

# Teambook Sindicato de Transporte 2880

Universidad Mayor de San Simón



2880

March 1, 2023



# Contents

<b>1</b>	<b>Mathematics</b>	<b>5</b>
1.1	GCD and LCM . . . . .	5
1.2	Prime Numbers . . . . .	5
1.3	Modular Arithmetic . . . . .	6
1.4	Matrix Exponentiation . . . . .	7
<b>2</b>	<b>Graphs</b>	<b>9</b>
2.1	Depth First Search (DFS) . . . . .	9
2.2	Breadth First Search (BFS) . . . . .	11



# Chapter 1

## Mathematics

This chapter is about some useful mathematical tools needed in order to solve problems.

### 1.1 GCD and LCM

In order to find the greatest common divisor (GCD) of two numbers, the Euclidean algorithm can be used. The implementation is as follows:

```
11 gcd(11 a, 11 b){return b==0? a:gcd(b,a%b);}

int x, y, d;
void extendedEuclid(int a, int b)//ecuacion diofantica ax + by = d
{
    if(b==0) {x=1; y=0; d=a; return;}
    extendedEuclid(b,a%b);
    int x1=y;
    y = x-(a/b)*y;
    x=x1;
}
```

../Mathematics/Euclid.cpp

Another (and faster) way to find the GCD is by using the following code:

```
int gcd(int a, int b) {
    if (!a || !b)
        return a | b;
    unsigned shift = __builtin_ctz(a | b);
    a >>= __builtin_ctz(a);
```

```
do {
    b >>= __builtin_ctz(b);
    if (a > b)
        swap(a, b);
    b -= a;
} while (b);
return a << shift;
}
```

../Mathematics/FastGCD.cpp

The way Halim suggests to find the GCD and the LCM is given by the following code:

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
int lcm(int a, int b) { return a / gcd(a, b) * b; }
```

../Mathematics/HalimGCD.cpp

### 1.2 Prime Numbers

The fastest way to check the primality of a number is by using Erathostenes' sieve. The typical implementation is as follows:

```
bitset<100000> bi;
vi primos; //primos
vector<ll> pric; //primos al cuadrado
void criba()
{
    bi.set();
```

```

for(int i=2;i<100000;i++)
    if(bi[i])
    {
        for(int j=i+i;j<100000;j+=i)
            bi[j]=0;
        primos.push_back(i);
        pric.push_back((ll)i*(ll)i);
    }
}
int euler(int n)
{
    int res=n;
    for(int i=0;pric[i]<=n;i++)
    {
        if(n%primos[i]==0)
        {
            res-= res/primos[i];
            while(n%primos[i]==0) n/=primos[i];
        }
    }
    if(n!=1) res-=res/n;
    return res;
}

```

../Mathematics/Erathostenes.cpp

Nevertheless, the following implementation is faster, since the statement `if (i % prime[j] == 0) break;` terminates the loop when `p` divides `i`. The inner loop is executed only once for each composite. Hence, the code performs in  $O(n)$  complexity, resulting in the 'linear' sieve:

```

// This algorithm allows to find Eratosthenes sieve in  $O(n \log n)$ 
// time.

std::vector<int> prime;
bool is_composite[MAXN];

void sieve (int n) {
    std::fill (is_composite, is_composite + n, false);
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) prime.push_back (i);

```

```

        for (int j = 0; j < prime.size () && i * prime[j] < n; ++j) {
            is_composite[i * prime[j]] = true;
            if (i % prime[j] == 0) break;
        }
    }

    // An application of this linear sieve is to find the Euler
    // totient function of a number in  $O(n \log n)$  time.
    std::vector<int> prime;
    bool is_composite[MAXN];
    int phi[MAXN];

    void sieve (int n) {
        std::fill (is_composite, is_composite + n, false);
        phi[1] = 1;
        for (int i = 2; i < n; ++i) {
            if (!is_composite[i]) {
                prime.push_back (i);
                phi[i] = i - 1; //i is prime
            }
            for (int j = 0; j < prime.size () && i * prime[j] < n; ++j) {
                is_composite[i * prime[j]] = true;
                if (i % prime[j] == 0) {
                    phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
                    divides i
                    break;
                } else {
                    phi[i * prime[j]] = phi[i] * phi[prime[j]]; //prime[j]
                    does not divide i
                }
            }
        }
    }
}

```

../Mathematics/LinearSieve.cpp

## 1.3 Modular Arithmetic

The modular inverse is defined by the following equation:

$$a \cdot a^{-1} \equiv 1 \pmod{m} \quad (1.1)$$

The following code shows how to find the modular inverse of a number:

```
int ModPow(int a, int b, int m) {
    int res = 1;
    while (b > 0) {
        if (b & 1) res = (res * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
    return res;
}

// Language: java
public static int modPow(int a, int b, int m) {
    int res = 1;
    while (b > 0) {
        if ((b & 1) == 1) res = (res * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
    return res;
}

int ModInverse(int a, int m) {
    return ModPow(a, m - 2, m);
}
```

../Mathematics/ModularInverse.cpp

Some other useful relationships in modular arithmetic are:

- $(a + b) \pmod{m} = (a \pmod{m} + b \pmod{m}) \pmod{m}$
- $(a - b) \pmod{m} = (a \pmod{m} - b \pmod{m}) \pmod{m}$
- $(a * b) \pmod{m} = (a \pmod{m} * b \pmod{m}) \pmod{m}$
- $(a/b) \pmod{m} = (a \pmod{m} * b^{-1} \pmod{m}) \pmod{m}$
- $(a^b) \pmod{m} = (a \pmod{m})^b \pmod{m}$
- $(a^b) \pmod{m} = (a \pmod{m})^{b \pmod{\phi(m)}} \pmod{m}$

- $(a^b) \pmod{m} = (a \pmod{m})^{b \pmod{m-1}} \pmod{m}$
- $\frac{a}{k} \equiv \frac{a}{k} \pmod{m} \iff a \equiv k \pmod{m}$
- $\frac{a}{k} \equiv \frac{a}{k} \pmod{\frac{n}{\gcd(n,k)}}$

## 1.4 Matrix Exponentiation

The following code shows how to find the nth power of a *mat*, noting that a data structure of type matrix is defined as follows:

```
typedef vector<vector<ll>> mat;
mat ans;
void mult(mat m1, mat m2)
{
    assert(m1[0].size() == m2.size());
    ans.clear();
    ll answer = 0;
    for(i, 0, m1.size())
    {
        vector<ll> fila;
        for(j, 0, m2[0].size())
        {
            answer = 0;
            for(k, 0, m2.size())
                answer = (answer + m1[i][k] * m2[k][j]) % MOD;
            fila.pb(answer);
        }
        ans.pb(fila);
    }
}

void pot(mat base, ll exp)
{
    mat res(base.size(), vector<ll>(base.size(), 0));
    for(i, 0, base.size())
        res[i][i] = 1;
    while(exp)
    {
        if(exp & 1)
        {
            mult(res, base);
        }
    }
}
```

```
        res = ans;
    }
    mult(base, base);
    base = ans;
    exp /= 2;
}
ans = res;
}
```

../Mathematics/MatrixPower.cpp



## Chapter 2

# Graphs

This chapter shows some of the basic algorithms and implementations required to solve problems that include graphs.

### 2.1 Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that visits all the nodes of a graph. It is used to find connected components, topological sorting, and to find bridges and articulation points. The algorithm is as follows:

The implementation can be done as follows:

```
vector<vector<int>> g(tam);
vector<bool> vis(tam);

void dfs(int u){
    vis[u]=true;
    ans++;
    for(int v: g[u]){
        if(!vis[v]){
            dfs(v);
        }
    }
}

signed main()
{
    int n,m;
    cin>>n>>m; // n nodes, m edges
    g.assign(tam,vector<int>());
```

```
    vis.assign(tam, false);
    for(int i=0; i<m;i++){
        int u,v;
        cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    ll res = 0;
    for(int i=1; i<=n;i++){
        if(!vis[i]){
            ans=0;
            dfs(i);
            res = max(res,ans);
        }
    }
    g.clear();
    vis.clear();
    return 0;
}
```

../Graphs/DFS.cpp

An application of this algorithm in order to find the shortest path between two nodes can be done as follows:

```
// The following code represents the implementation of a DFS
// algorithm
// to find the shortest path between two nodes in a graph.
// The graph is represented as an adjacency list.
// The algorithm is implemented using a stack.
```

**Algorithm 1** Depth First Search (DFS)

---

```

1: procedure DFS( $G$ )
2:    $visited \leftarrow \emptyset$ 
3:    $time \leftarrow 0$ 
4:    $parent \leftarrow \emptyset$ 
5:    $low \leftarrow \emptyset$ 
6:    $disc \leftarrow \emptyset$ 
7:    $AP \leftarrow \emptyset$ 
8:    $bridge \leftarrow \emptyset$ 
9:   for all  $v \in V$  do
10:     $visited[v] \leftarrow false$ 
11:     $parent[v] \leftarrow -1$ 
12:     $low[v] \leftarrow \infty$ 
13:     $disc[v] \leftarrow \infty$ 
14:   end for
15:   for all  $v \in V$  do
16:     if  $visited[v] = false$  then
17:       DFSUtil( $G, v, visited, time, parent, low, disc, AP, bridge$ )
18:     end if
19:   end for
20: end procedure
21: procedure DFSUTIL( $G, v, visited, time, parent, low, disc, AP, bridge$ )
22:    $visited[v] \leftarrow true$ 
23:    $disc[v] \leftarrow time$ 
24:    $low[v] \leftarrow time$ 
25:    $time \leftarrow time + 1$ 
26:    $children \leftarrow 0$ 
27:   for all  $u \in Adj(v)$  do
28:     if  $visited[u] = false$  then
29:        $parent[u] \leftarrow v$ 
30:        $children \leftarrow children + 1$ 
31:       DFSUtil( $G, u, visited, time, parent, low, disc, AP, bridge$ )
32:        $low[v] \leftarrow \min(low[v], low[u])$ 
33:       if  $parent[v] = -1$  and  $children > 1$  then
34:          $AP[v] \leftarrow true$ 
35:       end if
36:       if  $parent[v] \neq -1$  and  $low[u] \geq disc[v]$  then
37:          $AP[v] \leftarrow true$ 
38:       end if
39:       if  $low[u] > disc[v]$  then
40:          $bridge[v][u] \leftarrow true$ 
41:       end if
42:     else
43:        $low[v] \leftarrow \min(low[v], disc[u])$ 
44:     end if
45:   end for

```

---

```

#include <bits/stdc++.h>

using namespace std;

vector<int> DFS(vector<vector<int>> &adj, int s, int t) {
    stack<vector<int>> path_stack;
    vector<int> path;
    vector<int> visited(adj.size(), 0);
    path_stack.push({s});
    while (!path_stack.empty()) {
        path = path_stack.top();
        path_stack.pop();
        int last = path[path.size() - 1];
        if (last == t) {
            return path;
        }
        if (visited[last] == 0) {
            visited[last] = 1;
            for (int i = 0; i < adj[last].size(); i++) {
                if (visited[adj[last][i]] == 0) {
                    vector<int> new_path(path);
                    new_path.push_back(adj[last][i]);
                    path_stack.push(new_path);
                }
            }
        }
    }
    return {};
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> adj(n, vector<int>());
    for (int i = 0; i < m; i++) {
        int x, y;
        cin >> x >> y;
        adj[x - 1].push_back(y - 1);
        adj[y - 1].push_back(x - 1);
    }
}

```

```

}
int x, y;
cin >> x >> y;
x--, y--;
vector<int> path = DFS(adj, x, y);
for (int i = 0; i < path.size(); i++) {
    cout << path[i] + 1 << " ";
}
}

```

../Graphs/DFS-application.cpp

## 2.2 Breadth First Search (BFS)

The BFS algorithm is a non-recursive algorithm that visits all the nodes of a graph. It is used to find connected components, topological sorting, and to find bridges and articulation points, to better understand it, a propagating fire can be imagined. The algorithm is as follows:

The implementation can be done as follows:

```

#include <bits/stdc++.h>

using namespace std;
signed main()
{
    vector<vector<int>> adj; // adjacency list representation
    int n; // number of nodes
    int s; // source vertex

    queue<int> q;
    vector<bool> used(n);
    vector<int> d(n), p(n);

    q.push(s);
    used[s] = true;
    p[s] = -1;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int u : adj[v]) {
            if (!used[u]) {

```

---

### Algorithm 2 Breadth First Search (BFS)

---

```

1: procedure BFS( $G$ )
2:    $visited \leftarrow \emptyset$ 
3:    $time \leftarrow 0$ 
4:    $parent \leftarrow \emptyset$ 
5:    $low \leftarrow \emptyset$ 
6:    $disc \leftarrow \emptyset$ 
7:    $AP \leftarrow \emptyset$ 
8:    $bridge \leftarrow \emptyset$ 
9:   for all  $v \in V$  do
10:     $visited[v] \leftarrow false$ 
11:     $parent[v] \leftarrow -1$ 
12:     $low[v] \leftarrow \infty$ 
13:     $disc[v] \leftarrow \infty$ 
14:   end for
15:   for all  $v \in V$  do
16:     if  $visited[v] = false$  then
17:       BFSUtil( $G, v, visited, time, parent, low, disc, AP, bridge$ )
18:     end if
19:   end for
20: end procedure
21: procedure BFSUtil( $G, v, visited, time, parent, low, disc, AP, bridge$ )
22:    $visited[v] \leftarrow true$ 
23:    $disc[v] \leftarrow time$ 
24:    $low[v] \leftarrow time$ 
25:    $time \leftarrow time + 1$ 
26:    $children \leftarrow 0$ 
27:   for all  $u \in Adj(v)$  do
28:     if  $visited[u] = false$  then
29:        $parent[u] \leftarrow v$ 
30:        $children \leftarrow children + 1$ 
31:       BFSUtil( $G, u, visited, time, parent, low, disc, AP, bridge$ )
32:        $low[v] \leftarrow \min(low[v], low[u])$ 
33:       if  $parent[v] = -1$  and  $children > 1$  then
34:          $AP[v] \leftarrow true$ 
35:       end if
36:       if  $parent[v] \neq -1$  and  $low[u] \geq disc[v]$  then
37:          $AP[v] \leftarrow true$ 
38:       end if
39:       if  $low[u] > disc[v]$  then
40:          $bridge[v][u] \leftarrow true$ 
41:       end if
42:     else
43:        $low[v] \leftarrow \min(low[v], disc[u])$ 
44:     end if
45:   end for

```

```
        used[u] = true;
        q.push(u);
        d[u] = d[v] + 1;
        p[u] = v;
    }
}
return 0;
}
```

../Graphs/BFS.cpp