//Mathematics portion

```
ll expo(ll base, int exp) {
	ll res = 1LL;
	while (exp) {
		if (1 & exp) res *= base;
		base *= base;
		exp >>= 1;
	}
	return res;
}
int phi(int n) { //a^phi(m) = 1 (mod m) when a and m are relatively prime, Euler totient
	int result = n;
	for (int i = 2; (i * i) <= n; ++i)
		if (0 == (n % i)) {
			while (0 == (n % i)) n /= i;
			result -= (result / i);
		}
	if (1 < n) result -= (result / n);
	return result;
}
ll gcd(ll a, ll b) {
	while (b) {
		a %= b;
		(a ^= b), (b ^= a), (a ^= b);
	}
	return a;
}
int lcm(int a, int b) {
	return a * (b / gcd(a, b));
}
bitset<10000000010> p;
vector<ll> P;
ll Max = 1LL;
void sOE(ll upperBound) { // sOE = sieveOfEratosthenes
	if (10000000010LL <= upperBound) upperBound = 10000000009LL;
	if (upperBound <= Max) return ;
	if (1LL == Max) p[0] = p[1] = 1;
	for (ll i = 1LL + Max; i <= upperBound; ++i)
		if (0 == p[i]) {
			for (ll j = i * i; j <= upperBound; j += i) p[j] = 1;
			P.push_back(i);
		}
	Max = upperBound;
}
```

```
bool isPrime(ll num) {
	if (num <= Max) return p[num];
	sOE(num);
	for (int i = 0; i < (int)P.size(); ++i)
		if (0LL == num % P[i]) return false;
	return true;
}
ll ex_gcd(ll a, ll b, ll & x, ll & y) {
	ll xx = y = 0LL;
	ll yy = x = 1LL;
	while (b) {
		int q = a / b, 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;
}
//Binomial and Catalan
ll calculateNCK(ll n, ll k) {
    if (n < 0LL || k < 0LL)
        return -1LL;
    else if (n < k) return 0LL;
    ll ans = 1LL, r = 0LL;
    k = ((k << 1) <= n) ? k : (n - k);
    while (r < k) {
        ans *= (n - r);
        ++r;
        ans /= r;
    }
    return ans;
}
vector<int> primeFactors(ll N) {
  vector<int> factors;
  ll PF_idx = 0LL, PF = P[PF_idx];
  while (N != 1LL && (PF * PF <= N)) {
    while (N % PF == 0LL) { N /= PF; factors.push_back(PF); }
    PF = P[++PF_idx];
  }
  if (N != 1LL) factors.push_back(N);
  return factors;
}
ll numPF(ll N) {		//number of prime factors
	sOE(N);
	ll PF_idx = 0LL, PF = P[PF_idx], ans = 0LL;
```

```
	while (N != 1LL && (PF * PF <= N)) {
	while (N % PF == 0LL) { N /= PF; ++ans; }
	PF = P[++PF_idx];
	}
	if (N != 1LL) ++ans;
	return ans;
}
ll numDiv(ll N) {
	sOE(N);
	ll PF_idx = 0LL, PF = P[PF_idx], ans = 1LL;		// start from ans = 1
	while (N != 1 && (PF * PF <= N)) {
	ll power = 0LL;					// count the power
	while (N % PF == 0LL) { N /= PF; ++power; }
	ans *= (1LL + power);				// according to the formula
	PF = P[++PF_idx];
	}
	if (N != 1LL) ans *= 2LL;		// (last factor has pow = 1, we add 1 to it)
	return ans;
}
ll sumDiv(ll N) {
	sOE(N);
	ll PF_idx = 0, PF = P[PF_idx], ans = 1; // start from ans = 1
	while (PF * PF <= N) {
		ll power = 0;
		while (N % PF == 0) { N /= PF; power++; }
		ans *= ((ll)pow((double)PF, power + 1.0) - 1LL) / (PF - 1LL);
		PF = P[++PF_idx];
	}
	if (N != 1LL) ans *= ((ll)pow((double)N, 2.0) - 1LL) / (N - 1LL); // last
	return ans;
}
ll mod(ll a, ll b) {
	return ((a%b) + b) % b;
}
// finds all solutions to ax = b (mod n)
vector<ll> modular_linear_equation_solver(ll a, ll b, ll n) {
	ll x, y;
	vector<ll> ret;
	ll g = ex_gcd(a, n, x, y);
	if (!(b%g)) {
		x = mod(x*(b / g), n);
		for (int i = 0; i < g; i++)
			ret.push_back(mod(x + i*(n / g), n));
	}
```

```
	return ret;
}
ll mod_inverse(ll a, ll n) {
	ll x, y;
	ll g = ex_gcd(a, n, x, y);
	if (g > 1LL) return -1LL;
	return mod(x, n);
}
// Chinese remainder theorem (special case): find z such that
// z % m1 = r1, z % m2 = r2. Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M). On failure, M = -1.
pair<ll,ll> chinese_remainder_theorem(ll m1, ll r1, ll m2, ll r2) {
	ll s, t;
	ll g = ex_gcd(m1, m2, s, t);
	if (r1%g != r2%g) return make_pair(0, -1);
	return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);
}
// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(ll a, ll b, ll c, ll &x, ll &y) {
	if (!a && !b) {
		if (c) return false;
		x = 0; y = 0;
		return true;
	}
	if (!a) {
		if (c % b) return false;
		x = 0; y = c / b;
		return true;
	}
	if (!b) {
		if (c % a) return false;
		x = c / a; y = 0;
		return true;
	}
	int g = gcd(a, b);
	if (c % g) return false;
	x = c / g * mod_inverse(a / g, b / g);
	y = (c - a*x) / b;
	return true;
}
int f(int x) {
	return x;
}
```

```
pair<int,int> floydCycleFinding(int x0) { // find mu and lambda such that xv(mu) = xv(mu + lambda)
        int tortoise = f(x0), hare = f(tortoise);
        while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
        int mu = 0; hare = x0;
        while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); ++mu;}
        int lambda = 1; hare = f(tortoise);
        while (tortoise != hare) { hare = f(hare); ++lambda; }
        return pair<int,int>(mu, lambda);
}
```

//Segment with lazy

```
//lazy tree
void updateRange(int node, int start, int end, int l, int r, int val)
{
    if(lazy[node] != 0)
    {
        // This node needs to be updated
        tree[node] += (end - start + 1) * lazy[node];    // Update it
        if(start != end)
        {
            lazy[node*2] += lazy[node];                  // Mark child as lazy
            lazy[node*2+1] += lazy[node];                // Mark child as lazy
        }
        lazy[node] = 0;                                  // Reset it
    }
    if(start > end or start > r or end < l)              // Current segment is not within range [l, r]
        return;
    if(start >= l and end <= r)
    {
        // Segment is fully within range
        tree[node] += (end - start + 1) * val;
        if(start != end)
        {
            // Not leaf node
            lazy[node*2] += val;
            lazy[node*2+1] += val;
        }
        return;
    }
    int mid = (start + end) / 2;
    updateRange(node*2, start, mid, l, r, val);        // Updating left child
    updateRange(node*2 + 1, mid + 1, end, l, r, val);   // Updating right child
    tree[node] = tree[node*2] + tree[node*2+1];        // Updating root with max value
```

```
}

int queryRange(int node, int start, int end, int l, int r)
{
    if(start > end or start > r or end < l)
        return 0;         // Out of range
    if(lazy[node] != 0)
    {
        // This node needs to be updated
        tree[node] += (end - start + 1) * lazy[node];            // Update it
        if(start != end)
        {
            lazy[node*2] += lazy[node];         // Mark child as lazy
            lazy[node*2+1] += lazy[node];    // Mark child as lazy
        }
        lazy[node] = 0;                 // Reset it
    }
    if(start >= l and end <= r)             // Current segment is totally within range [l, r]
        return tree[node];
    int mid = (start + end) / 2;
    int p1 = queryRange(node*2, start, mid, l, r);         // Query left child
    int p2 = queryRange(node*2 + 1, mid + 1, end, l, r); // Query right child
    return (p1 + p2);
}


/*   NORMAL SEGMENT TREE */
void build(int node, int start, int end)
{
    if(start == end)
    {
        // Leaf node will have a single element
        tree[node] = A[start];
    }
    else
    {
        int mid = (start + end) / 2;
        // Recurse on the left child
        build(2*node, start, mid);
        // Recurse on the right child
        build(2*node+1, mid+1, end);
        // Internal node will have the sum of both of its children
        tree[node] = tree[2*node] + tree[2*node+1];
    }
```

```
}
void update(int node, int start, int end, int idx, int val)
{
    if(start == end)
    {
        // Leaf node
        A[idx] += val;
        tree[node] += val;
    }
    else
    {
        int mid = (start + end) / 2;
        if(start <= idx and idx <= mid)
        {
            // If idx is in the left child, recurse on the left child
            update(2*node, start, mid, idx, val);
        }
        else
        {
            // if idx is in the right child, recurse on the right child
            update(2*node+1, mid+1, end, idx, val);
        }
        // Internal node will have the sum of both of its children
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
int query(int node, int start, int end, int l, int r)
{
    if(r < start or end < l)
    {
        // range represented by a node is completely outside the given range
        return 0;
    }
    if(l <= start and end <= r)
    {
        // range represented by a node is completely inside the given range
        return tree[node];
    }
    // range represented by a node is partially inside and partially outside the given range
    int mid = (start + end) / 2;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r);
    return (p1 + p2);
}
```

//Fast Dijkstra's algorithm

```
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)
#include <queue>
#include <cstdio>
using namespace std;
const int INF = 2000000000;
typedef pair<int, int> PII;
int main() {
int N, s, t;
scanf("%d%d%d", &N, &s, &t);
vector<vector<PII> > edges(N);
for (int i = 0; i < N; i++) {
int M;
scanf("%d", &M);
for (int j = 0; j < M; j++) {
int vertex, dist;
scanf("%d%d", &vertex, &dist);
edges[i].push_back(make_pair(dist, vertex)); // note order of arguments here
}
}
// use priority queue in which top element has the "smallest" priority
priority_queue<PII, vector<PII>, greater<PII> > Q;
vector<int> dist(N, INF), dad(N, -1);
Q.push(make_pair(0, s));
dist[s] = 0;
while (!Q.empty()) {
PII p = Q.top();
Q.pop();
int here = p.second;
if (here == t) break;
if (dist[here] != p.first) continue;
for (vector<PII>::iterator it = edges[here].begin(); it != edges[here].end(); it++) {
if (dist[here] + it->first < dist[it->second]) {
dist[it->second] = dist[here] + it->first;
dad[it->second] = here;
Q.push(make_pair(dist[it->second], it->second));
}
}
}
printf("%d\n", dist[t]);
```

```
if (dist[t] < INF)
for (int i = t; i != -1; i = dad[i])
printf("%d%c", i, (i == s ? '\n' : ' '));
return 0;
}
/*
Sample input:
5 0 4
2 1 2 3 1
2 2 4 4 5
3 1 4 3 3 4 1
2 0 1 2 3
2 1 5 2 1
Expected:
5
4 2 3 0
*/----------------
```

//Floyd warshall

```
#include <algorithm>
#include <cstdio>
using namespace std;
#define INF 1000000000
int main() {
  int V, E, u, v, w, AdjMatrix[200][200];
  /*
  // Graph in Figure 4.30
  5 9
  0 1 2
  0 2 1
  0 4 3
  1 3 4
  2 1 1
  2 4 1
  3 0 1
  3 2 3
  3 4 5
  */
  freopen("in_07.txt", "r", stdin);
  scanf("%d %d", &V, &E);
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++)
      AdjMatrix[i][j] = INF;
    AdjMatrix[i][i] = 0;
  }
```

```
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjMatrix[u][v] = w; // directed graph
  }
  for (int k = 0; k < V; k++) // common error: remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMatrix[i][j] = min(AdjMatrix[i][j], AdjMatrix[i][k] + AdjMatrix[k][j]);
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      printf("APSP(%d, %d) = %d\n", i, j, AdjMatrix[i][j]);

  return 0;
}-----------------------
//bellman-ford
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000
int main() {
  int V, E, s, a, b, w;
  vector<vii> AdjList;
  /*
  // Graph in Figure 4.18, has negative weight, but no negative cycle
  5 5 0
  0 1 1
  0 2 10
  1 3 2
  2 3 -10
  3 4 3
  // Graph in Figure 4.19, negative cycle exists
  3 3 0
  0 1 1000
  1 2 15
  2 1 -42
  */
  freopen("in_06.txt", "r", stdin);
  scanf("%d %d %d", &V, &E, &s);
  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList
```

```
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &a, &b, &w);
    AdjList[a].push_back(ii(b, w));
  }
  // Bellman Ford routine
  vi dist(V, INF); dist[s] = 0;
  for (int i = 0; i < V - 1; i++)  // relax all E edges V-1 times, overall O(VE)
    for (int u = 0; u < V; u++)                        // these two loops = O(E)
      for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];        // we can record SP spanning here if needed
        dist[v.first] = min(dist[v.first], dist[u] + v.second);         // relax
      }
  bool hasNegativeCycle = false;
  for (int u = 0; u < V; u++)                          // one more pass to check
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] > dist[u] + v.second)                 // should be false
        hasNegativeCycle = true;     // but if true, then negative cycle exists!
    }
  printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");
  if (!hasNegativeCycle)
    for (int i = 0; i < V; i++)
      printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);
  return 0;
}--------------
```

//BFS

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;
typedef pair<int, int> ii;      // In this chapter, we will frequently use these
typedef vector<ii> vii;      // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but shortcuts are useful in competitive programming
int V, E, a, b, s;
vector<vii> AdjList;
vi p;                                // addition: the predecessor/parent vector
void printPath(int u) {    // simple function to extract information from `vi p'
  if (u == s) { printf("%d", u); return; }
  printPath(p[u]);   // recursive call: to make the output format: s -> ... -> t
  printf(" %d", u); }
int main() {
  /*
  // Graph in Figure 4.3, format: list of unweighted edges
```

```
  // This example shows another form of reading graph input
  13 16
  0 1    1 2    2  3   0  4   1  5   2  6    3  7   5  6
  4 8    8 9    5 10   6 11   7 12   9 10   10 11  11 12
  */
  freopen("in_04.txt", "r", stdin);
  scanf("%d %d", &V, &E);
  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d", &a, &b);
    AdjList[a].push_back(ii(b, 0));
    AdjList[b].push_back(ii(a, 0));
  }
  // as an example, we start from this source, see Figure 4.3
  s = 5;
  // BFS routine
  // inside int main() -- we do not use recursion, thus we do not need to create separate
function!
  vi dist(V, 1000000000); dist[s] = 0;      // distance to source is 0 (default)
  queue<int> q; q.push(s);                                // start from source
  p.assign(V, -1); // to store parent information (p must be a global variable!)
  int layer = -1;                              // for our output printing purpose
  bool isBipartite = true;       // addition of one boolean flag, initially true
  while (!q.empty()) {
    int u = q.front(); q.pop();                        // queue: layer by layer!
    if (dist[u] != layer) printf("\nLayer %d: ", dist[u]);
    layer = dist[u];
    printf("visit %d, ", u);
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                          // for each neighbors of u
      if (dist[v.first] == 1000000000) {
        dist[v.first] = dist[u] + 1;                  // v unvisited + reachable
        p[v.first] = u;          // addition: the parent of vertex v->first is u
        q.push(v.first);                              // enqueue v for next step
      }
      else if ((dist[v.first] % 2) == (dist[u] % 2))             // same parity
        isBipartite = false;
  } }
  printf("\nShortest path: ");
  printPath(7), printf("\n");
  printf("isBipartite? %d\n", isBipartite);
  return 0;
}--------------------
```

//kruskal-prim

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
// Union-Find Disjoint Sets Library written in OOP manner, using both path compression and
union by rank heuristics
class UnionFind {                                        // OOP style
private:
  vi p, rank, setSize;                       // remember: vi is vector<int>
  int numSets;
public:
  UnionFind(int N) {
    setSize.assign(N, 1); numSets = N; rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) { numSets--;
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }
    else                   { p[x] = y; setSize[y] += setSize[x];
                           if (rank[x] == rank[y]) rank[y]++; } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[findSet(i)]; }
};
vector<vii> AdjList;
vi taken;                                  // global boolean flag to avoid cycle
priority_queue<ii> pq;            // priority queue to help choose shorter edges
void process(int vtx) {    // so, we use -ve sign to reverse the sort order
  taken[vtx] = 1;
  for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
    ii v = AdjList[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
} }                                // sort by (inc) weight then by (inc) id
int main() {
  int V, E, u, v, w;
  /*
  // Graph in Figure 4.10 left, format: list of weighted edges
  // This example shows another form of reading graph input
```

```
  5 7
  0 1 4
  0 2 4
  0 3 6
  0 4 6
  1 2 2
  2 3 8
  3 4 9
  */
  freopen("in_03.txt", "r", stdin);
  scanf("%d %d", &V, &E);
  // Kruskal's algorithm merged with Prim's algorithm
  AdjList.assign(V, vii());
  vector< pair<int, ii> > EdgeList;   // (weight, two vertices) of the edge
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);            // read the triple: (u, v, w)
    EdgeList.push_back(make_pair(w, ii(u, v)));               // (w, u, v)
    AdjList[u].push_back(ii(v, w));
    AdjList[v].push_back(ii(u, w));
  }
  sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log E)
                    // note: pair object has built-in comparison function
  int mst_cost = 0;
  UnionFind UF(V);                     // all V are disjoint sets initially
  for (int i = 0; i < E; i++) {                      // for each edge, O(E)
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second)) {  // check
      mst_cost += front.first;                // add the weight of e to MST
      UF.unionSet(front.second.first, front.second.second);    // link them
  } }                        // note: the runtime cost of UFDS is very light
  // note: the number of disjoint sets must eventually be 1 for a valid MST
  printf("MST cost = %d (Kruskal's)\n", mst_cost);
// inside int main() --- assume the graph is stored in AdjList, pq is empty
  taken.assign(V, 0);                // no vertex is taken at the beginning
  process(0);   // take vertex 0 and process all edges incident to vertex 0
  mst_cost = 0;
  while (!pq.empty()) {  // repeat until V vertices (E=V-1 edges) are taken
    ii front = pq.top(); pq.pop();
    u = -front.second, w = -front.first;  // negate the id and weight again
    if (!taken[u])                 // we have not connected this vertex yet
      mst_cost += w, process(u); // take u, process all edges incident to u
  }                                         // each edge is in pq only once!
  printf("MST cost = %d (Prim's)\n", mst_cost);
  return 0;
```

```
}
```

//DFS

```
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;
typedef pair<int, int> ii;      // In this chapter, we will frequently use these
typedef vector<ii> vii;      // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but shortcuts are useful in competitive programming
#define DFS_WHITE -1 // normal DFS, do not change this with other values (other than 0), because we usually use memset with conjunction with DFS_WHITE
#define DFS_BLACK 1
vector<vii> AdjList;
void printThis(char* message) {
  printf("==================================\n");
  printf("%s\n", message);
  printf("==================================\n");
}
vi dfs_num;     // this variable has to be global, we cannot put it in recursion
int numCC;
void dfs(int u) {          // DFS for normal usage: as graph traversal algorithm
  printf(" %d", u);                                  // this vertex is visited
  dfs_num[u] = DFS_BLACK;      // important step: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                      // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == DFS_WHITE)         // important check to avoid cycle
      dfs(v.first);      // recursively visits unvisited neighbors v of vertex u
} }
// note: this is not the version on implicit graph
void floodfill(int u, int color) {
  dfs_num[u] = color;                            // not just a generic DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      floodfill(v.first, color);
} }
vi topoSort;             // global vector to store the toposort in reverse order
void dfs2(int u) {    // change function name to differentiate with original dfs
  dfs_num[u] = DFS_BLACK;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      dfs2(v.first);
  }
```

```
  topoSort.push_back(u); }                    // that is, this is the only change
#define DFS_GRAY 2              // one more color for graph edges property check
vi dfs_parent;      // to differentiate real back edge versus bidirectional edge
void graphCheck(int u) {                // DFS for checking graph edge properties
  dfs_num[u] = DFS_GRAY;   // color this as DFS_GRAY (temp) instead of DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {     // Tree Edge, DFS_GRAY to DFS_WHITE
      dfs_parent[v.first] = u;                  // parent of this children is me
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == DFS_GRAY) {             // DFS_GRAY to DFS_GRAY
      if (v.first == dfs_parent[u])          // to differentiate these two cases
        printf(" Bidirectional (%d, %d) - (%d, %d)\n", u, v.first, v.first, u);
      else  // the most frequent application: check if the given graph is cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
    }
    else if (dfs_num[v.first] == DFS_BLACK)             // DFS_GRAY to DFS_BLACK
      printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = DFS_BLACK;     // after recursion, color this as DFS_BLACK (DONE)
}
vi dfs_low;       // additional information for articulation points/bridges/SCCs
vi articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;
void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {                          // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;  // special case, count children of root

      articulationPointAndBridge(v.first);

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

```
vi S, visited;                                   // additional global variables
int numSCC;
void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <= dfs_num[u]
  S.push_back(u);           // stores u in a vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      tarjanSCC(v.first);
    if (visited[v.first])                                // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
  }
  if (dfs_low[u] == dfs_num[u]) {         // if this is a root (start) of an SCC
    printf("SCC %d:", ++numSCC);            // this part is done after recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);

      if (u == v) break;
    }
    printf("\n");
} }
int main() {
  int V, total_neighbors, id, weight;
  /*
  // Use the following input:
  // Graph in Figure 4.1
  9
  1 1 0
  3 0 0 2 0 3 0
  2 1 0 3 0
  3 1 0 2 0 4 0
  1 3 0
  0
  2 7 0 8 0
  1 6 0
  1 6 0
  // Example of directed acyclic graph in Figure 4.4 (for toposort)
  8
  2 1 0 2 0
  2 2 0 3 0
  2 3 0 5 0
  1 4 0
```

```
0
0
0
1 6 0
// Example of directed graph with back edges
3
1 1 0
1 2 0
1 0 0
// Left graph in Figure 4.6/4.7/4.8
6
1 1 0
3 0 0 2 0 4 0
1 1 0
1 4 0
3 1 0 3 0 5 0
1 4 0
// Right graph in Figure 4.6/4.7/4.8
6
1 1 0
5 0 0 2 0 3 0 4 0 5 0
1 1 0
1 1 0
2 1 0 5 0
2 1 0 4 0

// Directed graph in Figure 4.9
8
1 1 0
1 3 0
1 1 0
2 2 0 4 0
1 5 0
1 7 0
1 4 0
1 6 0
*/
freopen("in_01.txt", "r", stdin);

scanf("%d", &V);
AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList
for (int i = 0; i < V; i++) {
  scanf("%d", &total_neighbors);
  for (int j = 0; j < total_neighbors; j++) {
```

```
    scanf("%d %d", &id, &weight);
    AdjList[i].push_back(ii(id, weight));
  }
}
printThis("Standard DFS Demo (the input graph must be UNDIRECTED)");
numCC = 0;
dfs_num.assign(V, DFS_WHITE);    // this sets all vertices' state to DFS_WHITE
for (int i = 0; i < V; i++)                    // for each vertex i in [0..V-1]
  if (dfs_num[i] == DFS_WHITE)           // if that vertex is not visited yet
    printf("Component %d:", ++numCC), dfs(i), printf("\n");   // 3 lines here!
printf("There are %d connected components\n", numCC);

printThis("Flood Fill Demo (the input graph must be UNDIRECTED)");
numCC = 0;
dfs_num.assign(V, DFS_WHITE);
for (int i = 0; i < V; i++)
  if (dfs_num[i] == DFS_WHITE)
    floodfill(i, ++numCC);
for (int i = 0; i < V; i++)
  printf("Vertex %d has color %d\n", i, dfs_num[i]);
// make sure that the given graph is DAG
printThis("Topological Sort (the input graph must be DAG)");
topoSort.clear();
dfs_num.assign(V, DFS_WHITE);
for (int i = 0; i < V; i++)           // this part is the same as finding CCs
  if (dfs_num[i] == DFS_WHITE)
    dfs2(i);
reverse(topoSort.begin(), topoSort.end());                 // reverse topoSort
for (int i = 0; i < (int)topoSort.size(); i++)       // or you can simply read
  printf(" %d", topoSort[i]);           // the content of `topoSort' backwards
printf("\n");
printThis("Graph Edges Property Check");
numCC = 0;
dfs_num.assign(V, DFS_WHITE); dfs_parent.assign(V, -1);
for (int i = 0; i < V; i++)
  if (dfs_num[i] == DFS_WHITE)
    printf("Component %d:\n", ++numCC), graphCheck(i);       // 2 lines in one
printThis("Articulation Points & Bridges (the input graph must be UNDIRECTED)");
dfsNumberCounter = 0; dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0);
dfs_parent.assign(V, -1); articulation_vertex.assign(V, 0);
printf("Bridges:\n");
for (int i = 0; i < V; i++)
  if (dfs_num[i] == DFS_WHITE) {
    dfsRoot = i; rootChildren = 0;
```

```
      articulationPointAndBridge(i);
      articulation_vertex[dfsRoot] = (rootChildren > 1); }       // special case
  printf("Articulation Points:\n");
  for (int i = 0; i < V; i++)
    if (articulation_vertex[i])
      printf(" Vertex %d\n", i);
  printThis("Strongly Connected Components (the input graph must be DIRECTED)");
  dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0); visited.assign(V, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      tarjanSCC(i);
  return 0;
}---------------------------
```

//Binary indexed tree

```
#include <iostream>
using namespace std;
#define LOGSZ 17
int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);
// add v to value at x
void set(int x, int v) {
while(x <= N) {
tree[x] += v;
x += (x & -x);
}
}
// get cumulative sum up to and including x
int get(int x) {
int res = 0;
while(x) {
res += tree[x];
x -= (x & -x);
}
return res;
}
// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
int idx = 0, mask = N;
while(mask && idx < N) {
int t = idx + mask;
if(x >= tree[t]) {
idx = t;
```

```
x -= tree[t];
}
mask >>= 1;
}
return idx;
}
int main(){
        return 0 ;
}
```

//Trie simple implementation

```
#include<bits/stdc++.h>
using namespace std;
struct trienode
{
   bool leafnode;
   struct trienode* tarr[26];
};
typedef struct trienode trienode;
trienode* createnode()
{
   trienode* newnode=(trienode*)malloc(sizeof(trienode));
   newnode->leafnode=false;
   for(int i=0;i<26;i++)
   {
      newnode->tarr[i]=0;
   }
}
void inserttnode(trienode *root,string str)
{
   int len,i,j;
   len=str.size();
   trienode* ptrnode=root;
   for(i=0;i<len;i++)
   {
      int idx=(str[i]-'a');
      if(!ptrnode->tarr[idx])
      {
         ptrnode->tarr[idx]=createnode();
      }
      ptrnode=ptrnode->tarr[idx];
   }
   ptrnode->leafnode=true;
}
bool searchstr(trienode *root,string key)
```

```
{
   int len,i,j;
   len=key.size();
   trienode *ptrnode=root;
   for(i=0;i<len;i++)
   {
      int idx=int(key[i]-'a');
      if(!ptrnode->tarr[idx])
      {
         return false;
      }
      ptrnode=ptrnode->tarr[idx];
   }
   return (ptrnode!=NULL && ptrnode->leafnode);
}
int main()
{
   int n,i,j;
   string str[1000];
   cin>>n;
   trienode *root=createnode();
   for(i=0;i<n;i++)
   {
      cin>>str[i];
      inserttnode(root,str[i]);
   }
   string strtemp;
   int q;
   for(cin>>q;q;q--)
   {
      cin>>strtemp;
      bool val=searchstr(root,strtemp);
      if(val)
      {
         cout<<"the string exist\n";
      }
      else
      {
         cout<<"the string not found in the trie\n";
      }
   }
   return 0;
}
```

```
///* This function calculates (a^b)%c */
int modexpo(int a,int b,int c){
    long long x=1,y=a; // long long is taken to avoid overflow of intermediate results
    while(b > 0){
        if(b%2 == 1){
            x=(x*y)%c;
        }
        y = (y*y)%c; // squaring the base
        b /= 2;
    }
    return x%c;
}

/* this function calculates (a*b)%c taking into account that a*b might overflow */
long long mulmod(long long a,long long b,long long c){
    long long x = 0,y=a%c;
    while(b > 0){
        if(b%2 == 1){
            x = (x+y)%c;
        }
        y = (y*2)%c;
        b /= 2;
    }
    return x%c;
}

/* Miller-Rabin primality test, iteration signifies the accuracy of the test */
```

```
//Testing whether p is prime or not (20 is sufficient)
bool Miller(long long p,int iteration){
    if(p<2){
        return false;
    }
    if(p!=2 && p%2==0){
        return false;
    }
    long long s=p-1;
    while(s%2==0){
        s/=2;
    }
    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1,temp=s;
        long long mod=modulo(a,temp,p);
        while(temp!=p-1 && mod!=1 && mod!=p-1){
            mod=mulmod(mod,mod,p);
            temp *= 2;
        }
        if(mod!=p-1 && temp%2==0){
            return false;
        }   }
    return true;   }
```