— MACROS —

```
// Macros
#define forn(i,e) for(ll i = 0; i < e; i++)
#define forsn(i,s,e) for(ll i = s; i < e; i++)
#define rforn(i,s) for(ll i = s; i >= 0; i--)
#define ln "\\n"
#define mp make_pair
#define pb push_back
#define fi first
#define se second
#define all(x) (x).begin(), (x).end()
#define sz(x) ((ll)(x).size())
#define INF 2e9

// Typedefs
typedef long long ll;
typedef long double ld;
typedef pair<int,int> pii;
typedef pair<ll,ll> pll;
typedef vector<ll> vll;
typedef vector<int> vi;
typedef vector<bool> vb;
typedef vector<vector<int>> vv;
typedef vector<pll> vpll;
```

— MAIN FUNCTION —

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

    ll t;
    cin >> t;
    for (ll i = 0; i < t; i++) {
        solve();
    }
    return 0;
}
```

— DYNAMIC PROGRAMMING —

Coin Change Min Coins

```
//Retorna INF caso seja impossivel
ll min_coins(ll change, vll &coins){
	vll dp(change + 1, INF);
	dp[0] = 0;

	forsn(sub, 1, change + 1){
		for(auto coin: coins){
			if(coin <= sub){
				dp[sub] = min(dp[sub], dp[sub-coin] + 1);
			}
		}
	}
	return dp[change];
}

void solve(){
	vll coins = {5, 4, 1};
	ll change = 13;

	cout << min_coins(change, coins) << endl;
}
```

Coin Change Number of Combinations

```
void solve(){
	int n, change;
	cin >> n >> change;
	vi coins(n);
	for(int &i: coins){
		cin >> i;
	}

	vi dp(change+1, 0);
	dp[0] = 1;
	forsn(sub, 1, change+1){
		for(int coin = 0; coin < n; coin++){
			if(coins[coin] <= sub){
				(dp[sub] += dp[sub-coins[coin]]) %= 1000000007;
			}
		}
	}

	cout << dp[change] << endl;
}
```

Coin Change Number of Combinations (Ordered)

```
ll combs (ll change, vll coins){
```

```
	vll dp(change+1, 0);
	dp[0] = 1;

	forn(coin, sz(coins)){
		forn(sub, change+1){
			if(coins[coin] <= sub){
				dp[sub] += dp[sub - coins[coin]] % 1000000007;
			}
		}
	}
	return dp[change];
}

void solve(){
	ll n, change;
	cin >> n >> change;
	vll coins(n);
	for(ll &i: coins){
		cin >> i;
	}

	cout << combs(change, coins) % 1000000007 << endl;
}
```

Box Stacking

```
// FIND TALLEST POSSIBLE STACK (LENGTH, WIDTH, HEIGHT)
bool compareLength(vll Box1, vll Box2){
	return Box1[0] < Box2[0];
}

bool canBeStacked(ll wTop, ll lTop, ll wBottom, ll lBottom){
	return wTop < wBottom && lTop < lBottom;
}

ll tallestStack (vvll boxes, ll n){
	sort(all(boxes), compareLength); // sort all boxes by length
	map<vll, ll> heights; // memoize the tallest stack with box n at the base
	for(auto box: boxes){
		heights[box] = box[2];
	}

	for(auto box_i: boxes){
		vll S; // vector of heights of stacks starting at boxes that can be stacked on top of box_i
		for(auto j: boxes){
			if(canBeStacked(j[1], j[0], box_i[1], box_i[0]))
				S.pub(heights[j]);
		}
		if(!S.empty())
			heights[box_i] = heights[box_i] + (*max_element(all(S)));
	}

	ll maxHeight = 0;
	for(auto i: heights){
		if(i.second > maxHeight)
			maxHeight = i.second;
	}
	return maxHeight;
}

void solve(){
	ll n = 6;
	vvll boxes = {{1, 2, 2}, {1, 5, 4}, {2, 3, 2}, {2, 4, 1}, {3, 6, 2}, {4, 5, 3}};
	cout << tallestStack(boxes, n) << endl;
}
```

Knapsack (Top-Down)

```
// Returns the value of maximum profit
int knapSackRec(int W, int wt[], int val[], int index, int** dp)
{
	// base condition
	if (index < 0)
		return 0;
	if (dp[index][W] != -1)
		return dp[index][W];

	if (wt[index] > W) {

		// Store the value of function call
		// stack in table before return
		dp[index][W] = knapSackRec(W, wt, val, index - 1, dp);
		return dp[index][W];
	}
	else {
		// Store value in a table before return
		dp[index][W] = max(val[index]
						+ knapSackRec(W - wt[index], wt, val,
								index - 1, dp),
					knapSackRec(W, wt, val, index - 1, dp));

		// Return value of table after storing
		return dp[index][W];
	}
}

int knapSack(int W, int wt[], int val[], int n)
{
	// double pointer to declare the
	// table dynamically
	int** dp;
	dp = new int*[n];

	// loop to create the table dynamically
	for (int i = 0; i < n; i++)
		dp[i] = new int[W + 1];

	// loop to initially filled the
	// table with -1
```

```
	for (int i = 0; i < n; i++)
		for (int j = 0; j < W + 1; j++)
			dp[i][j] = -1;
	return knapSackRec(W, wt, val, n - 1, dp);
}

// Driver Code
int main()
{
	int profit[] = { 60, 100, 120 };
	int weight[] = { 10, 20, 30 };
	int W = 50;
	int n = sizeof(profit) / sizeof(profit[0]);
	cout << knapSack(W, weight, profit, n);
	return 0;
}
```

Knapsack (Bottom-Up)

```
int knapsack(int W, int n, vi &price, vi &weight){ //linear memory
vi aux(W+1, 0);
for (int i = 0; i < n; i++){
	for (int j = W; j > 0; j--){
		if (weight[i] <= j){
			aux[j] = max(aux[j - weight[i]] + price[i], aux[j]);
		}
	}
}
return aux[W];
}
```

Longest Increasing Subsequence

```
void printLIS(int i, vi &p, vi &arr){ //imprime LIS, sabendo o ultimo indice
	if (p[i] == -1){
		cout<<arr[i];
		return;
	}
	printLIS(p[i], p, arr);
	cout<<' '<<arr[i];
}

pii LIS(int n, vi &p, vi &arr){ //retorna maior LIS e o ultimo indice do maior LIS
	int k= 0, lis_end = 0;
	vi L(n, 0), L_id(n, 0);
	p.assign(n, -1);

	for (int i = 0; i < n; i++){
		int pos = lower_bound(L.begin(), L.begin() + k, arr[i]) - L.begin();
		L[pos] = arr[i];
		L_id[pos] = i;
		p[i] = pos ? L_id[pos-1]:-1;
		if (pos == k){
			k = pos + 1;
			lis_end = i;
		}
	}
	return mp(k, lis_end);
}
```

Monotonic Paths

```
//n e m arestas, NAO vertices
ll Monotonic(int n, int m, ll p){    //Se for n*n, usar mod(Catalan(n)*(n+1), m)
	n++;
	m++;
	vvll T(n, vll(m));
	for (int i = 0; i < n; i++){
	T[i][0] = 1;
	}
	for (int i = 0; i < m; i++){
	T[0][i] = 1;
	}
	for (int i = 1; i < n; i++){
		for (int j = 1; j < n; j++){
			T[i][j] = mod(mod(T[i-1][j], p) + mod(T[i][j-1], p), p);
		}
	}
	return mod(T[n-1][m-1], p);
}
```

— DATA STRUCTURES —

Order Statistic Tree (Map and Set)

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

using oset = tree<int, // key type
              null_type, // value type
              less<int>, // compare function
              rb_tree_tag,
              tree_order_statistics_node_update>;
auto s = oset();

int main() {
  auto s = oset();

  s.insert(10);
  s.insert(50);
  s.insert(42);

  cout << *s.find_by_order(0) << '\n';
  cout << *s.find_by_order(1) << '\n';
  cout << *s.find_by_order(2) << '\n';
```

```
  cout << s.order_of_key(10) << '\n';
  cout << s.order_of_key(42) << '\n';
  cout << s.order_of_key(50) << '\n';
  cout << s.order_of_key(-2) << '\n';

  return 0;
}

// -------------------------------------

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

using omap = tree<int, // key type
                int, // value type
                less<int>, // compare function
                rb_tree_tag,
                tree_order_statistics_node_update>;
auto m = omap();

int main() {
  auto m = omap();

  m.insert(make_pair(10, 1));
  m.insert(make_pair(50, 5));
  m.insert(make_pair(42, 4));

  auto it0 = m.find_by_order(0);
  cout << it0->first << " " << it0->second << '\n';

  auto it1 = m.find_by_order(1);
  cout << it1->first << " " << it1->second << '\n';

  auto it2 = m.find_by_order(2);
  cout << it2->first << " " << it2->second << '\n';

  cout << m.order_of_key(10) << '\n';
  cout << m.order_of_key(42) << '\n';
  cout << m.order_of_key(50) << '\n';
  cout << m.order_of_key(-2) << '\n';

  return 0;
}
```

PATRICIA Trie Set

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/trie_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

using ptset = trie<string,
                                    null_type,
                                    trie_string_access_traits<>,
                                    pat_trie_tag,
                                    trie_prefix_search_node_update>;

int main() {
        auto s = ptset();

        s.insert("to");
        s.insert("tea");
        s.insert("ted");
        s.insert("ten");
        s.insert("a");
        s.insert("in");
        s.insert("inn");

        // AMMOUNT OF ELEMENTS
        cout << s.size() << "\n";

        // CHECKS IF A GIVEN STRING EXISTS
        cout << boolalpha << (s.find("tea") != s.end()) << "\n";
        cout << boolalpha << (s.find("te") != s.end()) << "\n";

        // FINDS AND COUNTS THE NUMBER OF STRINGS WITH A GIVEN PREFIX
        auto prefix_range = s.prefix_range("te");
        cout << distance(prefix_range.first, prefix_range.second) << "\n";

        return 0;
}
```

Segment Tree

```
#define op(l, r) (l + r);
#define DEFAULTVALUE 0
const ll inf = 1e9;

struct Node {
        Node *l = 0, *r = 0;
        ll lo, hi, mset = inf, madd = 0;
        ll val = DEFAULTVALUE;
        Node(ll lo,ll hi):lo(lo),hi(hi){} // Large interval of -inf
        Node(vector<int>& v, ll lo, ll hi) : lo(lo), hi(hi) {
                if (lo + 1 < hi) {
                        ll mid = lo + (hi - lo)/2;
                        l = new Node(v, lo, mid);
                        r = new Node(v, mid, hi);
                        val = op(l->val, r->val);
                }
                else val = v[lo];
        }
        ll query(ll L, ll R) {
                if (R <= lo || hi <= L) return 0;
                if (L <= lo && hi <= R) return val;
                push();
                return op(l->query(L, R), r->query(L, R));
        }
```

```
	void set(ll L, ll R, ll x) {
		if (R <= lo || hi <= L) return;
		if (L <= lo && hi <= R) mset = val = x, madd = 0;
		else {
			push(), l->set(L, R, x), r->set(L, R, x);
			val = op(l->val, r->val);
		}
	}
	void add(ll L, ll R, ll x) {
		if (R <= lo || hi <= L) return;
		if (L <= lo && hi <= R) {
			if (mset != inf) mset += x;
			else madd += x;
			val += x;
		}
		else {
			push(), l->add(L, R, x), r->add(L, R, x);
			val = op(l->val, r->val);
		}
	}
	void push() {
		if (!l) {
			ll mid = lo + (hi - lo)/2;
			l = new Node(lo, mid); r = new Node(mid, hi);
		}
		if (mset != inf)
			l->set(lo,hi,mset), r->set(lo,hi,mset), mset = inf;
		else if (madd)
			l->add(lo,hi,madd), r->add(lo,hi,madd), madd = 0;
	}
};

void solve(){
	int type, k, u, n, q;
	cin >> n >> q;

	vector<int> v(n);
	for(auto &i: v) cin >> i;

	Node *root = new Node(v, 0, n);

	while(q--){
		cin >> type >> k >> u;
		if(type == 1){ // Point update index k-1 to u
			root->set(k-1, k, u);
			continue;
		}
		cout << root->query(k-1, u) << endl; // Get the sum from index k-1 to u (
		    ↪ exclusive end)
	}
}
```

Segment Tree (Max Prefix Sum)

```
const ll inf = 1e9;
#define DEFAULTVALUE -inf

//#define op(l, r) max(l, r+l)
pair<ll, ll> op(pair<ll, ll> l, pair<ll, ll> r){
	return make_pair((max(l.first, r.first + l.second)), (l.second + r.second));
}

struct Node {
	Node *l = 0, *r = 0;
	ll lo, hi, mset = inf, madd = 0;
	pair<ll, ll> val = {DEFAULTVALUE, 0};

	Node(ll lo,ll hi):lo(lo),hi(hi){} // Large interval of -inf
	Node(vector<int>& v, ll lo, ll hi) : lo(lo), hi(hi) {
		if (lo + 1 < hi) {
			ll mid = lo + (hi - lo)/2;
			l = new Node(v, lo, mid);
			r = new Node(v, mid, hi);
			val = op(l->val, r->val);
			//cout << lo << " " << hi << " " << val << endl;
		}
		else {
			val = {v[lo], v[lo]};
			// cout << val << endl;
		}
	}
	pair<ll, ll> query(ll L, ll R) {
		if (R <= lo){
			return make_pair(DEFAULTVALUE, 0);
		}else if (hi <= L){
			return make_pair(DEFAULTVALUE, 0);
		}
		if (L <= lo && hi <= R) return val;
		// cout << "prepush: " << lo << " " << hi << " " << val << endl;
		push();
		// cout << "pospush: " << lo << " " << hi << " " << val << endl;
		// cout << "\n\n\n";

		return op(l->query(L, R), r->query(L, R));
	}
	void set(ll L, ll R, ll x) {
		if (R <= lo || hi <= L) return;
		if (L <= lo && hi <= R){
			mset = val.first = x, madd = 0;
			val.second -= val.second;
			val.second += x;
		}
		else {
			push(), l->set(L, R, x), r->set(L, R, x);
			val = op(l->val, r->val);
		}
	}
	void add(ll L, ll R, ll x) {
		if (R <= lo || hi <= L) return;
		if (L <= lo && hi <= R) {
			if (mset != inf) mset += x;
			else madd += x;
```

```
			val.first += x;
		}
		else {
			push(), l->add(L, R, x), r->add(L, R, x);
			val = op(l->val, r->val);
		}
	}
	void push() {
		if (!l) {
			ll mid = lo + (hi - lo)/2;
			l = new Node(lo, mid); r = new Node(mid, hi);
		}
		if (mset != inf)
			l->set(lo,hi,mset), r->set(lo,hi,mset), mset = inf;
		else if (madd)
			l->add(lo,hi,madd), r->add(lo,hi,madd), madd = 0;
	}
};
```

Persistent Segtree

```
struct Node{ int mn, l, r; };

int init(int l, int r, Node st[], int* curr){
	if (l == r){ st[++(*curr)].mn = INF; return (*curr); }
	int m = l+(r-l)/2;
	int p = ++(*curr);
	st[p] = {0, init(l, m, st, curr), init(m+1, r, st, curr)};
	st[p].mn = min(st[st[p].l].mn, st[st[p].r].mn);
	return p;
}

int update(int i, int l, int r, int k, int x, Node st[], int* curr){
	if (l == r){ st[++(*curr)].mn = x; return *curr; }
	int m = l+(r-l)/2;
	int p = ++(*curr);
	if (k <= m){
		st[p] = {0, update(st[i].l, l, m, k, x, st, curr), st[i].r};
	} else {
		st[p] = {0, st[i].l, update(st[i].r, m+1, r, k, x, st, curr)};
	}
	st[p].mn = min(st[st[p].l].mn, st[st[p].r].mn);
	return p;
}

int query(int i, int l, int r, int tl, int tr, Node st[]){
	if (l > tr || r < tl) return INF;
	if (tl <= l && r <= tr) return st[i].mn;
	int m = l+(r-l)/2;
	return min(query(st[i].l, l, m, tl, tr, st), query(st[i].r, m+1, r, tl, tr, st));
}

int arr[n+1], root[n+2], curr = 1;    //Tres linhas seguintes por no solve
map<int, int> pos;
Node st[22*n];
```

Trie

```
template<char MIN_CHAR = 'a', int ALPHABET = 26>
struct array_trie {
	struct trie_node {
		array<int, ALPHABET> child;
		int words_here = 0, starting_with = 0;

		trie_node() {
			memset(&child[0], -1, ALPHABET * sizeof(int));
		}
	};

	static const int ROOT = 0;

	vector<trie_node> nodes = {trie_node()};

	array_trie(int total_length = -1) {
		if (total_length >= 0)
			nodes.reserve(total_length + 1);
	}

	int get_or_create_child(int node, int c) {
		if (nodes[node].child[c] < 0) {
			nodes[node].child[c] = int(nodes.size());
			nodes.emplace_back();
		}
		return nodes[node].child[c];
	}

	int build(const string &word, int delta) {
		int node = ROOT;
		for (char ch : word) {
			nodes[node].starting_with += delta;
			node = get_or_create_child(node, ch - MIN_CHAR);
		}
		nodes[node].starting_with += delta;
		return node;
	}

	int add(const string &word) {
		int node = build(word, +1);
		nodes[node].words_here++;
		return node;
	}

	int erase(const string &word) {
		int node = build(word, -1);
		nodes[node].words_here--;
		return node;
	}

	int find(const string &str) const {
		int node = ROOT;
		for (char ch : str) {
			node = nodes[node].child[ch - MIN_CHAR];
```

```
                if (node < 0)
                    break;
            }
            return node;
        }

        int count_prefixes(const string &str, bool include_full) const {
            int node = ROOT, count = 0;
            for (char ch : str) {
                count += nodes[node].words_here;
                node = nodes[node].child[ch - MIN_CHAR];
                if (node < 0)
                    break;
            }
            if (include_full && node >= 0)
                count += nodes[node].words_here;
            return count;
        }

        int count_starting_with(const string &str, bool include_full) const {
            int node = find(str);
            if (node < 0)
                return 0;
            return nodes[node].starting_with - (include_full ? 0 : nodes[node].
                ↪ words_here);
        }
};
```

Persistent Trie

```
// Node for lowercase strings
struct Node {
        array<shared_ptr<Node>, 26> children;
        bool end; // whether this node represents the end of a key
        size_t count; // optional (depending on queries)

        Node() : children{}, end{false}, count{0} {}
};

class Trie {
private:
        shared_ptr<Node> root;
        explicit Trie(shared_ptr<Node> root) : root(root) {}

public:
        Trie() : root(new Node()) {}
        size_t size() const {
        return root->count;
        }

        bool exists(string_view s) const {
                auto node = root;
                for (auto c : s) {
                        auto idx = c - 'a';
                        if (node->children[idx]) {
                                node = node->children[idx];
                        } else {
                                return false;
                        }
                }
                return node->end;
        }

        optional<Trie> insert(string_view s) {
                if (exists(s)) {
                        return {};
                }

                auto nroot = make_shared<Node>(*root);
                auto node = nroot;
                node->count += 1;
                for (auto c : s) {
                        auto idx = c - 'a';
                        if (node->children[idx]) {
                                node = node->children[idx] = make_shared<Node>(*(node->
                                    ↪ children[idx]));
                        } else {
                                node = node->children[idx] = make_shared<Node>();
                        }
                        node->count += 1;
                }
                node->end = true;
                return Trie(nroot);
        }

        size_t count(string_view prefix) const {
                auto node = root.get();
                for (auto c : prefix) {
                        auto idx = c - 'a';
                        if (node->children[idx]) {
                                node = node->children[idx].get();
                        } else {
                                return 0;
                        }
                }
                return node->count;
        }
};
```

Sparse Table

```
class SparseTable{
        private:
                vi A, P2, L2; //A -> o array, P2 -> P2[x] = 2^x, L2 -> L2[x] = floor(log2(x
                    ↪ ))
                vv SpT;
                public:
                SparseTable(){}

                SparseTable(vi &initialA){
                A = initialA;
                int n = (int) A.size();
```

```
		int L2_n = (int) log2(n)+1;
		P2.assign(L2_n+1, 0);
		L2.assign((1<<L2_n)+1, 0);
		for (int i = 0; i <= L2_n; i++){
				P2[i] = (1<<i);
					L2[(1<<i)] = i;
				}
		for (int i = 2; i < P2[L2_n]; i++){
				if (L2[i] == 0) L2[i] = L2[i-1];
		}

		// the initialization phase
		SpT = vv (L2[n]+1, vi(n));
		for (int j = 0; j < n; j++){
				SpT[0][j] = j;
		}

		//the two nested loops below have overall time complexity = O(n log(n))
		for (int i = 1; P2[i] <= n; i++){
				for (int j = 0; j+P2[i]-1 < n; j++){
			int x = SpT[i-1][j];
						int y = SpT[i-1][j+P2[i-1]];
						SpT[i][j] = A[x] <= A[y] ? x : y;
					}
		}
		}

		int RMQ(int i, int j){
		int k = L2[j-i+1];
		int x = SpT[k][i];
		int y = SpT[k][j-P2[k]+1];
		return A[x] <= A[y] ? x : y;
		}
	};

	//Dentro de solve ou main
	SparseTable Spt = SparseTable(L);
```

— GRAPHS —

DFS

```
void dfs (int v, vector<bool> &visited, vv &graph){
	visited[v] = true;
	for(int no: graph[v]){
		if (!visited[no]){
			dfs(no, visited, graph);
		}
	}
	return;
}
```

BFS

```
vector<bool> visited(1001, false);
vv graph(1001);

void BFS (int root, int goal){
	int cur;
	queue<int> Q;
	visited[root] = true;
	cout << "visiting root\n";
	Q.push(root);

	while(!Q.empty()){
		cur = Q.front(); Q.pop();
		cout << "visiting node " << cur << endl;

		if (cur == goal)
			return;

		for(auto w: graph[cur]){
			if(visited[w] == false){
				visited[w] = true;
				Q.push(w);
			}
		}
	}
}
```

Flood Fill

```
bool valid(int i, int j, int n, int m, vector<vector<char>> &grid){
	return i>=0 && j>= 0 && i < n && j < m && grid[i][j] == '.';
}

void dfs(int i, int j, int n, int m, vv& visited,vector<vector<char>> &grid){
	visited[i][j] = 1;
	for (int k = 0; k < 4; k++){
		int ni = i + di[k], nj = j + dj[k];
		if (valid(ni, nj, n, m, grid) && !visited[ni][nj]){
			dfs(ni, nj, n, m, visited, grid);
		}
	}
}

void solve(){
	int n, m;
	cin>>n>>m;
	vector<vector<char>> grid(n, vector<char> (m));
	vv visited(n, vi (m));
	for (int i = 0; i < n; i++){
		for (int j = 0; j < m; j++){
			cin>>grid[i][j];
		}
	}
	int ans = 0;
	for (int i = 0; i < n; i++){
		for (int j = 0; j < m; j++){
```

```
                if (valid(i, j, n, m, grid) && !visited[i][j]){
                    dfs(i, j, n, m, visited, grid);
                    ans++;
                }
            }
        }
        cout<<ans<<endl;
}
```

Monsters/Avalanche Flood Fill

```
vector<vector<ll>> dist;
vector<vector<char>> grid;
vector<vector<pair<ll, ll>>> parents;
queue<pair<ll, ll>> q;
ll n, m;
string out;

bool possible = false;
bool advPath = false;

const int di[] = {1, 0, -1, 0};
const int dj[] = {0, -1, 0, 1};

bool edge(pair<ll, ll> coords){
    return (coords.first == 0 || coords.second == 0 || coords.first == n-1 || coords.second == m-1);
}

bool valid(ll i, ll j){
    return i >= 0 && j >= 0 && i < n && j < m && grid[i][j] == '.';
}

void getPath(pair<ll, ll> node){
    pair<ll, ll> parent = parents[node.first][node.second];
    if(parent.first == -1)
        return;
    if(parent.first == node.first + 1)
        out.push_back('U');
    if(parent.first == node.first - 1)
        out.push_back('D');
    if(parent.second == node.second + 1)
        out.push_back('L');
    if(parent.second == node.second - 1)
        out.push_back('R');
    getPath(parent);
}

void bfs(){
    ll curDist = 0;
    while(!q.empty()){
        pair<ll, ll> cur = q.front(); q.pop();
        curDist = dist[cur.first][cur.second];

        for(int k = 0; k < 4; k++){
            pair<ll, ll> next = {cur.first + di[k], cur.second + dj[k]};
            // curDist + 1 < dist[next.first][next.second] ensures that it is worth it to visit de adjacent node
            if((valid(next.first, next.second) && (curDist + 1 < dist[next.first][next.second]))){
                dist[next.first][next.second] = curDist + 1; // The distance from the origin to the next node is always the current distance + 1
                q.push(next);
                parents[next.first][next.second] = cur; // The next node's parent is the current one
            }
        }

        if(advPath && (edge(cur))){
            cout << "YES" << endl << dist[cur.first][cur.second] << endl;
            getPath(cur);
            reverse(out.begin(), out.end());
            cout << out << endl;
            possible = true;
        }
    }
}

void solve(){
    cin >> n >> m;
    char add;
    grid.resize(n, vector<char>(m));
    dist.resize(n, vector<ll>(m, INT_MAX));

    pair<ll, ll> start;
    parents.resize(n, vector<pair<ll, ll>>(m));

    for(int i = 0; i < n; ++i){
        for(int j = 0; j < m; ++j){
            cin >> add;
            if(add == 'A'){
                start = {i, j};
            }
            if(add == 'M'){
                q.push({i, j});
                dist[i][j] = 0;
            }
            grid[i][j] = add;
        }
    }

    // BFS for each one of the monsters
    bfs();

    advPath = true; // Flag to indicate that the next BFS will define the adventurer's path
    q.push(start);
    parents[start.first][start.second] = {-1, -1}; // This is set to {-1, -1} in order for the getPath function to know when it has reached the origin
    dist[start.first][start.second] = 0;
```

```
    bfs();

    if(!possible)
        cout << "NO" << endl;
}
```

Disjoint Set Union (Union Find)

```
//Cada valor comeca por ser o seu proprio set
void makeSet(int v, vi &parent) {
        parent[v] = v;
}
int findSet(int v, vi &parent) {
        if (v != parent[v])
        parent[v] = findSet(parent[v], parent);
        return parent[v];
}
void unionSets(int u, int v, vi &parent) {
        int root1 = findSet(u, parent);
        int root2 = findSet(v, parent);
        parent[root2] = root1;
}
bool check(int u, int v, vi &parent) {
        return findSet(u, parent) == findSet(v, parent);
}
```

Dijkstra

```
vector<int> dijkstra(vector<vector<pii>>& adjMatrix, int source, int target) {
        int n = adjMatrix.size();
        vector<int> dist(n, INF);
        vector<bool> visited(n, false);
        dist[source] = 0;
        priority_queue<pii, vector<pii>, greater<pii>> pq;
        pq.push(make_pair(0, source));
        while (!pq.empty()) {
                int u = pq.top().second;
                pq.pop();
                if (visited[u]) {
                        continue;
                }
                visited[u] = true;
                if (u == target) {
                        break;
                }
                for (auto& neighbor : adjMatrix[u]) {
                        int v = neighbor.first;
                        int weight = neighbor.second;
                        if (dist[v] > dist[u] + weight) {
                                dist[v] = dist[u] + weight;
                                pq.push(make_pair(dist[v], v));
                        }
                }
        }
        return dist;
```

```
}
```

Bellman-Ford

```
vi BF(vvpii &adjList, int source){
        int n = adjList.size();
        vi dist(n+1, INF);
        dist[source] = 0;
        for (int i = 1; i < n; i++){
                bool modified = false;
                for (int j = 1; j <= n; j++){
                        if (dist[j] != INF){
                                for (auto nbr: adjList[j]){
                                        int v = nbr.fi;
                                        int weight = nbr.se;
                                        if (dist[v] > dist[j] + weight){
                                                dist[v] = dist[j] +weight;
                                                modified = true;
                                        }
                                }
                        }
                }
                if (!modified) break;
        }
        bool hasNegativeCycle = false;
        for (int i = 1; i <= n; i++){
                if (dist[i] != INF){
                        for (auto nbr: adjList[i]){
                                int v = nbr.fi;
                                int weight = nbr.se;
                                if (dist[v] > dist[i] + weight){
                                        hasNegativeCycle = true;
                                }
                        }
                }
        }
        if (hasNegativeCycle){
                for (int i = 0; i <= n; i++){
                        dist[i] = -1;
                }
        }
        return dist;
}
```

Floyd-Warshall

```
void FW(vv &matrix, vv *p = NULL){
        int numVertices = (int) matrix.size();
        if (p){
                for (int i = 0; i < numVertices; i++){
                        for (int j = 0; j < numVertices; j++){
                                        p[i][j] = i;
                        }
                }
        }
```

```
        for (int k = 0; k < numVertices; k++){
                for (int i = 0; i < numVertices; i++){
                        for (int j = 0; j < numVertices; j++){
                                if (matrix[i][k] != INT_MAX && matrix[k][j] != INT_MAX){
                                        matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix
                                             ↪ [k][j]);
                                        if (p) p[i][j] = p[k][j];
                                }
                        }
                }
        }
}

void printPath(int i, int j){      //Nao sei se esta funcao esta 100% correta mas a ideia
     ↪ esta la
        if (i != j) printPath(i, p[i][j]);
        cout<<j<<endl;
}
```

Bipartite Matching

```
vv graph(1001);
vi color (1001, -1);

bool bipartite(int start){
        int cur;
        queue<int> Q;
        color[start] = 1;
        Q.push(start);

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

                for(auto u: graph[cur]){
                        if(color[u] == -1){
                                color[u] = 1 - color[cur];
                                Q.push(u);
                        }
                        else if(color[u] == color[cur])
                                return false;
                }
        }
        return true;
}

int main(){
        int m, u, v, start;
        cin >> m;
        forn(i, m){
                cin >> u >> v;
                graph[u].push_back(v);
                graph[v].push_back(u);
        }
        cin >> start;

        if (bipartite(start) == true)
                cout << "Yes\n";
        else
                cout << "No\n";

        return 0;
}
```

Max Bipartite Matching

```
vector<vector<int>> graph;
vector<int> match;

bool bpm(int u, vector<bool>& visited) {
        for(int v : graph[u]) {
                if(!visited[v]) {
                        visited[v] = true;
                        if(match[v] == -1 || bpm(match[v], visited)) {
                                match[v] = u;
                                return true;
                        }
                }
        }
        return false;
}

void solve(){
        int n, m;
        cin >> n >> m;
        graph.resize(n+1);

        int col;
        int add;
        for(int i = 1; i <= n; i++){
                cin >> col;

                for(int j = 0; j < col; j++){
                        cin >> add;
                        graph[i].push_back(add);
                }
        }

        match.assign(m + 1, -1);

        int result = 0;
        for(int u = 1; u <= n; u++) {
                vector<bool> visited(m + 1, false);
                if(bpm(u, visited))
                        result++;
        }

        cout << result << endl;
}
```

Topological Sort

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
	visited[v] = true;
	for (int u : adj[v]) {
		if (!visited[u])
			dfs(u);
	}
	ans.push_back(v);
}

void topological_sort() {
	visited.assign(n, false);
	ans.clear();
	for (int i = 0; i < n; ++i) {
		if (!visited[i]) {
			dfs(i);
		}
	}
	reverse(ans.begin(), ans.end());
}
```

Tarjan (Strongly Connected Components)

```
vector<vector<ll>> graph;
vector<vector<ll>> SCCs;
vector<bool> visited;
vector<ll> ids;
vector<ll> low;
ll counter;
stack<ll> S;
vector<bool> onStack;
ll id;

void dfs(ll cur){
	S.push(cur);
	onStack[cur] = true;
	ids[cur] = low[cur] = id;
	id++;

	for(auto adj: graph[cur]){
		if(ids[adj] == -1){
			dfs(adj);
		}
		// If statement after the DFS callback
		if(onStack[adj]){
			low[cur] = min(low[cur], low[adj]);
		}
	}

	// SCC root found
	ll top = -1;
	if(ids[cur] == low[cur]){
		vector<ll> newSCC;
		while(top != cur){
			top = S.top(); S.pop();
			onStack[top] = false;
			low[top] = ids[cur];
			newSCC.push_back(top);
		}
		SCCs.push_back(newSCC);
		counter++;
	}
}

void tarjan(){
	id = 1;
	counter = 0;

	for(ll i = 1; i <= n; ++i){
		if(ids[i] == -1){
			dfs(i);
		}
	}

}
```

Building Roads (Tarjan Example)

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

using namespace std;

typedef long long ll;

ll n, m;
vector<vector<ll>> graph;
vector<vector<ll>> SCCs;
vector<bool> visited;
vector<ll> ids;
vector<ll> low;
ll counter;
stack<ll> S;
vector<bool> onStack;
ll id;

void dfs(ll cur){
	S.push(cur);
	onStack[cur] = true;
	ids[cur] = low[cur] = id;
	id++;

	for(auto adj: graph[cur]){
		if(ids[adj] == -1){
			dfs(adj);
		}
		// If statement after the DFS callback
```

```
            if(onStack[adj]){
                low[cur] = min(low[cur], low[adj]);
            }
        }

        // SCC root found
        ll top = -1;
        if(ids[cur] == low[cur]){
            vector<ll> newSCC;
            while(top != cur){
                top = S.top(); S.pop();
                onStack[top] = false;
                low[top] = ids[cur];
                newSCC.push_back(top);
            }
            SCCs.push_back(newSCC);
            counter++;
        }
}

void tarjan(){
        id = 1;
        counter = 0;

        for(ll i = 1; i <= n; ++i){
            if(ids[i] == -1){
                dfs(i);
            }
        }

}

void solve(){
        cin >> n >> m;
        graph.resize(n+1);
        ids.resize(n+1, -1);
        low.resize(n+1, 0);
        onStack.resize(n+1, false);
        ll u, v;

        for(ll i = 0; i < m; ++i){
            cin >> u >> v;
            graph[u].push_back(v);
            graph[v].push_back(u);
        }

        tarjan();

        cout << counter-1 << endl;
        for(ll i = 0; i < counter-1; i++){
            cout << SCCs[i][0] << " " << SCCs[i+1][0] << endl;
        }
}
int main()
{
        ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
        ll t = 1;
        //cin >> t;
        for(int it=1;it<=t;it++) {
            //cout << "Case #" << it+1 << ": ";
            solve();
        }
        return 0;
}
```

Eulerian Path

```
//Para grafo direcionado, nao e preciso arestas. Guarda-se o vertices de saida diretamente
    ↪ na list. Outras mudancas sao necessarias
//Verificar se e conexo (dfs) e todos os vertices tem grau par. Para semi-eulariano, 2
    ↪ vertices com grau impar, restantes par
vi hierholzer(int s, vector<list<int>> &graph, vector<pair<pii, bool>> &arestas){
        int n = graph.size();
        vi ans, idx(n, 0), st;
        st.pb(s);
        while (!st.empty()){
            int u = st.back();
            //ciclo nao necessario para grafo direcionado
            while (!graph[u].empty() && arestas[graph[u].front()].se){
                graph[u].pop_front();
            }
            if (!graph[u].empty()){
                pii are = arestas[graph[u].front()].fi;
                if (are.fi == u) st.pb(are.se);
                else st.pb(are.fi);
                arestas[graph[u].front()].se = true;
                graph[u].pop_front();
            }else{
                ans.pb(u);
                st.pop_back();
            }
        }
        reverse(all(ans));
        return ans;
}
```

Max-Flow/Min-Cut

```
template<class T> void dfs(int s, vector<unordered_map<int, T>> &graph, vv &adjacency, vb
    ↪ &visited){
        visited[s] = true;
        for (int ver: adjacency[s]){
            if (!visited[ver] && graph[s][ver] != 0){
                dfs(ver, graph, adjacency, visited);
            }
        }
}

#define rep(i, a, b) for(int i = a; i < (b); ++i)
template<class T> T edmondsKarp(vector<unordered_map<int, T>>&graph, int source, int sink,
    ↪ vpii *arestas = NULL) {
```

```
	assert(source != sink);
	T flow = 0;
	vi par(sz(graph)), q = par;
	int n = graph.size();
	vv adjacency(n);
	if (arestas){
		for (int i = 0; i < n; i++){
			for (pii are: graph[i]){
				adjacency[i].pb(are.fi);
			}
		}
	}
	for (;;) {
		fill(all(par), -1);
		par[source] = 0;
		int ptr = 1;
		q[0] = source;

		rep(i,0,ptr) {
			int x = q[i];
			for (auto e : graph[x]) {
				if (par[e.first] == -1 && e.second > 0) {
				par[e.first] = x;
				q[ptr++] = e.first;
				if (e.first == sink) goto out;
				}
			}
		}
		if (arestas){
			vb visited(n, false);
			dfs(source, graph, adjacency, visited);
			for (int i = 0; i < n; i++){
				for (pair<int, T> ver: graph[i]){
					if (!visited[i] && visited[ver.fi] && graph[ver.fi][i]
					    ↪ == 0){
					   (*arestas).pb(mp(ver.fi, i));
					}
				}
			}
		}
		return flow;
out:
		T inc = numeric_limits<T>::max();
		for (int y = sink; y != source; y = par[y])
			inc = min(inc, graph[par[y]][y]);

		flow += inc;
		for (int y = sink; y != source; y = par[y]) {
			int p = par[y];
			if ((graph[p][y] -= inc) <= 0) graph[p].erase(y);
			graph[y][p] += inc;
		}
	}
}
```

MIUP 2022 B (Max-Flow/Min-Cut

Example)

```
void solve(){
	//reset e leitura de valores
	ll n, m;
	cin>>n>>m;
	//criar sempre um "novo" sink e source
	ll i_source = 0, i_sink = n*2 + 1;
	vi pop(n + 1);
	vi custos(n + 1);
	vector<unordered_map<int, ll>> graph((n+1)*2);
	for (ll i = 1; i <= n; i++){
		cin>>pop[i]>>custos[i];
		graph[(i*2) - 1][i*2] = custos[i];
	}
	while(m--) {
		//Se a aresta nao for de duplo sentido, o res do sentido contrario tem de
		    ↪ ser 0
		ll n_1, n_2;
		cin>>n_1>>n_2;
		graph[n_1*2][(n_2*2) - 1] = INF;
		graph[n_2*2][(n_1*2) - 1] = INF;
	}
	ll safe;
	cin>>safe;
	//ligar source e sink aos vertices necessarios
	for (ll i = 1; i <= n; i++){
		graph[i_source][(i*2) - 1] = pop[i];
	}
	graph[(safe*2) - 1][i_sink] = INF;
	ll maxFlow = edmondsKarp(graph, i_source, i_sink);
	cout<<maxFlow<<endl;
}
```

Min-Cost/Max-Flow

```
typedef tuple<int, ll, ll, ll> edge;
class min_cost_max_flow {
private:
	int V;
	ll total_cost;
	vector<edge> EL;
	vector<vi> AL;
	vll d;
	vi last, vis;

	bool SPFA(int s, int t) { // SPFA to find augmenting path in residual graph
	d.assign(V, INF); d[s] = 0; vis[s] = 1;
	queue<int> q({s});
	while (!q.empty()) {
		int u = q.front(); q.pop(); vis[u] = 0;
		for (auto &idx : AL[u]) { // explore neighbors of u
		auto &[v, cap, flow, cost] = EL[idx]; // stored in EL[idx]
		if ((cap-flow > 0) && (d[v] > d[u] + cost)) { // positive residual edge
			d[v] = d[u]+cost;
			if(!vis[v]) q.push(v), vis[v] = 1;
```

```
			}
			}
		}
		return d[t] != INF; // has an augmenting path
		}

		ll DFS(int u, int t, ll f = INF) { // traverse from s->t
		if ((u == t) || (f == 0)) return f;
		vis[u] = 1;
		for (int &i = last[u]; i < (int)AL[u].size(); ++i) { // from last edge
			auto &[v, cap, flow, cost] = EL[AL[u][i]];
			if (!vis[v] && d[v] == d[u]+cost) { // in current layer graph
			if (ll pushed = DFS(v, t, min(f, cap-flow))) {
						total_cost += pushed * cost;
				flow += pushed;
				auto &[rv, rcap, rflow, rcost] = EL[AL[u][i]^1]; // back edge
				rflow -= pushed;
				vis[u] = 0;
				return pushed;
			}
			}
		}
		vis[u] = 0;
		return 0;
		}

public:
		min_cost_max_flow(int initialV) : V(initialV), total_cost(0) {
		EL.clear();
		AL.assign(V, vi());
		vis.assign(V, 0);
		}

		// if you are adding a bidirectional edge u<->v with weight w into your
		// flow graph, set directed = false (default value is directed = true)
		void add_edge(int u, int v, ll w, ll c, bool directed = true) {
		if (u == v) return; // safeguard: no self loop
		EL.emplace_back(v, w, 0, c); // u->v, cap w, flow 0, cost c
		AL[u].push_back(EL.size()-1); // remember this index
		EL.emplace_back(u, 0, 0, -c); // back edge
		AL[v].push_back(EL.size()-1); // remember this index
		if (!directed) add_edge(v, u, w, c); // add again in reverse
		}

		pair<ll, ll> mcmf(int s, int t) {
		ll mf = 0; // mf stands for max_flow
		while (SPFA(s, t)) { // an O(V^2*E) algorithm
			last.assign(V, 0); // important speedup
			while (ll f = DFS(s, t)) // exhaust blocking flow
			mf += f;
		}
		return {mf, total_cost};
		}
};

void solve(){
		int v, e, s, t;
		cin>>v>>e>>s>>t;
		min_cost_max_flow mf(v);
		for (int i = 0; i < e; i++){
			int a, b, cap, cost;
			cin>>a>>b>>cap>>cost;
			mf.add_edge(a, b, cap, cost);
		}
		pll res = mf.mcmf(s, t);
		cout<<res.fi<<' '<<res.se<<endl;
}
```

MIUP 2023 E (Min-Cost/Max-Flow Example)

```
void solve(){
		int d, n, c, m, vals, valc, source = 0;
		cin>>d>>n>>c>>m;
		vi capacity(n+1);
		int sink = c + n + 1, maxProfit = 100;
		min_cost_max_flow mf(c + n + 2);
		for (int i = 1; i <= n; i++){
				cin>>capacity[i];
		}
		vi shipTime(n+1);
		for (int i = 1; i <= n; i++){
				cin>>shipTime[i];
				int count = 0, aux = d;
				while ((aux > 0)){
					aux -= shipTime[i];
					int cap = capacity[i];
					while ((aux > 1) && (cap > 0)){
						aux-=2;
						cap--;
						count++;
					}
					aux -= shipTime[i];
				}
				capacity[i] = count;
		}
		vi lucro(c+1);
		for (int i = 1; i <= c; i++){
			cin>>lucro[i];
		}
		for (int i = 0; i < m; i++){
			cin>>vals>>valc;
			mf.add_edge(valc, c + vals, 1, maxProfit-lucro[valc]);
		}
		for (int i = 1; i <= n; i++){
			mf.add_edge(c + i, sink, capacity[i], 0);
		}
		for (int i = 1; i <= c; i++){
			mf.add_edge(source, i, 1, 0);
		}
		pll res = mf.mcmf(source, sink);
```

```
	cout<<res.fi*maxProfit - res.se<<endl;
}
```

Articulation Points

```
void AP(int v, vv &adj, vb &check, vi &dfs, vi &low, vi &parent, int &t, int &c){
	low[v] = dfs[v] = t++;
	for (auto nbr: adj[v]){
		if (dfs[nbr] == 0){
			parent[nbr] = v;
			AP(nbr, adj, check, dfs, low, parent, t, c);
			low[v] = min(low[v], low[nbr]);
			if (!check[v]){
				if (dfs[v] == 1){
					if (dfs[nbr] != 2) c++;
				}else{
					if (low[nbr] >= dfs[v]) c++;
				}
			}
			check[v] = true;
		}else if (parent[v] != nbr){
			low[v] = min(low[v], dfs[nbr]);
		}
	}
}

void solve(){
	int n, m;
	cin>>n>>m;
	vv adj(n+1);
	vb check(n+1);
	vi dfs(n+1, 0);
	vi low(n+1, -1);
	vi parent(n+1, -1);
	int t = 1;
	int c = 0;
	AP(1, adj, check, dfs, low, parent, t, c);
}
```

Kruskal (Minimum Spanning Tree)

```
//Cada valor comeca por ser o seu proprio set
void makeSet(int v, vi &parent) {
	parent[v] = v;
}
int findSet(int v, vi &parent) {
	if (v != parent[v]) parent[v] = findSet(parent[v], parent);
	return parent[v];
}
void unionSets(int u, int v, vi &parent) {
	int root1 = findSet(u, parent);
	int root2 = findSet(v, parent);
	parent[root2] = root1;
}
bool check(int u, int v, vi &parent) {
	return findSet(u, parent) == findSet(v, parent);
}

template<class T> T KruskalMST(vector<tuple<T, int, int>> edges, int V){
	sort(all(edges));
	vi parent(V);
	for (int i = 0; i < V; i++){
		makeSet(i, parent);
	}
	T mst_cost = 0, num_taken = 0;
	for (auto &[w, u, v]: edges){
		if (check(u, v, parent)) continue;
		mst_cost += w;
		unionSets(u, v, parent);
		++num_taken;
		if (num_taken == V-1) break;
	}
	return mst_cost;
}
```

Lowest Common Ancestor

```
void dfs(int cur, int depth, vv &adjMatrix, vb &visited, vi &L, vi &E, vi &H, int &idx){
	H[cur] = idx;
	E[idx] = cur;
	L[idx++] = depth;
	visited[cur] = true;
	for (int nxt: adjMatrix[cur]){
		if (!visited[nxt]){
			dfs(nxt, depth+1, adjMatrix, visited, L, E, H, idx);
			E[idx] = cur;
			L[idx++] = depth;
		}
	}
}

void buildRMQ(int n, vv &adjMatrix, int m){
	vi L(2*n), E(2*n), H(n, -1);
	vb visited(n, false);
	int idx = 0;
	dfs(0, 0, adjMatrix, visited, L, E, H, idx);
	//LCA(i, j) e o E[ indice do min( L(H[i]...H[j]) ) ]. Para isto usamos uma SegTree
	    ↪ ou SparseTable em L (E[SpT.query(H[i], H[j])])
	SparseTable Spt = SparseTable(L);
	int a, b;
	for (int i = 0; i < m; i++){
		cin>>a>>b;
		a--;b--;
		int lca = E[Spt.RMQ(min(H[a], H[b]), max(H[a], H[b]))];
	}
}
```

— MATH —

Cicle Finding

```
	int f(int x){ //Avancar na expressao onde estamos a encontrar ciclo
	return (26*x + 11)%80;
}

pii floydCicleFinding(int x){ //Index (x) onde comeca a sequencia (arr)
	int t = f(x), h = f(f(x));
	while (t != h){
		t = f(t);
		h = f(f(h));
	}
	int fase = 0, h = x;
	while (t != h){
		t = f(t);
		h = f(h);
		fase++;
	}
	int T = 1;
	h = f(t);
	while (t != h){
		h = f(h);
		T++;
	}
	return mp(T, fase);
}
```

Count Digits

```
int countDigits(double num, double baseNum, double baseNova){
	return floor(1 + log(num)/log(baseNova));
}
```

Max Range Sum (1D and 2D)

```
ll maxRangeSum1D(int n, vll &arr){
	ll ans = 0;
	//limpeza dos negativos
	ans = arr[0];
	for (int j = 0; j < n; j++){
		if (arr[j] >= 0){
			ans = 0;
			break;
		}else{
			if (arr[j] > ans) ans = arr[j];
		}
	}
	if (ans < 0) return ans;
	//fim de limpeza
	ans = 0;
	ll sum = 0;
	for (int j = 0; j < n; j++){
		sum += arr[j];
		ans = max(ans, sum);
		if (sum < 0) sum = 0;
	}
	return ans;
}

// ------------------------------

ll maxRangeSum2D(int n, vvll &mat){
	for (int i = 0; i < n; i++){
		for (int j = 1; j < n; j++){
			mat[i][j] += mat[i][j-1];
		}
	}
	ll maior = -INF;
	for (int i = 0; i < n; i++){
		for (int j = i; j < n; j++){
			ll subrect = 0;
			for (int k = 0; k < n; k++){
				if (i > 0) subrect += mat[k][j] - mat[k][i-1];
				else subrect += mat[k][j];
				if (subrect < 0) subrect = 0;
				maior = max(maior, subrect);
			}
		}
	}
	return maior;
}
```

Max Subarray Sum

```
ll MaximumSubarraySumN(int n, vll &arr){
	ll maior = 0;
	//limpeza dos negativos
	maior = arr[0];
	for (int j = 0; j < n; j++){
		if (arr[j] >= 0){
			maior = 0;
			break;
		}else{
			if (arr[j] > maior) maior = arr[j];
		}
	}
	if (maior < 0) return maior;
	//fim de limpeza
	ll atual = 0, cache = -1, flag = 0;
	for (int j = 0; j < n; j++){
		if ((atual + arr[j]) < 0){
			if (cache != -1){
				if (cache > maior) maior = cache;
				cache = -1;
				flag = 0;
			}else{
				if (atual > maior) maior = atual;
			}
			atual = 0;
		}else{
			if ((atual + arr[j] >= atual) || flag){
```

```
                        atual += arr[j];
                        if (atual > cache){
                            cache = -1;
                            flag = 0;
                        }
                    }else{
                        cache = atual;
                        atual += arr[j];
                        flag = 1;
                    }
                }
            }
    if (cache != -1){
        if (cache > maior) maior = cache;
    }else{
        if (atual > maior) maior = atual;
    }
    return maior;
}
```

— MODULAR / MATRICES —

Modular Arithmetic

```
// Modular function to avoid negative results
inline int mod(int a, int m) {
    return ((a % m) + m) % m;
}

int modPow(int b, int p, int m){
    if (p == 0) return 1;
    int ans = modPow(b, p/2, m);
    ans = mod(ans*ans, m);
    if (p&1) ans = mod(ans*b, m);
    return ans;
}

int modInverse(int A, int M){
    int m0 = M;
    int y = 0, x = 1;

    if (M == 1)
        return 0;

    while (A > 1) {
        // q is quotient
        int q = A / M;
        int t = M;

        // m is remainder now, process same as
        // Euclid's algo
        M = A % M, A = t;
        t = y;
```

```
        // Update y and x
        y = x - q * y;
        x = t;
}
```

Matrix Operations

```
vvll matMul(vvll &a, vvll &b, int MOD){ //Duas matrizes nao nulas, i -> linhas, j ->
    ↪ colunas
    int lin = a.size();
    int col = b[0].size();
    vvll ans(lin, vll(col, 0));
    int par = b.size();
    for (int i = 0; i < lin; i++){
        for (int k = 0; k < par; k++){
            if (a[i][k] == 0) continue;
            for (int j = 0; j < col; j++){
                ans[i][j] += mod(a[i][k], MOD) * mod(b[k][j], MOD);
                ans[i][j] = mod(ans[i][j], MOD);
            }
        }
    }
    return ans;
}

vvll matPow(vvll base, int p, int MOD){ //So matrizes quadradas
    int lin = base.size();
    vvll ans(lin, vll(lin));
    for (int i = 0; i < lin; i++){
        for (int j = 0; j < lin; j++){
            ans[i][j] = (i == j);
        }
    }
    while (p){
        if (p&1){
            ans = matMul(ans, base, MOD);
        }
        base = matMul(base, base, MOD);
        p >>= 1;
    }
    return ans;
}
```

Gaussian Elimination

```
#define MAX_N 100 //adjust this value as needed
struct AugmentedMatrix{ double mat[MAX_N][MAX_N + 1];};
struct ColumnVector{ double vec[MAX_N];};
ColumnVector GaussianElimination(int N, AugmentedMatrix Aug){ //O(n^3)
    //input: N, Augmented Matriz aug; output: Column Vector x, the answer
    for (int i = 0; i < N-1; i++){      //forward elimination
        int l = i;
        for (int j = i + 1; j < N; j++){          row with max col value
            if (fabs(Aug.mat[j][i]) > fabs(Aug.mat[l][i])) l = j;      remember
                ↪ this row l
```

```
        }
        //swap this pivot row, reason: minimize floating point error
        for (int k = i; k <= N; k++){
            swap(Aug.mat[i][k], Aug.mat[l][k]);
        }
        for (int j = i+1; j < N; j++){     //actual fwd elimination
            for (int k = N; k >= i; k--){
                Aug.mat[j][k] -= Aug.mat[i][k] * Aug.mat[j][i] / Aug.mat[i][i
                    ↪ ];
            }
        }
    }
    ColumnVector Ans;        //back substitution phase
    for (int j = N-1; j >= 0; j--){     //start from back
        double t = 0.0;
        for (int k = j+1; k < N; k++){
            t += Aug.mat[j][k] * Ans.vec[k];
        }
        Ans.vec[j] = (Aug.mat[j][N]-t) / Aug.mat[j][j];    //the answer is here
    }
    return Ans;
}

int main(){
    AugmentedMatrix Aug;
    Aug.mat[0][0] = 1; Aug.mat[0][1] = 1; Aug.mat[0][2] = 2; Aug.mat[0][3] = 9; //x + y
        ↪  + 2z = 9
    Aug.mat[1][0] = 2; Aug.mat[1][1] = 4; Aug.mat[1][2] = 3; Aug.mat[1][3] = 1; //2x +
        ↪ 4y - 3z = 1
    Aug.mat[2][0] = 3; Aug.mat[2][1] = 6; Aug.mat[2][2] = 5; Aug.mat[2][3] = 0; //3x +
        ↪ 6y - 5z = 0
    ColumnVector X = GaussianElimination(3, Aug);
    cout<<"x = "<<X.vec[0]<<endl;
    cout<<"y = "<<X.vec[1]<<endl;
    cout<<"z = "<<X.vec[2]<<endl;
}
```

— Number Theory —

Combinatorics

```
int modInverse(int A, int M){
    int m0 = M;
    int y = 0, x = 1;

    if (M == 1)
        return 0;

    while (A > 1) {
        // q is quotient
        int q = A / M;
        int t = M;
```

```
            // m is remainder now, process same as
            // Euclid's algo
            M = A % M, A = t;
            t = y;

            // Update y and x
            y = x - q * y;
            x = t;
        }

        // Make x positive
        if (x < 0)
            x += m0;

        return x;
}

vpll fat;

void fatoriais(int tam, int m, vpll &res){
    res.pb(mp(1,1));
    for (int j = 1; j <= tam; j++){
        res.pb(mp((res[j-1].fi*j)%m, 0));
    }
    ll inv = modInverse(res[tam].fi, m);
    res[tam].se = inv;
    for (int j = tam-1; j > 0; j--){
        res[j].se = (res[j+1].se*(j+1))%m;
    }
}

ll comb(int c, int d, int m){
    if (d == 0) return 1;
    if ((d > 0) && (d > c)) return 0;
    return (((fat[c].fi*fat[d].se)%m)*fat[c-d].se)%m;
}
fatoriais(5000, MOD, fat); //Colocar dentro da main
```

Number Theory

```
int extEuclidean(int a, int b, int &x, int &y){
    int xx = y = 0;
    int yy = x = 1;
    while (b){
        int q = a/b;
        int t = b;
        b = a%b;
        a = t;
        t = xx;
        xx = x-q*xx;
        x = t;
        t = yy;
        yy = y - q*yy;
        y = t;
    }
    return a;
```

```
}

int modInverse(int A, int M){ //Para combinacoes/fatoriais, escrever comb ou fatoriais
	int x, y;
	int d = extEuclidean(A, M, x, y);
	if (d != 1) return -1;
	return mod(x, M);
}

di
pii diophantine(int a, int b, int sol){ //a*x + b*y = sol
	int x, y;
	int d = extEuclidean(a, b, x, y); //gcd(a, b)
	int mult = sol/d;
	x *= mult;
	y *= mult;
	b /= d;
	a /= d;
	int liminf = 0, limsup = INF;
	if ((x < 0) != (b < 0)){
		liminf = abs(x/b);
		if (x%b) liminf++;
	}else{
		limsup = abs(x/b);
	}
	if ((y < 0) != (a < 0)){
		int aux = abs(y/a);
		if (y%a) aux++;
		liminf = max(liminf, aux);
	}else{
		limsup = min(limsup, abs(y/a));
	}
	if (liminf > limsup) return mp(-1, -1); //So devolve uma solucao para a equacao,
	    ↪ mas ha um limite (finito ou infinito de solucoes)
	else return mp(x + b*liminf, y + a*liminf);
}

int crt(vi &r, vi &m){
	int mt = accumulate(m.begin(), m.end(), 1, multiplies<>());
	int x = 0;
	for (int i = 0; i < (int) m.size(); i++){
		int a = mod((ll)r[i] * modInverse(mt/m[i], m[i]), m[i]);
		x = mod(x + (ll)a * (mt/m[i]), mt);
	}
	return x;
}

vll Catalan(int n, ll m){ //n inclusive
	vll cat(n+1);
	cat[0] = 1;
	for (int i = 0; i < n; i++){
		cat[i+1] = mod(mod(mod((4*i)+2,m) * mod(cat[i],m), m) * modInverse(i+2, m),
		    ↪ m);
	}
	return cat;
}
```

```
inline long long int gcd(int a, int b){
	while (b) {
		a %= b;
		swap(a, b);
	}
	return a;
}

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

Primes

```
ll sieve_size;
bitset<10000010> bs;
vll p;

void gerador(ll upperbound){ //Nao maior de 10^7
	sieve_size = upperbound+1;
	bs.set();
	bs[0] = bs[1] = 0;
	for (ll i = 0; i < sieve_size; i++){
		if (bs[i]){
			for (ll j = i*i; j < sieve_size; j+=i) bs[j] = 0;
			p.push_back(i);
		}
	}
}

bool isPrime(ll N){
	if (N < sieve_size) return bs[N];
	for (int i = 0; i < (int) p.size() && p[i]*p[i] <= N; i++){
		if (N%p[i] == 0) return false;
	}
	return true;
}

//Por no solve
gerador(10000000);

vll primeFactor(ll N){ //Fatorizar em numeros primos, nao esquecer de gerar numeros primos
	vll factors;
	int tam = p.size();
	for (int i = 0; (i < tam) && (p[i]*p[i] <= N); i++){
		while (N%p[i] == 0){
			N /= p[i];
			factors.pb(p[i]);
		}
	}
	if (N != 1) factors.pb(N);
	return factors;
}

int numFatPrimos(ll N){ //Quantos fatores primos tem um numero
```

```
	int ans = 1;
	for (int i = 0; (i < (int) p.size()) && (p[i]*p[i] <= N); i++){
		while (N%p[i] == 0) {
			N/=p[i];
			ans++;
		}
	}
	return ans + (N != 1);
}

int numDivisores(ll N){ //Multiplicatorio de (n+1), sendo 'n' o numero de vezes que cada
    ↪ fator primos aparece
	int ans = 0;
	for (int i = 0; (i < (int) p.size()) && (p[i]*p[i] <= N); i++){
		int power = 0;
		while (N%p[i] == 0){
			N /= p[i];
			++power;
		}
		ans *= power+1;
	}
	return (N != 1) ? 2*ans : ans;
}

ll sumDivisores(ll N){ //Multiplicatorio de (a^(n+1) - 1)/(a-1), sendo 'a' cada fator
    ↪ primo e 'n' o numero de vezes que 'a' se repete
	ll ans = 1;
	for (int i = 0; (i < (int) p.size()) && (p[i]*p[i] <= N); i++){
		ll multiplier = p[i], total = 1;
		while (N%p[i] == 0){
			N /= p[i];
			total += multiplier;
			multiplier *= p[i];
		}
		ans *= total;
	}
	if (N != 1) ans *= (N+1);
	return ans;
}

ll numCoprimos(ll N){ //N * Multiplicatorio de (1 - 1/a), sendo 'a' cada fator primo de N
	ll ans = N;
	for (int i = 0; (i < (int) p.size()) && (p[i]*p[i] <= N); i++){
		if (N%p[i] == 0) ans -= ans/p[i];
		while (N%p[i] == 0) N/=p[i];
	}
	if (N != 1) ans -= ans/N;
	return ans;
}

vi numDiffFatPrimos(ll MAX_N){ //MAX_N <= 10^7 Numero de fatores primos diferentes para mt
    ↪  queries
	vi arr(MAX_N + 10, 0);
	for (int i = 2; i <= MAX_N; i++){
		if (arr[i] == 0){
			for (int j = i; j <= MAX_N; j+=i){
				++arr[j];
			}
		}
	}
	return arr;
}

vi numCoprimosMtQueries(ll Max_n){ //Max_n <= 10^7 Numero de coprimos para mt queries
	vi arr(Max_n);
	for (int i = 1; i <= Max_n; i++) arr[i] = i;
	for (int i = 2; i <= Max_n; i++){
		if (arr[i] == i){
			for (int j = i; j <= Max_n; j+=i){
				arr[j] = (arr[j]/i) * (i-1);
			}
		}
	}
	return arr;
}
```

— Strings —

Aho-Corasick

```
string text; //Text
int n; //Size of text,
int k; //Number of keys
int maxs = 0; // Should be equal to the sum of the length of all keywords.
int maxc = 26; // Maximum number of characters in input alphabet

// Returns the number of states that the built machine has.
// States are numbered 0 up to the return value - 1, inclusive .
int buildMatchingMachine(string arr[], int k, vector<map<int, bool>> &out, vi &f, vv &g){
	int states = 1;
	for ( int i = 0; i < k; ++i){ // Construct values for goto function, i .e ., fill g
		const string &word = arr[i];
		int currentState = 0;
		for ( int j = 0; j < (int) word.size(); ++j){
			int ch = word[j]-'a';
			if (g[currentState][ch] == -1){ // Allocate a new node (create a new
			    ↪  state) if a node for ch doesnt exist .
				g[currentState][ch] = states++;
			}
			currentState = g[currentState][ch];
		}
		out[currentState][i] = true; // Add current word in output function
	}
	for ( int ch = 0; ch < maxc; ++ch){
		if (g[0][ch] == -1){
			g[0][ch] = 0;
		}
	}
	queue<int> q;
```

```
	for ( int ch = 0; ch < maxc; ++ch){
		if (g[0][ch] != 0){
			f [g[0][ch]] = 0;
			q.push(g[0][ch]) ;
		}
	}
	while (q.size () ) {
		int state = q.front () ;
		q.pop();
		for ( int ch = 0; ch < maxc; ++ch){
			if (g[state][ch] != -1){
				int failure = f [state];
				while (g[ failure][ch] == -1){ // Find the deepest node
				    ↪ labeled by proper suffix of string from root to
				    ↪ current state .
					failure = f [ failure ];
				}
				failure = g[failure][ch];
				f [g[state][ch]] = failure ;
				for (pair<int, bool> par: out[failure]){
					out[g[state][ch]][par.fi] = par.se;
				}
				q.push(g[state][ch]) ;
			}
		}
	}
	return states ;
}

int findNextState(int currentState, char nextInput, vector<map<int, bool>> &out, vi &f, vv
    ↪  &g){ //Returns the next state the machine will transition to using goto and
    ↪ failure functions.
	int answer = currentState;
	int ch = nextInput -'a';
	while (g[answer][ch] == -1){
		answer = f[answer];
	}
	return g[answer][ch];
}

void searchWords(string arr[], int k, string text, vector<map<int, bool>> &out, vi &f, vv
    ↪ &g, vv &ocor, vi &tam) {
	buildMatchingMachine(arr, k, out, f, g); // Build machine with goto, failure and
	    ↪ output functions
	int currentState = 0;
	for ( int i = 0; i < (int) text.size() ; ++i){
		currentState = findNextState(currentState, text[i], out, f, g) ;
		/*if (out[currentState] == 0){ // If match not found, move to next state,
		    ↪ uncomment if number of keys is less of 64
			continue;
		}*/
		for (pair<int, bool> par: out[currentState]){ // Match found, print all
		    ↪ matching words of arr[]
			ocor[i-tam[par.fi]+1].pb(par.fi);
		}
	}
}

void solve(){
	cin>>text;
	n = (int) text.size();
	vv ocor(n); //To store the index where each key starts in texts
	cin>>k;
	string arr[k]; //Stores every key
	vi tam(k); //Stores every key size
	for (int j = 0; j < k; j++){
		cin>>arr[j];
		tam[j] = arr[j].size();
		maxs += tam[j];
	}
	vector<map<int, bool>> out(maxs); // Stores the word number for each state (letter
	    ↪ in text)
	//vi out(maxs, 0); // Bit i in this mask is one if the word with index i in that
	    ↪ state. To use if there are less than 64 keys
	vi f (maxs, -1); // FAILURE FUNCTION IS IMPLEMENTED USING f[]
	vv g (maxs, vi(maxc, -1)); // GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[][]
	searchWords(arr, k, text, out, f, g, ocor, tam); // Each state (char in text) has
	    ↪ the key numbers of the keys that start in that state in ocor
	return;
}
```

Word Combination (Aho-Corasick Example)

```
void solve(){
	cin>>text;
	n = (int) text.size();
	vv ocor(n); //To store the index where each key starts in texts
	cin>>k;
	string arr[k]; //Stores every key
	vi tam(k); //Stores every key size
	for (int j = 0; j < k; j++){
		cin>>arr[j];
		tam[j] = arr[j].size();
		maxs += tam[j];
	}
	vector<map<int, bool>> out(maxs); // Stores the word number for each state (letter
	    ↪ in text)
	//vi out(maxs, 0); // Bit i in this mask is one if the word with index i in that
	    ↪ state. To use if there are less than 64 keys
	vi f (maxs, -1); // FAILURE FUNCTION IS IMPLEMENTED USING f[]
	vv g (maxs, vi(maxc, -1)); // GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[][]
	searchWords(arr, k, text, out, f, g, ocor, tam); // Each state (char in text) has
	    ↪ the key numbers of the keys that start in that state in ocor
	return;
}
```

Edit Distance

```
int EditDistance(string a, string b, int tamA, int tamB){
```

```
	vv bu(tamA + 1, vi(tamB + 1, 0));
	for (int i = 0; i <= tamA; i++){
		bu[i][0] = i;
	}
	for (int i = 0; i <= tamB; i++){
		bu[0][i] = i;
	}
	for (int i = 1; i <= tamA; i++){
		for (int j = 1; j <= tamB; j++){
			if (a[i-1] == b[j-1]) bu[i][j] = min(min(bu[i-1][j-1], bu[i-1][j] +
			    ↪ 1), bu[i][j-1] + 1);
			else bu[i][j] = min(min(bu[i-1][j] + 1, bu[i][j-1] + 1), bu[i-1][j
			    ↪ -1] + 1);
		}
	}
	return bu[tamA][tamB];
}
```

KMP

```
string T, P; // T = text, P = pattern
int n, m; // n = |T|, m = |P|

void kmpPreprocess(vi &b) { // call this first
	int i = 0, j = -1; b[0] = -1; // starting values
	while (i < m) { // pre-process P
	while ((j >= 0) && (P[i] != P[j])) j = b[j]; // different, reset j
		++i; ++j; // same, advance both
		b[i] = j;
	}
}

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

void solve(){
	cin>>T;
	cin>>P;
	n = (int) T.size();
	m = (int) P.size();
	vi b(m+1); // b = back table
	kmpPreprocess(b);
	kmpSearch(b);
}
```

String Matching (KMP Example)

```
void solve(){
   cin>>T;
   cin>>P;
   n = (int) T.size();
   m = (int) P.size();
   vi b(m+1); // b = back table
   kmpPreprocess(b);
   kmpSearch(b);
}
```

Longest Common Subsequence

```
int LCS(string a, string b, int tamA, int tamB){
	vv bu(tamA + 1, vi(tamB + 1, 0));
	for (int i = 1; i <= tamA; i++){
		for (int j = 1; j <= tamB; j++){
			if (a[i-1] == b[j-1]) bu[i][j] = bu[i-1][j-1] + 1;
			else bu[i][j] = max(bu[i-1][j], bu[i][j-1]);
		}
	}
	return bu[tamA][tamB];
}
```

— MISCELLANEOUS —

Binary Search

```
bool F(ll target){
   return true or false;
}

ll bestXforF (){
   ll leftBound = 0, rightBound = 1, mid;

   while(F(rightBound) == false)
      rightBound *= 2;

   while(rightBound > leftBound + 1){
      mid = leftBound + (rightBound - leftBound)/2;
      if(F(mid) == true)
         rightBound = mid;
      else
         leftBound = mid;
   }

   return leftBound;
}
```

Permutations

```
//l = 0, r = n-1
void permute(vector<int> &a, int l, int r){
```

```
	if (l >= r){
		//verificar permutacao, guarda-la leva a MLE
		verifica();
	}
	else{
		//Fazer todas as permutacoes
		for (int i = l; i <= r; i++){
			swap(a[l], a[i]);
			permute(a, l+1, r);
			swap(a[l], a[i]);
		}
	}
}
```