namespace std;

typedef long long ll;
typedef unsigned long long ull;
typedef pair<ll,ll> pl;
typedef vector<ll> vl;
typedef vector<pair<ll,ll>> vll;
typedef vector<pair<ll,pl>> vlll;
typedef priority_queue <ll, vector<ll>, greater<ll>> minh;

const int N = 1e6 + 3, Mod = 1e9 + 7;
const int maxN=1e3+3;

void solve(){

}

int main(){
	ios_base::sync_with_stdio(false);
	cin.tie(nullptr);
	cout.tie(nullptr);

	int t=1;
```

```
    cin>>t;
    for(int i=0;i<t;i++){
        //cout<<"Case #"<<i+1<<": ";
        solve();
    }
}
```

Data Structres

```
#include<bits/stdc++.h>

using namespace std;

typedef vector<int> vi;

const int N=1e6+3, Mod=1e9+7;

const int maxN=1e4+2;


//union find disjoint set

class dsu{

	private:

		vl p, rank, setSize;

		int numSets;

	public:

		dsu(int n){

			setSize.assign(n,1);

			numSets=n;

			rank.assign(n,0);

			p.assign(n,0);

			for(int i=0;i<n;i++)p[i]=i;

		}

		int findP(int i){

			return (p[i]==i)?i:findP(p[i]);

		}

		bool isSameP(int a, int b){

			return findP(a)==findP(b);

		}

		void unite(int a, int b){

			if(!isSameP(a,b)){

				numSets--;

				int x=findP(a);

				int y=findP(b);

				if(rank[x]>rank[y]){
```

```
                        p[y]=x;
                        setSize[x]+=setSize[y];
                    }else{
                        p[x]=y;
                        setSize[y]+=setSize[x];
                        if(rank[x]==rank[y])rank[y]++;
                    }
                }
        }
        int totSets(){
                return numSets;
        }
        int sizeOfSet(int i){
                return setSize[findP(i)];
        }
};

//segment tree (single update)
int a[maxN]; //tree source
struct info {
        int l, r, fl, fr, sz;
        long long tot;
} d[maxN * 4]; //tree storage

info operator + (info a, info b) {
        info c;
        c.l = a.l;
        c.r = b.r;
        c.sz = a.sz + b.sz;
        c.fl = (a.fl == a.sz && a.r <= b.l) ? (a.fl + b.fl) : a.fl;
        c.fr = (b.fr == b.sz && a.r <= b.l) ? (b.fr + a.fr) : b.fr;
```

```
        c.tot = a.tot + b.tot + ((a.r <= b.l) ? (1ll * a.fr * b.fl) : 0);
        return c;
} //merging two datas (ans)

void build(int k, int l, int r) { //(1,1,n)
        if (l == r) {
                d[k] = (info) {a[l], a[l], 1, 1, 1, 1ll};
        } else {
                int mid = (l + r) / 2;
                build(k * 2, l, mid);
                build(k * 2 + 1, mid + 1, r);
                d[k] = d[k * 2] + d[k * 2 + 1]; //operator +
        }
}

//rebuild, only with node x, change the array directly
void update(int k, int l, int r, int x) { //(1,1,n,node)
        if (l == r) {
                d[k] = (info) {a[l], a[l], 1, 1, 1, 1ll};
        } else {
                int mid = (l + r) / 2;
                if (x <= mid) update(k * 2, l, mid, x);
                else update(k * 2 + 1, mid + 1, r, x);
                d[k] = d[k * 2] + d[k * 2 + 1]; //operator +
        }
}

info query(int k, int l, int r, int x, int y) { //(1,1,n,leftq,rightq)
        if (l == x && r == y) {
                return d[k];
        } else {
```

```
            int mid = (l + r) / 2;
            if (y <= mid) return query(k * 2, l, mid, x, y);
            else if (x > mid) return query(k * 2 + 1, mid + 1, r, x, y);
            else return query(k * 2, l, mid, x, mid) + query(k * 2 + 1, mid + 1, r, 
mid + 1, y); //operator +
      }
}
//fenwick tree
class fenwick{
      private:
            vl ft;
      public:
            fenwick(int n){
                  ft.assign(n+1,0);
            }
            int rsq(int b){
                  int ret=0;
                  for(;b;b-=(b&(-b)))ret+=ft[b];
                  return ret;
            }
            int rsq(int a,int b){
                  return rsq(b)-(a==1?0:rsq(a-1));
            }
            void update(int i, int val){
                  for(;i<(int)ft.size();i+=(i&-i))ft[i]+=val;
            }
};
```

Algorithms

```
//Binary Search

ll binser(ll l,ll r,ll val){
	if(l>=r)return l;
	ll mid=(l+r)/2;
	//printf(".%d",mid);
	if(arr[mid]==val)return mid;
	if(arr[mid]>val&&arr[mid-1]<val)return mid;
	if(arr[mid]>val) return binser(l,mid-1,val);
	if(arr[mid]<val) return binser(mid+1,r,val);
}


//fast C(N,K)

int n, k;

long long fact[N], invf[N], inv[N];


long long modpow(long long x, long long y) {
    long long ret = 1;
    while (y > 0) {
        if (y & 1) ret = (ret * x) % Mod;
        y >>= 1;
        x = (x * x) % Mod;
    }
    return ret;
}


void preprocess() {
    fact[0] = invf[0] = 1;
    for (int i = 1; i < N; i++) {
        fact[i] = (fact[i - 1] * i) % Mod;
        invf[i] = modpow(fact[i], Mod - 2);
        inv[i]=modpow(i,Mod-2);
```

```
    }
}

long long C(int a, int b) {
    if (a < b) return 0;
    long long ret = (fact[a] * invf[a - b]) % Mod;
    ret = (ret * invf[b]) % Mod;
    return ret;
}
//Dijkstra
```

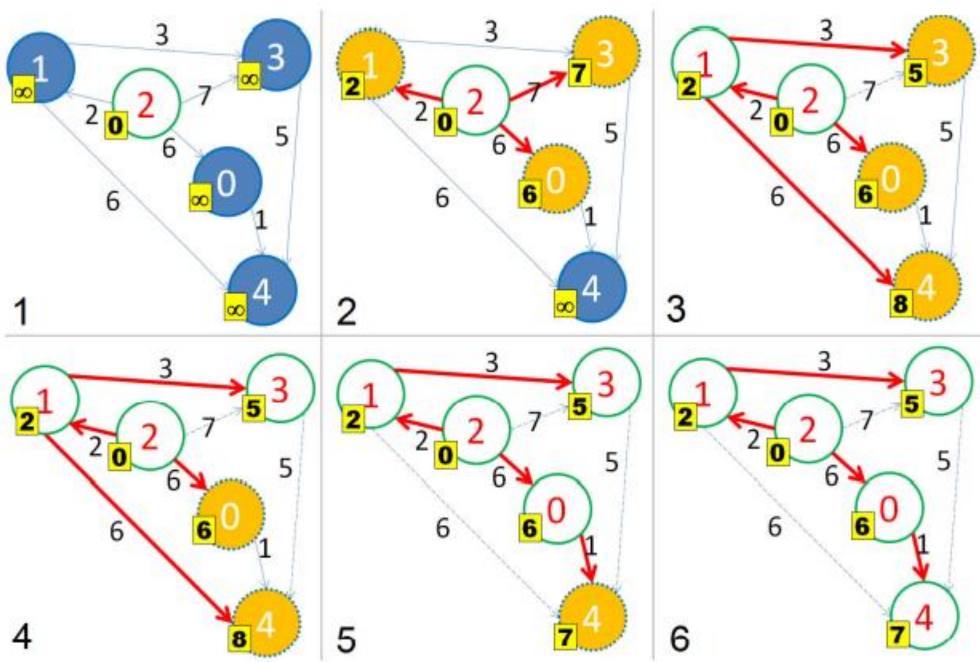

Figure 4.17: Dijkstra Animation on a Weighted Graph (from UVa 341 [47])

```
//Bellman Ford
```

```
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++)              // relax all E edges V-1 times
  for (int u = 0; u < V; u++)     // these two loops = O(E), overall O(VE)
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];           // record SP spanning here if needed
      dist[v.first] = min(dist[v.first], dist[u] + v.second);     // relax
    }
```

//Floyd Warshall

```
// inside int main()
  // precondition: AdjMat[i][j] contains the weight of edge (i, j)
  // or INF (1B) if there is no such edge
  // AdjMat is a 32-bit signed integer array
  for (int k = 0; k < V; k++)        // remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMat[i][j] = min(AdjMat[i][j], AdjMat[i][k] + AdjMat[k][j]);
```

//Articulation Point

```
void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {                      // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;      // special case if u is a root

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])          // for articulation point
        articulation_vertex[u] = true;      // store this information first
      if (dfs_low[v.first] > dfs_num[u])                      // for bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);  // update dfs_low[u]
    }
    else if (v.first != dfs_parent[u])  // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);  // update dfs_low[u]
} }
```

//Tarjan SCC

```
void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  S.push_back(u);       // stores u in a vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      tarjanSCC(v.first);
    if (visited[v.first])                             // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); }

  if (dfs_low[u] == dfs_num[u]) {    // if this is a root (start) of an SCC
    printf("SCC %d:", ++numSCC);        // this part is done after recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break; }
    printf("\n");
} }
```

```
//Toposort, lexicographically smallest (inside main)
	ll n,m; cin>>n>>m;
	for(int i=0;i<m;i++){
		ll x,y; cin>>x>>y;
		adj[x].pb(y);
		w[y]++;
	}
	vl ans;
	priority_queue <ll, vector<ll>, greater<ll>>pq;
	for(ll i=1;i<=n;i++){
		if(w[i])continue;
		pq.push(i);
	}
	while(!pq.empty()){
		ll idx=pq.top(); pq.pop();
		ans.pb(idx);
		for(auto v: adj[idx]){
			w[v]--;
			if(!w[v])pq.push(v);
		}
	}
// Cycle-finding DFS
ll adj[maxN];
ll w[maxN];
int visit[maxN];
vl path;
ll ans=0;

void dfs(ll i){
	path.pb(i);
	visit[i]=1;
```

```
        if(visit[adj[i]]==1){
                int n=path.size()-1;
                ll ret=w[path[n]];
                while(path[n]!=adj[i]){
                        ret=min(ret,w[path[n]]); //or pb to cycle list
                        n--;
                }
                ret=min(ret,w[path[n]]);
                ans+=ret;
        }
        if(visit[adj[i]]==0){
                dfs(adj[i]);
        }
        visit[i]=2;
}

//Bipartite (Color 1,2)
vl adj[maxN];
int col[maxN];
int cnt[3];
bool f=1;

int xxor(int a){
        if(a==1)return 2;
        else return 1;
}

bool dfs(int u){
        for(auto v:adj[u]){
                if(col[v]==0){
                        col[v]=xxor(col[u]);
```

```
                cnt[col[v]]++;
                if(!dfs(v))return 0;
            }
            else if(col[v]!=xxor(col[u])){
                f=0;
                return 0;
            }
        }
        return 1;
}
//KMP String
```

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N];                              // T = text, P = pattern
int b[MAX_N], n, m;     // b = back table, n = length of T, m = length of P

void kmpPreprocess() {                   // call this before calling kmpSearch()
  int i = 0, j = -1; b[0] = -1;                               // starting values
  while (i < m) {                              // pre-process the pattern string P
    while (j >= 0 && P[i] != P[j]) j = b[j];  // different, reset j using b
    i++; j++;                                   // if same, advance both pointers
    b[i] = j; // observe i = 8, 9, 10, 11, 12, 13 with j = 0, 1, 2, 3, 4, 5
} }                          // in the example of P = "SEVENTY SEVEN" above

void kmpSearch() {   // this is similar as kmpPreprocess(), but on string T
  int i = 0, j = 0;                                           // starting values
  while (i < n) {                                     // search through string T
    while (j >= 0 && T[i] != P[j]) j = b[j];  // different, reset j using b
    i++; j++;                                   // if same, advance both pointers
    if (j == m) {                                  // a match found when j == m
      printf("P is found at index %d in T\n", i - j);
      j = b[j];                          // prepare j for the next possible match
} } }
```

Math

//Floyd's Hare and Turtle

```
ii floydCycleFinding(int x0) {  // function int f(int x) is defined earlier
  // 1st part: finding k*mu, hare's speed is 2x tortoise's
  int tortoise = f(x0), hare = f(f(x0));    // f(x0) is the node next to x0
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
  // 2nd part: finding mu, hare and tortoise move at the same speed
  int mu = 0; hare = x0;
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu++;}
  // 3rd part: finding lambda, hare moves, tortoise stays
  int lambda = 1; hare = f(tortoise);
  while (tortoise != hare) { hare = f(hare); lambda++; }
  return ii(mu, lambda);
}
```

//Grundy Nim

Let $G(u)$ be the Grundy number of a pile with u stones. The Grundy number of a terminal state is 0; otherwise, $G(u)$ is recursively defined as the **minimum excludant** of the Grundy numbers of all the states it points to. We usually write minimum excludant as **mex**, and it basically means "return the smallest nonnegative integer that is **not** in this set." Let's consider a basic example, such as the Nim game where you can only take a square number of stones from a pile. If there is a pile with 5 stones, then we can take $1^2 = 1$ stone and transition to a pile with 4 stones, or take $2^2 = 4$ stones and transition to a pile with 1 stone. So, $G(5) = mex(G(5-1), G(5-4)) = mex(G(4), G(1)) = mex(2, 1) = 0$. So, it turns out that a pile with 5 stones in it is a loss state (you can verify the values of $G(4)$ and $G(1)$ for yourself, but even without Grundy numbers, you should be able to see why 5 stones is a losing state).

How about when there are multiple piles though? Amazingly, we can apply the same strategy we did earlier for Nim, except on the Grundy numbers. The important Sprague-Grundy theorem states that **these games are equivalent to playing Nim, but instead of getting the Nim-sum by taking the XOR of the piles, we take the XOR of their Grundy numbers.**

Let's consider that square number Nim again. We see that the Grundy numbers for $0, 1, 2, 3, 4, 5, 6$ are $0, 1, 0, 1, 2, 0, 1$, respectively. You can verify this. Therefore, if we have a game state with piles of $1, 2, 3, 4, 6$ stones, the Nim-sum of this game is $G(1) \oplus G(2) \oplus G(3) \oplus G(4) \oplus G(6) = 1 \oplus 0 \oplus 1 \oplus 2 \oplus 1 = 3$, which is non-zero, therefore the first player has a winning strategy.

//Catalan Numbers

5.4.3 Catalan Numbers

First, let's define the n-th Catalan number—written using binomial coefficients notation ${}^nC_k$ above—as: $Cat(n) = ({}^{(2\times n)}C_n)/(n+1)$; $Cat(0) = 1$. We will see its purpose below.

If we are asked to compute the values of $Cat(n)$ for *several* values of n, it may be better to compute the values using bottom-up Dynamic Programming. If we know $Cat(n)$, we can compute $Cat(n+1)$ by manipulating the formula like shown below.

$Cat(n) = \frac{2n!}{n!\times n!\times(n+1)}$.

$Cat(n+1) = \frac{(2\times(n+1))!}{(n+1)!\times(n+1)!\times((n+1)+1)} = \frac{(2n+2)\times(2n+1)\times 2n!}{(n+1)\times n!\times(n+1)\times n!\times(n+2)} = \frac{(2n+2)\times(2n+1)\times\ldots[2n!]}{(n+2)\times(n+1)\times\ldots[n!\times n!\times(n+1)]}$.

Therefore, $Cat(n+1) = \frac{(2n+2)\times(2n+1)}{(n+2)\times(n+1)} \times Cat(n)$.

Alternatively, we can set $m = n+1$ so that we have: $Cat(m) = \frac{2m\times(2m-1)}{(m+1)\times m} \times Cat(m-1)$.

[9] Binomial is a special case of polynomial that only has two terms.

Catalan numbers are found in various combinatorial problems. Here, we list down some of the more interesting ones (there are several others, see **Exercise 5.4.4.8***). All examples below use $n = 3$ and $Cat(3) = ({}^{(2\times3)}C_3)/(3+1) = ({}^6C_3)/4 = 20/4 = 5$.

1. $Cat(n)$ counts the number of distinct binary trees with n vertices, e.g. for $n = 3$:

```
    *    *    *    *    *
   /    /    / \    \    \
  *    *    *   *    *    *
 /      \           /      \
*        *         *        *
```

2. $Cat(n)$ counts the number of expressions containing n pairs of parentheses which are correctly matched, e.g. for $n = 3$, we have: ()()(), ()(()), (())(), ((())), and (()()).

3. $Cat(n)$ counts the number of different ways $n + 1$ factors can be completely parenthesized, e.g. for $n = 3$ and $3 + 1 = 4$ factors: {a, b, c, d}, we have: (ab)(cd), a(b(cd)), ((ab)c)d, (a(bc))(d), and a((bc)d).

4. $Cat(n)$ counts the number of ways a convex polygon (see Section 7.3) of $n + 2$ sides can be triangulated. See Figure 5.1, left.

5. $Cat(n)$ counts the number of monotonic paths along the edges of an $n \times n$ grid, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. See Figure 5.1, right and also see Section 4.7.1.

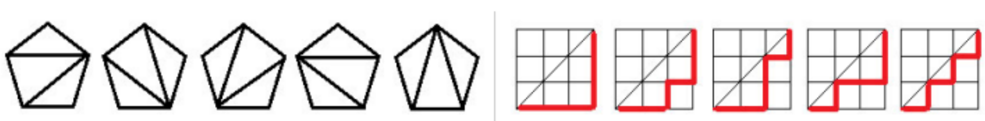

Figure 5.1: Left: Triangulation of a Convex Polygon, Right: Monotonic Paths