Teambook Sindicato de Transporte 2880

February 13, 2023

Contents

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:

```cpp
ll gcd(ll a, ll 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:

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

```cpp
    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:

```cpp
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:

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

```cpp
   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;
}
```

<div align="center">Mathematics/Erathostenes.cpp</div>

Nevertheless, the following implementation is faster, since the statement if

```cpp
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:

```cpp
// This algorithm allows to find Eratosthenes sieve in O(n logn)
   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);
```

```cpp
      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 lineatr sieve is to find the Euler
   totient function of a number in O(n logn) 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
         }
      }
   }
}
```

<div align="center">Mathematics/LinearSieve.cpp</div>

Chapter 2

Graphs

This chapter shows some of the basec 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:

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:

```cpp
#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;
```

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

Graphs/BFS.cpp

Algorithm 1 Depth First Search (DFS)

```
 1: procedure DFS(G)
 2:     visited ← ∅
 3:     time ← 0
 4:     parent ← ∅
 5:     low ← ∅
 6:     disc ← ∅
 7:     AP ← ∅
 8:     bridge ← ∅
 9:     for all v ∈ V do
10:         visited[v] ← false
11:         parent[v] ← −1
12:         low[v] ← ∞
13:         disc[v] ← ∞
14:     end for
15:     for all v ∈ 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] ← true
23:     disc[v] ← time
24:     low[v] ← time
25:     time ← time + 1
26:     children ← 0
27:     for all u ∈ Adj(v) do
28:         if visited[u] = false then
29:             parent[u] ← v
30:             children ← children + 1
31:             DFSUtil(G, u, visited, time, parent, low, disc, AP, bridge)
32:             low[v] ← min(low[v], low[u])
33:             if parent[v] = −1 and children > 1 then
34:                 AP[v] ← true
35:             end if
36:             if parent[v]! = −1 and low[u] ≥ disc[v] then
37:                 AP[v] ← true
38:             end if
39:             if low[u] > disc[v] then
40:                 bridge[v][u] ← true
41:             end if
42:         else
43:             low[v] ← min(low[v], disc[u])
44:         end if
45:     end for
```

Algorithm 2 Breadth First Search (BFS)

```
 1: procedure BFS(G)
 2:     visited ← ∅
 3:     time ← 0
 4:     parent ← ∅
 5:     low ← ∅
 6:     disc ← ∅
 7:     AP ← ∅
 8:     bridge ← ∅
 9:     for all v ∈ V do
10:         visited[v] ← false
11:         parent[v] ← −1
12:         low[v] ← ∞
13:         disc[v] ← ∞
14:     end for
15:     for all v ∈ 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] ← true
23:     disc[v] ← time
24:     low[v] ← time
25:     time ← time + 1
26:     children ← 0
27:     for all u ∈ Adj(v) do
28:         if visited[u] = false then
29:             parent[u] ← v
30:             children ← children + 1
31:             BFSUtil(G, u, visited, time, parent, low, disc, AP, bridge)
32:             low[v] ← min(low[v], low[u])
33:             if parent[v] = −1 and children > 1 then
34:                 AP[v] ← true
35:             end if
36:             if parent[v]! = −1 and low[u] ≥ disc[v] then
37:                 AP[v] ← true
38:             end if
39:             if low[u] > disc[v] then
40:                 bridge[v][u] ← true
41:             end if
42:         else
43:             low[v] ← min(low[v], disc[u])
44:         end if
45:     end for
```