Team notebook

UNAL

October 19, 2024

Contents

1 Data-structures

1.1 BIT

```
/*
Fenwick Tree (BIT)

Range and update queries of a group operation
in O(log n).
*/
template<typename T>
struct BIT{
  vector<T> bit;
  int n;
  BIT(int n){
    bit.assign(n, 0);
    this->n = n;
  }

  BIT (vector<T> &a) : BIT(a.size()){
    for (int i =0 ;i < n ; i++){
      bit[i] += a[i];
      int r = i | (i + 1);
      if (r < n) bit[r] += bit[i];
    }
  }

  // Increments ith element by val
  void update(int i, T val){
    for (; i < n; i = i | (i + 1))
      bit[i] += val;
  }

  // Get A[0, r]
  T query(int r){
    T ret = 0;
    for (; r >= 0; r = (r & (r + 1)) - 1)
      ret += bit[r];
    return ret;
  }

  // Get A[l, r]
  T query(int l, int r){
    return query(r) - query(l - 1);
  }
};
```

1.2 dsu

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

/*
DSU data structure:
- Create disjunct sets of nodes, and answer
queries on components of nodes in a graph.

Source: https://cp-algorithms.com/data_structures/disjoint_set_union.html

Complexity:
- Find set: O(lg*n) >= O(1)
- make_set: O(1)
- union_set: O(lg*n) >= O(1)
*/

const int N = 1e6;

int parent[N];
int rank_dsu[N];

// Finds representative of DSU
int find_set(int v) {
  return parent[v] = (v == parent[v] ? v : find_set(parent[v]));
}

void make_set(int v){
  parent[v] = v;
  rank_dsu[v] = 0;
}
```

```
void union_set(int u, int v){
  u = find_set(u);
  v = find_set(v);
  if (u != v){
    // Add tree of lower rank_dsued node to the higher one
    if (rank_dsu[v] > rank_dsu[u])
      swap(u, v);

    parent[v] = u;
    if (rank_dsu[v] == rank_dsu[u])
      rank_dsu[u]++;
  }
}
```

1.3 hilbert$_o$rder$_o$ptimization

```
/*
Creates optimal ordering for
mo's algorithm queries.

Changes complexity from O((n + q)sqrt(n)) to: O(n*sqrt(q))

Creates query objects as:
queries[i].l = l;
queries[i].r = r;
queries[i].ind = i;
queries[i].calcOrder();
*/
inline int64_t hilbertOrder(int x, int y, int pow, int rotate) {
	if (pow == 0) {
		return 0;
	}
	int hpow = 1 << (pow-1);
	int seg = (x < hpow) ? (
		(y < hpow) ? 0 : 3
	) : (
		(y < hpow) ? 1 : 2
	);
	seg = (seg + rotate) & 3;
	const int rotateDelta[4] = {3, 0, 0, 1};
	int nx = x & (x ^ hpow), ny = y & (y ^ hpow);
	int nrot = (rotate + rotateDelta[seg]) & 3;
	int64_t subSquareSize = int64_t(1) << (2*pow - 2);
	int64_t ans = seg * subSquareSize;
	int64_t add = hilbertOrder(nx, ny, pow-1, nrot);
	ans += (seg == 1 || seg == 2) ? add : (subSquareSize - add - 1);
	return ans;
}

struct Query {
	int l, r, ind;
	int64_t ord;

	void calcOrder() {
		ord = hilbertOrder(l, r, 21, 0);
	}

  bool operator < (Query &b){
    return ord < b.ord;
  }
};
```

1.4 lazy$_m$inimum$_c$ount$_s$egtree

```
// Lazy update segment tree
// Count the number of minimums (Assumes there are no negative values:
    Count number of zeroes)
struct node{
  int mn, cnt;
  node(){
    mn = 0;
    cnt = 1;
  }
  node (int m, int c){
    mn = m;
    cnt = c;
  }
};



struct segtree{
  vector<node> t;
  vector<int> lazy;
  int N;
  segtree(int n){
```

```
  N= n;
  t.assign(4 * n, node());
  lazy.assign(4 * n, 0);
  build(1, 0, N - 1);
}

node combine(node a, node b){
  return node(min(a.mn, b.mn), (a.mn == b.mn ? a.cnt + b.cnt : (a.mn <
      b.mn ? a.cnt : b.cnt)));
}

void build(int v, int tl, int tr){
  if (tl == tr){
    return;
  }
  int tm = (tl + tr) / 2;
  build(2 * v, tl, tm);
  build(2 * v + 1, tm + 1, tr);
  t[v] = combine(t[2 * v], t[2 * v + 1]);
}

void push(int v){
  t[2 * v].mn += lazy[v];
  t[2 * v + 1].mn += lazy[v];
  lazy[2 * v] += lazy[v];
  lazy[2 * v + 1] += lazy[v];

  lazy[v] = 0;
}

void update(int v, int tl, int tr, int l , int r, int val){
  if (l > r) return;
  if (tl == l && tr == r){
    t[v].mn += val;
    lazy[v] += val;
    return;
  }
  push(v);
  int tm = (tl + tr) / 2;
  update(2 * v, tl, tm, l, min(tm, r), val);
  update(2 * v + 1, tm + 1, tr, max(l, tm + 1), r, val);
  t[v] = combine(t[2 * v], t[2 * v + 1]);
}

void update(int l, int r, int val){
  update(1, 0, N - 1, l, r, val);
}

// mn, cnt in a range
node query(int v, int tl, int tr, int l, int r){
  if (l > r) return node(inf, 0);
  if (tl == l && tr == r){
    return t[v];
  }
  int tm = (tl + tr) / 2;
  // need to push in order to calculate from children
  push(v);
  return combine(
    query(2 * v, tl , tm, l, min(r, tm)),
    query(2 * v + 1, tm + 1 , tr, max(l, tm + 1), r)
  );
}

node query(int l, int r){
  return query(1, 0, N - 1, l, r);
}

// Count zeroes
int get(int l, int r){
  node temp = query(l, r);
  return (temp.mn == 0 ? temp.cnt : 0);
}
};
```

1.5 lazy$_{p}$*rogagation*

```
// Problem: Add to a range in the array, and query an element
// Build the segment by just assigning the values of the array
// to the leafs, then to update add the value to the nodes corresponding
// to the range given
// To answer the query traverse the tree adding up the values in t

const int mxn= 2e5+ 2;
ll t[mxn * 4];

void build(int v, int l, int r, int a[]){
  if (l == r){
    t[v] = a[l];
```

```
    return;
  }
  int m = (l + r) / 2;
  build(2 * v, l, m, a);
  build(2 * v + 1, m + 1, r, a);
}

void update(int v, int tl, int tr, int l, int r, int add){
  if (l > r) return;
  if (tl == l && tr == r){
    t[v] += add;
    return;
  }
  int tm = (tl + tr) / 2;
  update(2 * v, tl, tm, l, min(r, tm), add);
  update(2 * v + 1, tm + 1, tr, max(l, tm + 1), r, add);
}

ll query(int v, int tl, int tr, int k){
  if (tl == tr){
    return t[v];
  }

  int tm = (tl + tr) / 2;

  if (k <= tm){
    return t[v] + query(2 * v, tl, tm, k);
  }
  else{
    return t[v] + query(2 * v + 1, tm + 1, tr, k);
  }
}
```

1.6 mos$_{algorithm}$

```
/*
Answer offlines queries based on an ordering that guarantees
O((n + q)sqrt(n))

Set block = sqrt(n)
Uses global variables and other data structures if needed.

Used when: Can transition between [l +-1,r+-1] rapidly.
*/
int block;
struct Query{
  int l, r, ind;

  bool operator < (Query other) const{
    return make_pair(l / block, r) < make_pair(other.l/block, other.r);
  }
};

// Add and remove objects according to the problem
void add(int i){
}

void remove(int i){
}

int get_ans(){
}


void mo(vector<Query> &queries){
  sort(queries.begin(), queries.end());
  vector<ll> ans(queries.size());

  int cur_l = queries[0].l, cur_r = queries[0].l - 1;
  for (Query q: queries){
    // Make cur_l = q.l and cur_r = q.r
    while (cur_l < q.l) remove(cur_l++);
    while (cur_l > q.l) add(--cur_l);

    while(cur_r > q.r) remove(cur_r--);
    while (cur_r < q.r) add(++cur_r);

    ans[q.ind] = get_ans();
  }

  // Print answer
  for (auto x: ans) cout << x << "\n";
}
```

1.7 policy$_{tree}$

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

#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp> // Including
    tree_order_statistics_node_update
using namespace __gnu_pbds;

/*
Set with search by order of the comparator function.

O(log n) operations:
 - find(key)
 - erase(*pointer to element)
 - find_by_order(k) -> Find kth element ordered by comp function
 - order_of_key(key) -> Gets the position in which the key element would
     be in the set if inserted
*/
typedef tree<
int,
null_type,
less<int>,
rb_tree_tag,
tree_order_statistics_node_update>
ordered_set;

/*
Custom comparator, and other types.

Multiset could be implemented with pairs
*/

struct comp{

  bool operator() (const pair<long long, string> &p1, const pair<long
      long, string> &p2) {
    if (p1.first != p2.first) return p1.first > p2.first;
    return p1.second < p2.second;
  }

};

typedef tree<
pair<long long, string>,
null_type,
comp,
rb_tree_tag,
tree_order_statistics_node_update>
custom_ordered_set;

/*
Examples
*/

int main(){
  ordered_set X;
  X.insert(1);
  X.insert(2);
  X.insert(4);
  X.insert(8);
  X.insert(16);

  cout<<*X.find_by_order(1)<<endl; // 2
  cout<<*X.find_by_order(2)<<endl; // 4
  cout<<*X.find_by_order(4)<<endl; // 16
  cout<<(end(X)==X.find_by_order(6))<<endl; // true

  cout<<X.order_of_key(-5)<<endl; // 0
  cout<<X.order_of_key(1)<<endl; // 0
  cout<<X.order_of_key(3)<<endl; // 2
  cout<<X.order_of_key(4)<<endl; // 2
  cout<<X.order_of_key(400)<<endl; // 5


  custom_ordered_set t;
  t.insert({1, "ab"});

  auto p = *t.find({1, "ab"});
  cout << p.first << " " << p.second << endl; // 1 ab

  t.erase({1, "ab"});

  p = *t.find({1, "ab"}); // 0
  cout << p.first << " " << p.second << endl;
}
```

1.8 segment$_{t}ree$

```
// Segment tree nodes indexed in 1
// Array is indexed in 0
// Query: [l, r]
// Update: [l, r]
template<typename T>
struct segtree{
  vector<T> t;
  const T NEUTRAL = 0;

  // Size of array
  int N = 0;
  segtree(int n){
    N = n;
    t.assign(4 * N, NEUTRAL);
  }
  segtree(vector<T> &a){
    N = a.size();
    t.assign(4 * N, NEUTRAL);

    build(1, 0, N - 1, a);
  }

  T combine(T a, T b){
    // combine two elements into one
    return a + b;
  }

  void build(int v, int l, int r, vector<T> &a){
    if (l == r){
      t[v] = a[l];
    }
    else{
      int m = (l + r) /2 ;
      build(2 * v, l, m, a);
      build(2 * v + 1, m + 1, r, a);
      t[v] = combine(t[2 * v], t[2 * v + 1]);
    }
  }

  T query(int l, int r){
    return query(1, 0, N - 1, l, r);
  }

  T query(int v, int tl, int tr, int l, int r){
    if (l > r){
      return NEUTRAL;
    }
    if (tl == l && tr == r){
      return t[v];
    }

    int tm = (tl + tr) / 2;
    return combine(
      query(2 * v, tl, tm, l, min(tm, r)),
      query(2 * v + 1, tm + 1, tr, max(l, tm + 1), r)
    );
  }

  void update(int pos, int val){
    update(1, 0, N - 1, pos, val);
  }

  void update(int v, int tl, int tr, int pos, int val){
    if (tl == tr){
      t[v] = val;
      return;
    }
    int tm = (tl + tr) / 2;
    if (tl <= pos && pos <= tm){
      update(2 * v, tl, tm, pos, val);
    }
    else{
      update(2 * v + 1, tm + 1, tr, pos, val);
    }
    t[v] = combine(t[2 * v], t[2 * v + 1]);
  }

};
```

2 Geometry

2.1 circle

```
struct circle{
  double x, y, r;
  circle(double xx, double yy, double rr){
    x = xx;
```

```
    y = yy;
    r = rr;
  }

  bool in(double i, double j){
    // distance {i, j} - > {x, y}
    return r * r >= (x - i) * (x - i) + (y - j) * (y - j);
  }
};
```

2.2 point

```
typedef double T;
struct point{
  T x, y;

  point(){
    x = y = 0;
  }

  point(T xx, T yy){
    x = xx;
    y = yy;
  }

  point operator + (point p){
    return {x + p.x, y + p.y};
  }

  point operator * (T s){
    return {s * x, s * y};
  }

  bool operator == (point p){
    return x == p.x && y== p.y;
  }

  bool operator != (point p){
    return !(*this == p);
  }

  T dot(point p){
    return p.x * x + p.y *y;
  }
  /*
  Returns true if the line between the two points
  is not horizontal.
  */
  bool slope(point p, double &s){
    if (p.x == x){
      s = 0;
      return 0;
    }
    else{
      s = ( 1.0 * (y - p.y)) / ( 1.0 * (x - p.x));
      return 1;
    }
  }
};
```

3 Graphs

3.1 Directed Graphs

3.1.1 bellman ford

```
// This algorithm returns a list with minimum distances from start to all
    nodes

// Time complexity: O(|V|*|E|)
// Space complexiy: O(|V|)
// V = vertices, E = edges

// Tested: https://cses.fi/problemset/task/1673/
// https://cses.fi/paste/92b468920d420ca8a63b1e/

struct Edge {
  int from, to;
  int weight;
};

// Default graph: 1-indexed directed weighted graph
// Default params:
// edges: List of directed edges of the graph
// gn: number of vertices in the graph
```

```cpp
// start: Node from wich the distances begin
/* &reach_neg_cycle: you can pass a variable to save if the graph have at
    least one negative
  cycle reachable from start */
vector<ll> bellman_ford(vector<Edge> &edges, int gn, int start, bool
    &neg_cycle) {
  const ll inf = numeric_limits<ll>::max();
  assert(0 < start && start <= gn);

  vector<ll> dist(gn + 1, inf);
  dist[start] = 0;

  // if there is not any negative cycle, it's enough iterate until gn - 1
  for (int i = 0; i < gn; ++i) {
    bool improve = false;
    for (Edge e: edges) {
      if (dist[e.from] == inf) continue;
      if (dist[e.from] + e.weight >= dist[e.to]) continue; // If it
          doesn't improve
      dist[e.to] = dist[e.from] + e.weight;
      improve = true;
    }
    neg_cycle = improve;
    if (!improve) break;
  }
  return dist;
  // returns numeric_limits<ll>::max() if there's no path
}
```

3.1.2 find strongly connected components

```cpp
// let CNT be the number of diferent strongly connected components

// This algorithm assigns each SCC an identifier, these identifiers are
// numbered from 1 to CNT, and finally returns a list of size |V| + 1
    with the
// information of wich SCC each node belongs to.
// You can also pass a variable in &cnt to save CNT (number of SCCs)

// Time complexity: O(|V| + |E|)
// Space complexity O(|V|)
// V = vertices, E = edges
// Tested: https://codeforces.com/contest/1900/submission/285409426

// Find strongly connected components
// Default: 1-indexed Directed Unweighted graph not necessarily connected
vector<int> find_SCC(vector<vector<int>> &g, int &cnt) {
  const int inf = 1e9;
  const int gn = (int)g.size() - 1;

  vector<bool> in_stack(gn + 1, false);
  vector<bool> vis(gn + 1, false);
  vector<int> who_comp(gn + 1, -1);
  vector<int> tin(gn + 1);
  stack<int> st;
  int timer = 0, tag = 1;

  function<int(int)> dfs = [&](int node) -> int {
    vis[node] = true;
    tin[node] = ++timer;
    in_stack[node] = true;
    st.push(node);
    int low = inf;
    for (int adj: g[node]) {
      if (in_stack[adj]) low = min(low, tin[adj]);
      if (vis[adj]) continue;
      low = min(low, dfs(adj));
    }
    if (low == tin[node] || low == inf) {
      while (who_comp[node] == -1) {
        who_comp[st.top()] = tag;
        in_stack[st.top()] = false;
        st.pop();
      }
      ++tag;
      return inf;
    }
    return low;
  };

  for (int node = 1; node <= gn; ++node) {
    if (vis[node]) continue;
    dfs(node);
  }
  cnt = tag - 1;
  return who_comp;
}
```

3.2 General Graphs

3.2.1 dijkstra (distance list)

```cpp
// This algorithm returns a list with minimum distances from start to all
    nodes

// Time complexity: O(|E| log |V|)
// Space complexity: O(|V| + |E|)
// E = edges, V = vertices

// Requires: inf
// Edges should be represented as: pair<node, dist>
template<typename DT>
vector<ll> dijkstra(vector<vector<pair<int, DT>>> &g, int start) {
  const int gn = g.size();
  assert(0 <= start && start < gn);

  vector<ll> tmp(gn, inf);
  vector<ll> dist(gn, inf);
  priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<pair<ll,
      int>>> pq;

  tmp[start] = 0;
  pq.push({ tmp[start], start });

  while (!pq.empty()) {
    int node = pq.top().second;
    pq.pop();

    if (dist[node] < inf) continue;
    dist[node] = tmp[node];

    for (auto [adj, w]: g[node]) {
      if (dist[adj] < inf) continue;
      if (tmp[adj] <= dist[node] + w) continue; // If it doesn't improve
      tmp[adj] = dist[node] + w;
      pq.push({ tmp[adj], adj });
    }
  }
  return dist;
  // returns inf if there's no path
}
```

3.2.2 hamiltonian-path-dp

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

/*
Hamiltonian path:

Let G be a graph with n vertices.
A hamiltonian path is a simple path (No repeating vertices) with length n.

This is an NP complete problem.
Complexity: O(2^n * n^2)

Idea: Let a single path of G ending at v. To extend this single path
we don't care about the order of the previous vertices, thus
we can use a subset, and the ending to v to solve the problem.

In: Adjacency matrix, number of nodes
Out: Exists hamiltonian path

Tested: https://codeforces.com/contest/1950/submission/255328975
*/

bool is_hamiltonian(vector<vector<int>> &adj, int n){
  // Ending node, subset
  bool dp[n][1 << n];
  memset(dp, 0, sizeof dp);

  for (int i = 0; i < n; i++){
    dp[i][1 << i] = 1;
  }

  for (int mask = 0; mask < (1 << n); mask++){
    for (int u = 0; u < n; u++){
      if (mask & (1 << u)){
        for (int v = 0; v < n; v++){
          // Can create a single path from v to u
          // Iff there is a single path ending at v not including u
          if (mask & (1 << v) && v != u && adj[u][v] && dp[v][mask ^ (1 <<
              u)]){
            dp[u][mask] = 1;
          }
        }
      }
```

```
    }
  }

  for (int i = 0;i < n; i++){
    // Complete subsets for every ending node
    if (dp[i][(1 << n) - 1]){
      return true;
    }
  }

  return false;
}


int main(){
  /*
  4 3
  0 1
  1 2
  1 3
  */
        int n, m; cin >> n >> m;
        vector<vector<int>> adj(n, vector<int>(n));
        for (int i =0 ;i < m; i++){
                int u, v; cin >> u >> v;
    // Undirected
                adj[u][v] = adj[v][u] = 1;
        }

        bool f = is_hamiltonian(adj, n);
        cout << (f ? "YES": "NO") << '\n';
}
```

3.3 Trees

3.3.1 LCA (Binary Lifting)

```
vector<vector<int>> tree;

const int LOG = 20; /// max N < 2^LOG
const int mxN = 2e5; /// max N
int up[mxN + 1][LOG];
// You can declare more vectors with additional info of tree here
// Example: int maximum_edge[mxN + 1][LOG];
int dep[mxN + 1];
int root = 1;

void build_lca(int node, int par) {
  for (int i = 1; i < LOG; ++i) {
    if (up[node][i - 1] == -1) break;
    up[node][i] = up[up[node][i - 1]][i - 1];
    // You must calculate jumps in vectors with additional info here
    // Example: maximum_edge[node][i] = max(maximum_edge[node][i - 1],
        maximum_edge[up[node][i - 1]][i - 1]);
  }
  for (int child: tree[node]) {
    if (child == par) continue;
    dep[child] = dep[node] + 1;
    up[child][0] = node;
    // You must set child data in vectors with additional info here
    // Example: maximum_edge[child][0] = edge_weight;
    build_lca(child, node);
  }
}

void init_lca() {
  memset(up, -1, sizeof(up));
  memset(dep, -1, sizeof(dep));
  // You must set initial content of vectors with additional info here
  // Example: memset(maximum_edge, -1, sizeof(maximum_edge));
  build_lca(root, -1);
}

int lca(int a, int b) {
  if (dep[a] < dep[b]) swap(a, b);
  int dif = dep[a] - dep[b];
  for (int k = 0; k < LOG; ++k) {
    if (dif & (1 << k)) a = up[a][k];
  }
  if (a == b) return a;
  for (int k = LOG - 1; 0 <= k; --k) {
    if (up[a][k] == up[b][k]) continue;
    a = up[a][k];
    b = up[b][k];
  }
  return up[a][0];
}
```

3.3.2 lowest$_c$ommon$_a$ncestor

```
/*
Lowest common ancestor

Gets the lca of two nodes of a rooted tree in O(log n).

dfs(): Traverses the tree and calculates node ancestors in powers
of two.

Calculates other helper arrays:

- par: Parent of nodes
- tin, tout: DFS in and out times
- height: Height of each node starting at 0 (root)

Tested: https://codeforces.com/contest/1328/submission/284525277
*/
int n, l;

vector<int> tin, tout, par, height;
vector<vector<int>> g, up;

int timer = 0;

void dfs(int u, int p){


  par[u] = p;
  height[u] = (u == p ? 0 : height[p] + 1);

  tin[u] = timer++;

  up[u][0] = p;

  for (int i = 1;i <= l; i++){
    up[u][i] = up[up[u][i - 1]][i - 1];
  }

  for (auto v: g[u]){
    if (v == p) continue;

    dfs(v, u);
  }
  tout[u] = timer++;
}

bool is_ancestor(int u, int v){
  return tin[u] <= tin[v] && tout[v] <= tout[u];
}

int lca(int u, int v){
  if (is_ancestor(u, v)) return u;
  if (is_ancestor(v, u)) return v;

  for (int i = l; i >= 0; i--){
    if (!is_ancestor(up[u][i], v)) u = up[u][i];
  }

  return up[u][0];
}

void preprocess(int root){
  // G processed
  timer = 0;
  l = ceil(log2(n));

  tin.resize(n + 1);
  tout.resize(n + 1);
  par.resize(n + 1);
  height.resize(n + 1);
  up.assign(n + 1, vector<int>(l + 1));


  dfs(root, root);
}
```

3.3.3 tree$_d$iameter

```
// The diameter of a tree is the length of the longest path between any
    two nodes.
// In this algorithm the length of a path is considered as
// the number of edges in the path.

// This algorithm returns the value of the diameter

// Time complexity: O(|V|)
```

```
// Space complexity: O(1)
// V = Vertices

// Tested: https://codeforces.com/gym/102694/submission/286468111
    (Problem A)

// Default param: 1-indexed unweighted Tree with at least 1 node
int tree_diameter(vector<vector<int>> &tree) {
  int diameter = 0;
  function<int(int, int)> dfs = [&](int node, int par) -> int {
    int mx = 0, smx = 0;
    for (int child: tree[node]) {
      if (child == par) continue;
      int chdep = dfs(child, node);
      if (chdep > mx) smx = mx, mx = chdep;
      else if (chdep > smx) smx = chdep;
    }
    diameter = max(diameter, smx + mx);
    return mx + 1;
  };
  dfs(1, -1);
  return diameter;
}
```

3.4 Undirected Graphs

3.4.1 find bridges

```
// This algorithm returns a list of bridges in a graph
// The format of these bridges is an edge list where each edge
// is a pair<int, int> and satisfies (edge.first < edge.second)

// Time complexity: O(|V| + |E|)
// Space complexity: O(|V| + |E|)
// V = vertices, E = edges

// Tested: https://codeforces.com/contest/1986/submission/286603550

// Default param: (0/1)-indexed undirected unweighted graph not
    necessarily connected
vector<pair<int, int>> get_bridges(vector<vector<int>> &g) {
  const int inf = 1e9;
  const int gn = (int)g.size();
  vector<pair<int, int>> bridges;
  vector<bool> vis(gn, false);
  vector<int> tim(gn);

  function<int(int, int)> dfs = [&](int node, int par) -> int {
    vis[node] = true;
    int low = inf;
    for (int adj: g[node]) {
      if (adj == par) continue;
      if (vis[adj]) low = min(low, tim[adj]);
      if (vis[adj]) continue;
      tim[adj] = tim[node] + 1;
      int nxt_low = dfs(adj, node);
      if (nxt_low == inf) bridges.emplace_back(min(node, adj), max(node,
          adj));
      low = min(low, nxt_low);
    }
    if (low == tim[node]) return inf;
    return low;
  };
  for (int node = 0; node < gn; ++node) {
    if(vis[node]) continue;
    dfs(node, -1);
  }
  return bridges;
}
```

4 Math

4.1 binary$_e$*xponetiation*

```
/*
Modular binary exponetiation algorithm:
Returns: (a ^ b) % m

Usage:
binpow(2, 3) = 8
binpow(2, mod - 2) * 2 = 1 (If mod prime)

*/
const int mod = 1e9 + 7;
```

```
ll binpow(ll a, ll b){
  if (b == 0) return 1;
  if (b == 1) return a;
  ll x = binpow(a, b / 2);
  return (((x * x) % mod) * (b & 1 ? a : 1ll)) % mod ;
}

/*
Faster iterative implementation
*/
ll binpow(ll a, ll b) {
    a %= mod;
    ll res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return res;
}
```

4.2 linear-diophantine

```
#include <bits/stdc++.h>
using namespace std;
/*
Extended euclidean algorithm:
Calculates gcd of two positive numbers, a and b,
and coefficients of Bezut identity:
ax + by = gcd(a, b)

O(log min(a, b))
*/
int gcd(int a, int b, int& x, int& y) {
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }
  int x1, y1;
  int d = gcd(b, a % b, x1, y1);
  x = y1;
  y = x1 - y1 * (a / b);
  return d;
}

/*
Finds a solution to the linear diophantine equation:
ax + by = c, using extended euclidean algorithm.

Manages a, b < 0. If the gcd of a and b doesn't divide c
there aren't integer solutions. Scales x, and y after running extended
    gcd.
O(log min(a, b))
*/
bool find_any_solution(int a, int b, int c, int &x0, int &y0, int &g) {
  g = gcd(abs(a), abs(b), x0, y0);
  if (c % g) {
    return false;
  }

  x0 *= c / g;
  y0 *= c / g;
  if (a < 0) x0 = -x0;
  if (b < 0) y0 = -y0;
  return true;
}

void shift_solution(int & x, int & y, int a, int b, int cnt) {
  x += cnt * b;
  y -= cnt * a;
}
/*
Counts all solutions of the ax + by = c equation
with constraints for x and y.

Uses the fact that we can modify the linear diophantine values for x and
    y,
and still get a valid equation.

x = x0 + k(b / g)
y = y0 - k(a / g)

O(log min(a, b))
*/
int find_all_solutions(int a, int b, int c, int minx, int maxx, int miny,
    int maxy) {
```

```
  int x, y, g;
  if (!find_any_solution(a, b, c, x, y, g))
    return 0;
  a /= g;
  b /= g;

  int sign_a = a > 0 ? +1 : -1;
  int sign_b = b > 0 ? +1 : -1;

  shift_solution(x, y, a, b, (minx - x) / b);
  if (x < minx)
    shift_solution(x, y, a, b, sign_b);
  if (x > maxx)
    return 0;
  int lx1 = x;

  shift_solution(x, y, a, b, (maxx - x) / b);
  if (x > maxx)
    shift_solution(x, y, a, b, -sign_b);
  int rx1 = x;

  shift_solution(x, y, a, b, -(miny - y) / a);
  if (y < miny)
    shift_solution(x, y, a, b, -sign_a);
  if (y > maxy)
    return 0;
  int lx2 = x;

  shift_solution(x, y, a, b, -(maxy - y) / a);
  if (y > maxy)
    shift_solution(x, y, a, b, sign_a);
  int rx2 = x;

  if (lx2 > rx2)
    swap(lx2, rx2);
  int lx = max(lx1, lx2);
  int rx = min(rx1, rx2);

  if (lx > rx)
    return 0;
  return (rx - lx) / abs(b) + 1;
}

/*
Upper bounds:
Because x = x0 + k(b / g) and y = y0 - k(a / g),
if (x > (c / a)) -> ax + by = c, y = (c - ax) / b
then y < 0, thus we can set:
Maxx: (c / a) + 1, as a valid upper bound
Similarly,
Maxy: (c / b) + 1
*/
int find_non_negative(int a, int b, int c){
  return find_all_solutions(a, b, c, 0, (c / a) + 1, 0, (c / b) + 1);
}
```

4.3 matrix-exponentiation

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

#define ll long long

const int mod = 1e9 + 7;

/*
Tested: https://cses.fi/problemset/result/8944332/
*/

/*
Given A (m * n), and B (n * p) matrices,
compute C = AB, of size (m * p)
where C_(ij) = sum(k -> [0, n))( A_(ik) * B_(kj))

Complexity: O(n^3)

Implemeted with mod 10^9 + 7
*/
vector<vector<ll>> multiply_matrix(vector<vector<ll>>& A,
    vector<vector<ll>>& B){
  // Matrix m * n, n * p
  int m = A.size(), p = B[0].size();
  vector<vector<ll>> C(m, vector<ll>(p));

  int n = A[0].size(), nn = B.size();

  // Can't multiply
  if (n != nn){
```

```
    return C;
  }

  for (int i = 0;i < m; i++){
    for (int j = 0;j < p; j++){
      ll sum = 0;
      for (int k = 0; k < n; k++){
        sum = sum + A[i][k] * B[k][j];
        if (sum < 0){
          ll q = (abs(sum) + mod - 1) / mod;
          sum = (sum + (mod * q)) % mod;
        }
        else{
          sum %= mod;
        }
      }
      C[i][j] = sum;
    }
  }

  return C;
}

/*
Fast matrix exponentiation:

Given A (n * n) matrix, compute A^b (n * n)

Idea: We know A^b = A^(b / 2) * A^(b / 2).
Let X be the identity matrix.



Complexity: O(n^3 * log(b))
*/

vector<vector<ll>> expo_matrix(vector<vector<ll>>& A, ll b){
  int n = A.size();

  vector<vector<ll>> X(n, vector<ll>(n));
  for (int i = 0;i < n; i++) X[i][i] = 1;

  for(int i = (int)log2(b)+1; i >=0; i--){
    X = multiply_matrix(X,X);
    if(b & (1LL << i)) X = multiply_matrix(X,A);
  }

  return X;
}
```

4.4 matrix-template

```
#include <bits/stdc++.h>
using namespace std;
long long mod = 1e9 + 7;

/*
Struct matrix, numerical entries.
Usually used with long long or unsigned long long

M rows, N columns matrix
By default 2 * 2 identity matrix.

Addition of matrix O(m * n)
- A(mxn) + B(mxn) = C(mxn)
Multiplication of matrix O(m * n * p):
- A(mxn) * B(nxp) = C(mxp)

Matrix exponentiation O(n^3 * long(b)): Only squared matrices
- A ^ b
*/
template<typename T> struct matrix{
  int M, N;
  vector<vector<T>> A;
  matrix(){}

  matrix(int m, int n, bool iden = false){
    M = m;
    N = n;
    A.assign(M, vector<T>(N, 0));
    if (iden)
      for (int i =0 ;i < min(N, M); i++) A[i][i] = 1;
  }

  matrix(vector<vector<T>> B){
    M = B.size();
    N = B[0].size();
    A=B;
```

```
}

matrix<T> operator + (matrix<T> B){
  assert("Uncompatible matrix addition" && B.N == N && B.M == M);
  matrix C(M, N);
  for (int i = 0;i < M; i++){
    for (int j = 0;j < N; j++){
      C.A[i][j] = ( ((A[i][j] + B.A[i][j]) % mod) + mod) % mod;
    }
  }
  return C;
}

matrix<T> operator * (matrix<T> B){
  assert("Uncompatible matrix multiplication" && N == B.M);
  matrix C(M, B.N);

  for (int i =0;i < M; i++){
    for (int j = 0;j < B.N; j++){
      T sum = 0;
        for (int k = 0; k < N; k++){
        sum += A[i][k] * B.A[k][j];
        sum = ( (sum % mod) + mod) % mod;
      }
      C.A[i][j] = sum;
    }
  }
  return C;
}

matrix<T> operator ^ (T b){
  matrix<T> C(N, N, true), B(A);

  while (b){
    if (b & 1){
      C = C * B;
    }
    B = B * B;
    b /= 2;
  }

  return C;
}

void print(){
    for (int i = 0; i < M; i++){
      for (int j =0;j < N; j++){
        cout << A[i][j] << ' ';
      }
      cout << '\n';
    }
  }
};
```

4.5 phi

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

/*
Calculates phi or the euler totient function
for a number i. (# coprimes in 1 ... n)

n = mult ( pi ^ ei)
phi(n) = n * mult (1 - pi)

We have the equation:
n (1 - p) = n - (n / p)
Thus we can easily calculate the value of phi

Complexity: O(sqrt(n))
Trial division:
It is enough to iterate through all primes that divide n
in 1 ... sqrt(n).
The algorithm skips composite numbers, thus only operating with primes
    that divide n.
*/

int phi(int n){
  int res = n;
  for (int i = 2; i * i <= n; i++){
    if (n % i){
      while (n % i){
        n /= i;
      }
      res -= res / i;
    }
  }
```

```
  // n prime
  if (n > 1){
    res -= res / n;
  }

  return res;
}

/*
Compute phi from 1 to n
Use the sieve concept to only iterate through prime multiples
and modify the answer for each value
Complexity: O(n ln ln n)
*/

void phi_1_to_n(vector<int> &phi, int n){
  phi.resize(n + 1);
  for (int i = 0;i <= n; i++) phi[i] = i;

  for (int i = 2; i * i <= n; i++){
    for (int j = i; j <= n; j+=i){
      phi[j] -= phi[j] / i;
    }
  }
}
```

4.6 sieve

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

/*
Usage:
Calculate primes in the range 1, ... , n
Complexity: O(n ln ln n)
*/
void run_sieve(vector<bool> &sieve, int n){
  sieve.assign(n + 1, 1);
  sieve[0] = sieve[1] = 0;

  for (long long i = 2; i * i <= n; i++){
    if (sieve[i]){
      for (long long j = i * i; j <= n; j+=i){
        sieve[j] = false;
      }
    }
  }
}

/*
Create primes up to a number N.
Complexity: O(n ln ln n)
*/

void gen_primes(vector<bool> &sieve, vector<int> &primes, int n){
  sieve.assign(n + 1, 1);
  primes.clear();
  sieve[0] = sieve[1] = 0;

  for (long long i = 2; i <= n; i++){
    if (sieve[i]){
      primes.push_back((int)i);
      for (long long j = i * i; j <= n; j+=i){
        sieve[j] = false;
      }
    }
  }

}


/*
Example:
*/
int main(){
  int n = 100;
  vector<bool> sieve;
  run_sieve(sieve, n);
  for (int i =0 ;i <= n; i++){
    if (sieve[i]){
      cout << i << endl;
    }
  }

  int m = 1000000;
  vector<int> primes;
  gen_primes(sieve, primes, m);
  cout << primes.size() << '\n';
```

```
  // for (auto p: primes){
  //   cout << p << '\n';
  // }
}
```

5 Network flows

5.1 Maximum bipartite matching

```
/*
 The goal is to maximize the number of such edges, ensuring that no node
     is matched more than once. The maximum flow in this network
     corresponds to the size of the maximum matching.
/// Complexity: O(|E|*|V|)
Tested: https://vjudge.net/solution/54826125/1o9jH7bQJk9BOsHl6rbq
*/
struct mbm {
  vector<vector<int>> g;
  vector<int> d, match;
  int nil, l, r;
  /// u -> 0 to l, v -> 0 to r
  mbm(int l, int r) : l(l), r(r), nil(l+r), g(l+r),
                      d(1+l+r, INF), match(l+r, l+r) {}
  map<int, int> L, R;
  void add_edge(int a, int b) {//Adds edges to the graph
    if(!L.count(a)) { int t = L.size(); L[a] = t; }
    if(!R.count(b)) { int t = R.size(); R[b] = t; }
    a = L[a];
    b = R[b];
    g[a].push_back(l+b);
    g[l+b].push_back(a);
  }
  bool bfs() {
    queue<int> q;
    for(int u = 0; u < l; u++) {
      if(match[u] == nil) {
        d[u] = 0;
        q.push(u);
      } else d[u] = INF;
    }
    d[nil] = INF;
    while(q.size()) {
      int u = q.front(); q.pop();
      if(u == nil) continue;
      for(auto v : g[u]) {
        if(d[ match[v] ] == INF) {
          d[ match[v] ] = d[u]+1;
          q.push(match[v]);
        }
      }
    }
    return d[nil] != INF;
  }
  bool dfs(int u) {
    if(u == nil) return true;
    for(int v : g[u]) {
      if(d[ match[v] ] == d[u]+1 && dfs(match[v])) {
        match[v] = u; match[u] = v;
        return true;
      }
    }
    d[u] = INF;
    return false;
  }
  int max_matching() {//finds the maximum bipartite matching
    int ans = 0;
    while(bfs()) {
      for(int u = 0; u < l; u++) {
        ans += (match[u] == nil && dfs(u));
      }
    }
    return ans;
  }
};
```

5.2 edmonds karp

```
// This algorithm modify the residual graph and returns the value of
    maximum flow

// Time complexity: O(|V| * (|E| ** 2))
// Space complexity: O(|V|)
// V = vertices, E = edges
```

```
// tested: https://codeforces.com/group/Ohoz9kAFjS/contest/266572
    (Problem B)

ll Edmonds_karp() {
  const int gn = 2 * n;

  auto bfs = [&]() -> int {
    int par[gn + 1];
    memset(par, -1, sizeof(par));
    par[source] = -2;
    queue<int> que;
    que.push(source);

    while (!que.empty()) {
      int node = que.front();
      que.pop();

      if (node == sink) break;

      for (int adj: network[node]) {
        if (par[adj] != -1 || cap[node][adj] == 0) continue;
        par[adj] = node;
        que.push(adj);
      }
    }
    if (par[sink] == -1) return 0;
    int cur = sink;
    ll inc_flow = inf;
    while (cur != source) {
      assert(par[cur] > 0);
      inc_flow = min(inc_flow, cap[par[cur]][cur]);
      cur = par[cur];
    }
    cur = sink;
    while (cur != source) {
      cap[par[cur]][cur] -= inc_flow;
      cap[cur][par[cur]] += inc_flow;
      cur = par[cur];
    }
    return inc_flow;
  };
  ll flow, max_flow = 0;
  while (flow = (ll)bfs()) max_flow += flow;
  return max_flow;
}
```

5.3 ford fulkerson (dfs)

```
// Find value of maximum flow
// Let F be the value of maximum flow
// Time complexity: O(F * (|V| + |E|))
// Space complexity: O(|V|)

vector<vector<int>> network(n + 1);
vector<bool> vis(n + 1);
int cap[mxN + 1][mxN + 1];
memset(cap, 0, sizeof(cap));

int dfs(int node, int min_cap) {
  vis[node] = true;
  if (node == sink) return min_cap;

  for (int adj: network[node]) {
    if (vis[adj]) continue;
    int next = cap[node][adj];
    if (!next) continue;
    int flow = dfs(adj, min(min_cap, next));
    cap[node][adj] -= flow;
    cap[adj][node] += flow;
    if (flow) return flow;
  }
  return 0;
}
// Find maximum flow
int max_flow = 0;
int flow = 0;
while (flow = dfs(source, inf)) {
  max_flow += flow;
  fill(vis.begin(), vis.end(), false);
}
```

6 Notes

7 Strings

7.1 Hashing

```
/*
Classic hash
Another values = 1000234987, 1000567991, 1000111979, 1000777117
Complexity: O(n)
Tested: https://codeforces.com/contest/1968/submission/285218366
*/

const unsigned long long MOD[2] = { 1001864327, 1001265673};//primes
struct hashing {
    vector<ll>values[2] ;
    vector<ll> pot[2];
    ll base=29; //prime
    //creates two hashes in vector 'values' over string s
    hashing (string s){
        int n=s.size();
        values[ 0 ].assign( n , 0 );
        values[ 1 ].assign( n , 0 );
        for( int j = 0; j <=1; ++j){
            for( int i = 0; i < n; ++i){
                if(i == 0) values[j][i] = 1 + s[i] - 'a';
                else{
                    values[ j ][ i ]=( ( ( values[j][ i - 1 ] * base ) ) +
                        ( 1 + s[i] - 'a') ) % MOD[j];
                }
            }
        }

        int p0 = 1, p1 = 1;
        for(int i = 0; i < n; ++i){
            pot[ 0 ].push_back( p0 );
            pot[ 1 ].push_back( p1 );
            p0=(p0 * base)%MOD[0];
            p1=(p1 * base)%MOD[1];
        }
    }

    //query type [ l , r ]
    pair<ll,ll> query(int l, int r){
        if( l == 0 )return {values[ 0 ][ r ], values[ 1 ][ r ]};
        else{
            ll x = ((( values[ 0 ][ r ])-(( values[ 0 ][ l - 1 ] * pot[ 0
                ][ r - l + 1] )% MOD[ 0 ])) + MOD[ 0 ])% MOD[ 0 ];
            ll y = ((( values[ 1 ][ r ])-(( values[ 1 ][ l - 1 ] * pot[ 1
                ][ r - l + 1] )% MOD[ 1 ])) + MOD[ 1 ])% MOD[ 1 ];
            return { x , y };
        }
    }
};
```

7.2 KMP

```
// From cp-algorithms
/*
The prefix function for this string is defined as an array pi of length
n, where
pi_i is the length of the longest proper prefix of the substring
s[0...i]   which is also a suffix of this substring.
*/
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

7.3 Suffix Array

```
/*
A suffix array will contain integers that represent the starting indexes
of the all the suffixes of a given string, after the aforementioned
    suffixes are sorted.

(IN DESCOMUNAL)
*/
```

7.4 Z function

```
/*
From cp-algorithms

The Z-function for this string is an array of length
n where the
i-th element is equal to the greatest number of characters starting from
    the position
i that coincide with the first characters of s.
In other words,z[i] is the length of the longest string that is, at the
    same time, a prefix of
s and a prefix of the suffix of s starting at i.
*/

vector<int> z_function(string s) {
    int n = s.size();
    vector<int> z(n);
    int l = 0, r = 0;
    for(int i = 1; i < n; i++) {
        if(i < r) {
            z[i] = min(r - i, z[i - l]);
        }
        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;
}
```

7.5 manacher

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

const int N = 1e6;
int d1[N]; //d1[i] = max odd palindrome centered on i
int d2[N]; //d2[i] = max even palindrome centered on i and i - 1

//s  aabbaacaabbaa
//d1 1111117111111
//d2 0103010010301

void manacher(string& s){
        int l=0,r=-1,n=s.size();

        for (int i = 0;i < n; i++){
                int k=i > r ? 1 :min(d1[l+r-i], r-i) ;
                while(i+k<n&&i-k>=0&&s[i+k]==s[i-k])k++;
                d1[i]=k--;
                if(i+k>r)l=i-k,r=i+k;
        }
        l=0;r=-1;

        for (int i =0 ;i < n; i++){
                int k=i>r?0:min(d2[l+r-i+1],r-i+1);k++;
                while(i+k<=n&&i-k>=0&&s[i+k-1]==s[i-k])k++;
                d2[i]=--k;
                if(i+k-1>r)l=i-k,r=i+k-1;
        }
}
```

8 Templates

8.1 TEMPLATE *juanDavid*

```
#include<bits/stdc++.h>
#define all(a) a.begin(),a.end()
#define rall(a) a.rbegin(), a.rend()
#define forn(u,n) for(int u=0;u<n;++u)
#define forns(u,i,n) for(int u=i;u<n;++u)
#define todo0(a) memset(a,0,sizeof a)
#define todom1(a) memset(a,-1,sizeof a)
#define reverso int, vector<int>, greater<int>
#define sz(a) ((int)a.size())
#define N 100005
#define pb push_back
#define snd second
#define fst first

using namespace std;
```

```
typedef long long ll;

const unsigned long long MOD = 1000567999;

int main(){
    #ifdef LOCAL
    freopen("entra.in","r",stdin);
    #endif
    ios_base::sync_with_stdio(0);
    cin.tie(NULL);
}
```

8.2 template

```
// Made by Daniel Diaz (@Danidiaztech)
#include <bits/stdc++.h>
using namespace std;

#define ll long long

const int mod = 1e9 + 7;
const string yes = "YES", no = "NO";

void solve(){
  int n;
  cin >> n;
}

int main() {
  cin.tie(0); cout.tie(0); ios_base::sync_with_stdio(0);

  #if LOCAL
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
  #endif

  int tc = 1;
  // cin >> tc;

  for (int t = 1; t <= tc; t++){
    solve();
  }
}
```

9 Utilities

9.1 all$_{p}$ossible$_{s}$ubset$_{s}$ums

```
// This algorithms receives a list of integers and returns a bitset
// where if X is a possible subset sum, bitset[X] = 1 and
// bitset[X] = 0 otherwise

// Let S be the sum of the list
// Let N be the size of the list

// I'm not 100% sure about these complexities but you can use them as a
    reference
// Time complexity: O(S * S / 32);
// Space complexity: O(S + S)

// Tested: https://codeforces.com/contest/1856/submission/286483210

template<int BITSET_LEN>
bitset<BITSET_LEN> all_possible_subset_sums(vector<int> &arr) {
  bitset<BITSET_LEN> mask(1);

  map<int, int> frq;
  for (int e: arr) ++frq[e];

  for (auto [e, f]: frq) {
    for (int shift = 1; shift <= f; shift <<= 1) {
      mask |= (mask << (shift * e));
      f -= shift;
    }
    mask |= (mask << (f * e));
  }
  return mask;
}
```

9.2 big$_{p}$rimes

```
const long long bigprimes[] = {
  // Order 1e9
  1001864327,
  1001265673,
  1000234987,
```

```
  1000567991,
  1000111979,
  1000777117,
  1000000007,
  1000000009,
  // Between 1e15, 1e18
  163578793715417,
  894237155356163,
  7962031747340791,
  6390096652364827,
  93871023492391309,
  52343595705624173
};
```

9.3 constants

```
# define PI          3.14159265358979323846
```

9.4 hash$_t able$

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
gp_hash_table<int, int> mp;

const int RANDOM =
    chrono::high_resolution_clock::now().time_since_epoch().count();
struct chash {
   int operator()(int x) const { return x ^ RANDOM; }
};
gp_hash_table<int, int, chash> table;
```

9.5 int128

```
typedef __int128 ll;
// typedef __uint128 ll;

ostream &operator<<(ostream &os, const ll &p) {
    vector<int> v;
    ll x = p;
    if (x == 0) v.push_back(0);
    while(x){
      v.push_back(x % 10);
      x /= 10;
    }
    reverse(v.begin(), v.end());
    for(int i : v){
       os << i;
    }
    return os;
}
istream &operator>>(istream &is, ll &p) {
    string s;
    is >> s;
    p = 0;
    for(char i : s){
      p = p * 10 + (i - '0');
    }
    return is;
}
```

9.6 movements$_2 d$

```
/*
Used for movements in a 2d matrix. Useful for Graph problems on graphs.
Tested: https://codeforces.com/gym/518941/submission/257482117
*/
// L R U D LU LD RU RD
int dr[] = {0, 0, -1, 1, -1, 1, -1, 1};
int dc[] = {-1, 1, 0, 0, -1, -1, 1, 1};

// N rows, M columns
int n, m;
bool valid(int i, int j){
  return i >= 0 && j >= 0 && i < n && j < m;
}
```

9.7 pragma$_o ptimizations$

```
/*
Use with caution, may cause precision errors
```

```
with floats
*/
#pragma GCC optimize ("O3")
#pragma GCC target ("sse4")
#pragma GCC target ("avx,tune=native")


// Other pragmas
#pragma GCC optimize("Ofast")
#pragma GCC
    target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,avx,tune=native")
```

9.8 $\text{random}_{n}umber_{g}enerator$

```
/*
Function that generates a pseudo-random unsigned 64 bit number

Usage:
unsigned long long random_number = rng();
*/

std::mt19937_64
    rng(std::chrono::steady_clock::now().time_since_epoch().count());

/*
Static random number given an unsigned long long x
Complexity: O(1)
*/

static uint64_t splitmix64(uint64_t x) {
       // http://xorshift.di.unimi.it/splitmix64.c
       x += 0x9e3779b97f4a7c15;
       x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
       x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
       return x ^ (x >> 31);
}
```