Pitón++

Contents

1 C++

1.1 C++ template

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

//IMPRESINDIBLES PARA ICPC
#define form(i, s, e) for(int i = s; i < e; i++)
#define icin(x)   \
  int x;          \
  cin >> x;
#define llcin(x) \
  long long x;   \
  cin >> x;
#define scin(x)  \
string x;        \
cin >> x;
#define endl '\n'
#define S second
#define F first
#define pb push_back
```

```cpp
#define sz(x) x.size()
#define all(x) x.begin(),x.end()

typedef long long ll;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int,int> pii;

const ll INF =  1e9+7;//tambien es primo
const double PI = acos(-1);
//UTILES
#define DBG(x) cerr << #x << '=' << (x) << endl
#define coutDouble cout << fixed << setprecision(17)
#define numtobin(n) bitset<32>(n).to_string()
#define bintoint(bin_str) stoi(bin_str, nullptr, 2)  //
    bin_str should be a STRING
#define LSOne(S) ((S) & -(S))

typedef double db;
typedef vector<string> vs;
typedef vector<ll> vll;
typedef vector<vll> vvll;
typedef pair<int,bool> pib;
typedef pair<ll,ll> pll;
typedef vector<pii> vpii;
typedef vector<pib> vpib;
typedef vector<pll> vpll;


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

  icin(nn0)
  while (nn0--) {

  }

  return 0;
}
```

1.2 Librerias sino da bits/stdc++

```cpp
// En caso de que no sirva #include <bits/stdc++.h>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <sstream>
#include <fstream>
#include <cassert>
#include <climits>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <deque>
#include <stack>
#include <list>
#include <map>
#include <set>
#include <bitset>
#include <iomanip>
#include <unordered_map>

////
#include <tuple>
#include <random>
#include <chrono>
```

1.3 Comando para comparar salidas

1.3.1 Linux

```
./programa < in.txt > myout.txt
diff -u out.txt myout.txt
```

1.3.2 Windows

```
algo2.exe < in.txt > myout.txt
fc myout.txt out.txt
```

1.4 C++

1.4.1 Ejemplo problematico if-else

A primera vista podrías pensar que el else se asocia al primer if (if (a >0)), pero en realidad se asocia al segundo (if (b >0)).

```cpp
if (a > 0)
  if (b > 0)
      cout << "Ambos positivos";
else
    cout << "a no es positivo";
```

1.4.2 vector

```cpp
// RESIZE
vector<int> v = {1, 2, 3};
v.resize(5, 9);  // ahora: {1, 2, 3, 9, 9}
v.resize(2);     // ahora: {1, 2}
```

```
// ASSIGN
vector<int> v = {1, 2, 3};
v.assign(5, 9);  // ahora: {9, 9, 9, 9, 9}
```

2 Grafos

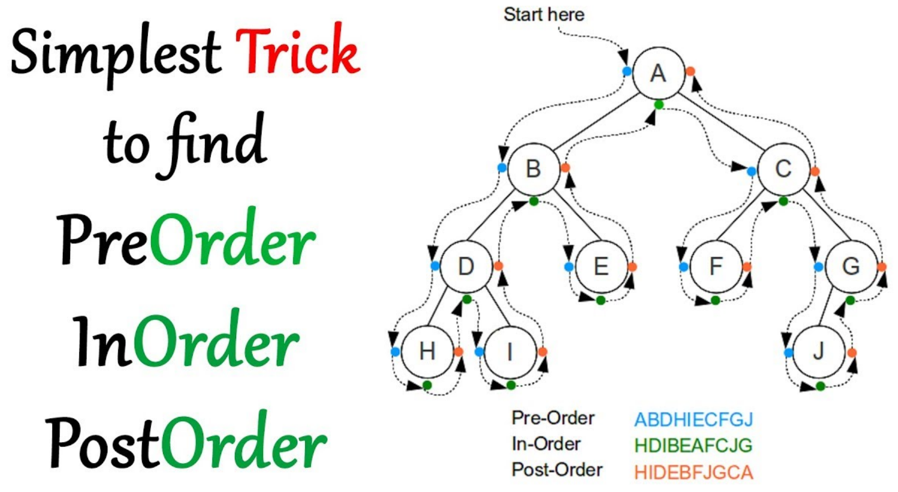

Figure 1: Pre-In-Post Orders DFS

2.1 DFS cpbook

```
enum { UNVISITED = -1, VISITED = -2 };
                        // basic flags

// these variables have to be global to be easily
   accessible by our recursion (other ways exist)
vector<vii> AL;
vi dfs_num;

void dfs(int u) {                                      //
   normal usage
  printf(" %d", u);                                    // this
     vertex is visited
  dfs_num[u] = VISITED;                                // mark
     u as visited
  for (auto &[v, w] : AL[u])                           // C++17
      style, w ignored
    if (dfs_num[v] == UNVISITED)                       // to
       avoid cycle
      dfs(v);                                          //
         recursively visits v
}

int main() {
  /*
  // Undirected 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
  */

  freopen("dfs_cc_in.txt", "r", stdin);

  int V; scanf("%d", &V);
  AL.assign(V, vii());
  for (int u = 0; u < V; ++u) {
    int k; scanf("%d", &k);
    while (k--) {
      int v, w; scanf("%d %d", &v, &w);
      AL[u].emplace_back(v, w);
    }
  }

  printf("Standard DFS Demo (the input graph must be
     UNDIRECTED)\n");
  dfs_num.assign(V, UNVISITED);
  int numCC = 0;
  for (int u = 0; u < V; ++u)                          // for
     each u in [0..V-1]
    if (dfs_num[u] == UNVISITED)                       // if
       that u is unvisited
      printf("CC %d:", ++numCC), dfs(u), printf("\n"); //
         3 lines here!
  printf("There are %d connected components\n", numCC);

  return 0;
}
```

2.2 DFS iterativo - Lucas

```
vector<bool> vis;
void dfs(int start, vector<vector<int>> & adj, int v){
  // v = Vertices
  stack<int> s;
  s.push(start);
```

```
    vis[start] = true;
    int cont = 1;
    while (!(s.empty())){
      int prox = s.top();
      if(!(adj[prox].empty())){
        if(vis[adj[prox].back()] == false){
          vis[adj[prox].back()] = true;
          s.push(adj[prox].back());
        }
        else{
          adj[prox].pop_back();
        }
      }
      else{
        s.pop();
      }
    }
  }
```

2.3 BFS - camino mas corto - O(V+E)

```
// inside int main()---no recursion
vi dist(V, INF); dist[s] = 0; // initial distances
queue<int> q; q.push(s);  // start from source
while (!q.empty()) { // queue: layer by layer!
   int u = q.front(); q.pop(); // C++17 style, w ignored
   for (auto &[v, w] : AL[u]) {
   if (dist[v] != INF) continue;// ALREADY VISITED, skip
   dist[v] = dist[u]+1;// now set dist[v] != INF
   q.push(v);// for the next iteratio
   }
}
```

2.4 BFS - bipartito check - Lucas - O(V+E)

```
// Realiza una BFS desde el nodo 'src' en un grafo
    dirigido o no dirigido
// representado como lista de adyacencia.
// Parametros:
//   n   : numero de nodos (0 .. n-1)
//   adj : vector de vectores, donde adj[u] contiene
    todos los v tales que u -> v
//   src : nodo de partida
// Devuelve:
//   true si es bipartito y false si no lo es
bool bfs(int n, vector<pair<vector<int>, char>> &adj, int
    src)
{
    queue<int> q;
    q.push(src);
    char decision = 'a';
    bool bipartito = true;
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        if (adj[u].second == 'c')
        {
            adj[u].second = decision;
        }
        if (adj[u].second == 'a')
            decision = 'b';
        else
            decision = 'a';
        for (int v : adj[u].first)
        {
            if (adj[v].second == 'c')
            {
                q.push(v);
                adj[v].second = decision;
            }
            if (adj[u].second == adj[v].second)
            {
                bipartito = false;
                break;
            }
        }
    }
    return bipartito;
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    // Leer numero de nodos y aristas
    cin >> n >> m;
    // Construir lista de adyacencia
    vector<pair<vector<int>, char>> adj(n);
    // a= 1er conjunto
    // b = 2do
    // c = sin conjunto
    for (int i = 0; i < m; ++i)
    {
        int u, v;
        cin >> u >> v;
        adj[u].first.push_back(v);
        adj[v].first.push_back(u);
    }
    // inicializacion en c para saber si no esta
        explorado
    for (int i = 0; i < n; i++)
        adj[i].second = 'c';
```

```
    bool es_bipartito = true;
    // Iterar por todos los nodos para manejar grafos no conexos
    for (int i = 0; i < n; ++i)
    {
        // Si el nodo 'i' no ha sido coloreado, iniciar un BFS desde el
        if (adj[i].second == 'c')
        {
            // Si cualquier componente no es bipartita, el grafo entero no lo es
            if (!bfs(n, adj, i))
            {
                es_bipartito = false;
                break; // Podemos detenernos en cuanto encontramos un fallo
            }
        }
    }
    cout << "res: " << es_bipartito << endl;
    return 0;
}
```

2.5 BFS - cpbook - camino mas corto - Bipartito check O(V+E)

```
const int INF = 1e9; // INF = 1B, not 2^31-1 to avoid overflow
vi p;                                            // addition:parent vector

void printPath(int u) {                          // extract info from vi p
  if (p[u] == -1) { printf("%d", u); return; }
  printPath(p[u]);                               // 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("bfs_in.txt", "r", stdin);

  int V, E; scanf("%d %d", &V, &E);
  vector<vii> AL(V, vii());
  for (int i = 0; i < E; ++i) {
    int a, b; scanf("%d %d", &a, &b);
    AL[a].emplace_back(b, 0);
    AL[b].emplace_back(a, 0);
  }

  // as an example, we start from this source, see Figure 4.3
  int s = 5;

  // BFS routine inside int main() -- we do not use recursion
  vi dist(V, INF); dist[s] = 0;                  // INF = 1e9 here
  queue<int> q; q.push(s);
  p.assign(V, -1);                               // p is global

  int layer = -1;                                // for output printing
  bool isBipartite = true;                       // additional feature

  while (!q.empty()) {
    int u = q.front(); q.pop();
    if (dist[u] != layer) printf("\nLayer %d: ", dist[u]);
    layer = dist[u];
    printf("visit %d, ", u);
    for (auto &[v, w] : AL[u]) {                 // C++17 style, w ignored
      if (dist[v] == INF) {
        dist[v] = dist[u]+1;                     // dist[v] != INF now
        p[v] = u;                                // parent of v is u
        q.push(v);                               // for next iteration
      }
      else if ((dist[v]%2) == (dist[u]%2))       // same parity
        isBipartite = false;
    }
  }

  printf("\nShortest path: ");
  printPath(7), printf("\n");
  printf("isBipartite? %d\n", isBipartite);

  return 0;
}
```

2.6 DFS - detect cycle - O(V+E)

```
vector<vector<int>> adj(5);
```

```
int n;
vector<char> state(5);
/*
a = no visitado
b = visitando
c = visitado
*/

bool dfs_detect_cycle(int node)
{
    if(state[node] == 'b')
        return true;
    state[node] = 'b';
    for(auto i: adj[node])
    {
        if(dfs_detect_cycle(i))
        {
            return true;
        }
    }
    state[node] = 'c';
    return false;
}

int main()
{
    ios::sync_with_stdio(0);cin.tie(0); cout.tie(0);
    n = 5;
    adj[1].push_back(2);
    // Componente 2 (con ciclo)
    adj[3].push_back(4);
    adj[4].push_back(0);
    // adj[0].push_back(3); // CON ESTO SI HAY CICLO

    form(i,0,5) state[i] = 'a';
    int i;
    for( i=0;i < 5; i++)
    {
        if(state[i] == 'a')
            if(dfs_detect_cycle(i))
            {
                cout << "Hay ciclo" << endl;
                return 0;
            }
    }
    if(i == 5)
        cout << "NO hay ciclo" << endl;

    return 0;
}
```

2.7 Dijkstra - O((E + V)log V)

- Para pesos >= 0 porque generan ciclos infinitos, pero si hay negativos pero no pueden generar ciclos entonces si se puede usar Dijkstra. - O((E + V)log V) implementación con Heap Binario (priorityQueue C++). - O(E + V(log V)) implementación con Heap Fibonacci (Teórico) que soporta decreaseKey.

```
vector<long long> dist;
struct cmp {
    bool operator()(const pair<int, long long>& a, const
        pair<int, long long>& b) const {
        return a.second > b.second;
    }
};
priority_queue<pair<int, long long>, vector<pair<int,
    long long>>, cmp> q;

void dijkstra(int n, vector<vector<pair<int,long long>>>
    &adj, int src)
{
    dist.resize(n+1, -1);
    dist[src] = 0;
    q.push({src,0});
    while (!q.empty())
    {
        auto u = q.top();
        q.pop();
        if (u.second > dist[u.first])
        {
            continue; // Ya encontramos un camino mas
                corto a 'u', ignoramos este.
        }
        for (auto v : adj[u.first])
        {
            if (dist[v.first] > dist[u.first] + v.second
                or dist[v.first] == -1)
            {
                dist[v.first] = dist[u.first] + v.second;
                q.push({v.first, dist[v.first]});
            }

        }
    }
    true;
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    int u, v;
    long long p;
    vector<vector<pair<int,long long>>> adj(n+1);//nodo
        destino, peso
    for (int i = 0; i < m; ++i)
    {
        cin >> u >> v >> p;
        adj[u].push_back({v,p});
```

```
    }
    dijkstra(n, adj, 1);// desde nodo origen a todos los
        demas
    for (int i = 1; i <= n; ++i)
    {
        cout << dist[i] << " ";
    }
    return 0;
}
```

2.8 Topological sort dfs O(V+E)

- Esta version no puede detectar ciclos entonces primero EJECUTAR CYCLE CHECK. - Si existe una arista u –>v entonces u aparece antes que v en el orden topológico. - Ya que es recursivo no soporta grafos muy grandes mayores o iguales 10^5 nodos. - El orden topológico solo existe en un DAG (Grafo Dirigido Acíclico). Si el grafo tiene un ciclo, no puedes ordenarlo.

```
// Para que exista un orden topologico
// el grafo tiene que ser DAG(grafo aciclico dirigido)
vi g[nax], ts;
bool seen[nax];
void dfs(int u){
  seen[u] = true;
  for(int v: g[u])
    if (!seen[v])
      dfs(v);
  ts.pb(u);
}
void topo(int n){
  forn(i,n) if (!seen[i]) dfs(i);
  reverse(all(ts));
}
```

2.9 Topological sort bfs (Kahn's algorithm con queue O(V+E)/ con priority_queue O((V+E)log V))

- Si existe una arista u –>v entonces u aparece antes que v en el orden topológico. - Gracias a que se puede usar una priority_queue en vez de una queue, obtenemos el orden lexicográficamente menor, mayor o cualquier otro orden dependiendo de la comparación que definamos en la priority_queue. - El orden topológico solo existe en un DAG (Grafo Dirigido Acíclico). Si el grafo tiene un ciclo, no puedes ordenarlo. - Al final del algoritmo, si el número de nodos en tu lista de orden topológico no es igual al número total de nodos en el grafo (N), entonces existe un ciclo. Los nodos que se quedaron con indegree ¿ 0 son parte del ciclo o son alcanzables desde uno.

```
// enqueue vertices with zero incoming degree into a (
   priority) queue pq
priority_queue<int, vi, greater<int>> pq;      // min
   priority queue
for (int u = 0; u < N; ++u)
  if (in_degree[u] == 0)                       // next to
    be processed
    pq.push(u);                                // smaller
      index first

while (!pq.empty()) {                          // Kahn's
   algorithm
  int u = pq.top(); pq.pop();
  printf(" %s", u);   // process u here
  for (auto &v : AL[u]) {
    --in_degree[v];                            // virtually
      remove u->v
    if (in_degree[v] > 0) continue;            // not a
      candidate, skip
    pq.push(v);                                // enqueue v
      in pq
  }
}
```

2.10 Tarjan cpbook - Strongly Connected Components O(V+E)

```
enum { UNVISITED = -1 };

int dfsNumberCounter, numSCC;
vector<vii> AL, AL_T;
vi dfs_num, dfs_low, S, visited;                      //
   global variables
stack<int> St;

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter;    //
    dfs_low[u]<=dfs_num[u]
  dfsNumberCounter++;                            //
    increase counter
  St.push(u);                                    //
    remember the order
  visited[u] = 1;
  for (auto &[v, w] : AL[u]) {
    if (dfs_num[v] == UNVISITED)
      tarjanSCC(v);
    if (visited[v])                              //
      condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v]);
  }
```

```
  if (dfs_low[u] == dfs_num[u]) {                      // a
      root/start of an SCC
    ++numSCC;                                          // when
        recursion unwinds
    while (1) {
      int v = St.top(); St.pop(); visited[v] = 0;
      if (u == v) break;
    }
  }
}

void Kosaraju(int u, int pass) { // pass = 1 (original),
    2 (transpose)
  dfs_num[u] = 1;
  vii &neighbor = (pass == 1) ? AL[u] : AL_T[u]; // by
      ref to avoid copying
  for (auto &[v, w] : neighbor)                        // C++17
       style, w ignored
    if (dfs_num[v] == UNVISITED)
      Kosaraju(v, pass);
  S.push_back(u); // as in finding topological order in
      Section 4.2.5
}

int main() {
  int N, M;
  while (scanf("%d %d", &N, &M), (N || M)) {
    AL.assign(N, vii());
    AL_T.assign(N, vii()); // the transposed graph
    while (M--) {
      int V, W, P; scanf("%d %d %d", &V, &W, &P); --V; --
          W; // to 0-based indexing
      AL[V].emplace_back(W, 1); // always
      AL_T[W].emplace_back(V, 1);
      if (P == 2) { // if this is two way, add the
          reverse direction
        AL[W].emplace_back(V, 1);
        AL_T[V].emplace_back(W, 1);
      }
    }

    // run Tarjan's SCC code here
    dfs_num.assign(N, UNVISITED); dfs_low.assign(N, 0);
        visited.assign(N, 0);
    while (!St.empty()) St.pop();
    dfsNumberCounter = numSCC = 0;
    for (int u = 0; u < N; ++u)
     if (dfs_num[u] == UNVISITED)
       tarjanSCC(u);

    // // run Kosaraju's SCC code here
    // S.clear(); // first pass: record the post-order of
        original graph
    // dfs_num.assign(N, UNVISITED);
    // for (int u = 0; u < N; ++u)
    //   if (dfs_num[u] == UNVISITED)
    //     Kosaraju(u, 1);
    // int numSCC = 0; // second pass: explore SCCs using
        first pass order
    // dfs_num.assign(N, UNVISITED);
    // for (int i = N-1; i >= 0; --i)
    //   if (dfs_num[S[i]] == UNVISITED)
    //     numSCC++, Kosaraju(S[i], 2);              // on
        transposed graph

    // if SCC is only 1, print 1, otherwise, print 0
    printf("%d\n", numSCC );
  }
  return 0;
}
```

3 Data Structures

3.1 unordered_map<clave,valor>

Almacena pares clave valor.

```
// hacer siempre RESERVE
unordered\_map<int,int> a;
a.reserve(n*1.33); IMPORTANTEEEEEEE
n = 1e6 aprox 42.6 MB

n = 3e6 aprox 128 MB

n = 5e6 aprox 213 MB (aún puede entrar, pero ojo con pila
, I/O buffers, otros contenedores).
```

3.1.1 Ejemplo basico Contar frecuencias

```
int n;
cin >> n;
vector<int> arr(n);
for (int &x : arr)
  cin >> x;

unordered_map<int,int> freq;//<clave, valor>
freq.reserve(n*1.33); // evita rehash

for (int x : arr)
  freq[x]++;

for (auto &p : freq)
  cout << p.first << " aparece " << p.second << " veces
      \n";
```

3.1.2 Buscar existencia de una llave

```
unordered_map<string,int> id;
id.reserve(1e5);

id["uva"] = 10;
id["manzana"] = 20;

// Con count
if (id.count("uva")) cout << "uva existe\n";
```

3.1.3 Transformar índices dispersos a continuos

```
vector<int> vals = {1000, 5000, 1000, 42};
unordered_map<int,int> comp;
comp.reserve(vals.size()*1.33);

int id = 0;
for (int v : vals)
    if (!comp.count(v))
      comp[v] = id++;
/*
  Antes -> Ahora
  1000 = 1
  5000 = 2
  42 = 3
*/
for (int v : vals)
    cout << v << " -> " << comp[v] << "\n";
```

3.1.4 Hashing pair

```
struct pair_hash {
    size_t operator()(const pair<int,int>& p) const {
        return ((long long)p.first << 32) ^ p.second;
    }
};

int main()
{
  unordered_map<pair<int,int>, int, pair_hash> edge_cost;
  edge_cost.reserve(1e6);
  //Muy usado para representar grafos dispersos.
  edge_cost[{1,2}] = 5;
  edge_cost[{2,3}] = 10;

  cout << edge_cost[{1,2}] << "\n"; // 5
}
```

3.2 unordered_set<clave>

// hacer siempre RESERVE
No existe acceso aleatorio con [] (índices),
pero se puede iterar con for auto.

```
int n = 3e5;
vi a = {1,2,3,42,42,42};
unordered_set<int> s;//<T>
s.reserve(n * 1.3);// evita rehash

//insert(T)
for (int x : a)
  s.insert(x);

//VERIFICAR EXISTENCIA
if (s.find(42) != s.end())
  cout << "42 existe" << endl;

//Iterar para ver claves existentes
for(auto x : s)
  cout << x << " ";

return 0;
```

3.3 unordered_multimap<clave,valores>

// hacer siempre RESERVE
Una misma clave puede tener varios valores asociados

3.3.1 Ejemplo básico

```
multimap<int,string> mm;

// insertar pares (clave, valor)
mm.insert({1, "uva"});
mm.insert({2, "manzana"});
mm.insert({2, "pera"});
mm.insert({3, "melon"});

// Iterar (se imprime ordenado por clave)
for (auto &p : mm)
    cout << p.first << " -> " << p.second << "\n";
/*
1 -> uva
2 -> manzana
2 -> pera
3 -> melon
*/
```

3.3.2 Buscar por clave

```
multimap<int,string> mm;
// insertar pares (clave, valor)
mm.insert({1, "uva"});
mm.insert({2, "manzana"});
mm.insert({2, "pera"});
mm.insert({3, "melon"});
```

```
// Buscar la primera aparicion de clave 2
auto it = mm.find(2);
if (it != mm.end())
    cout << "Encontrado: " << it->second << "\n";

// Contar cuantos con clave=2
cout << "Claves con 2: " << mm.count(2) << "\n";

// Obtener todos los con clave=2
auto [ini, fin] = mm.equal_range(2);
for (auto it = ini; it != fin; ++it)
    cout << it->second << " ";
/*
SALIDA
Encontrado: manzana
Claves con 2: 2
manzana pera
*/
```

3.3.3 Delete

```
mm.erase(2);  // borra *todas* las entradas con clave=2

// Si quieres borrar solo uno:
auto it = mm.find(2);
if (it != mm.end())
  mm.erase(it);
```

3.4 Union find - cpbook

Cada que unimos dos Sets del mismo RANK(rank=r=altura - size=s=vertices) nuestro rank aumenta en +1. Entonces para formar un RANK r se necesitan por lo menos 2^r vertices.

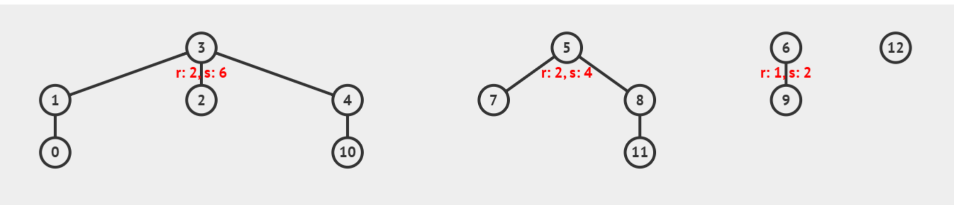

Figure 2: Inicializacion de Union-Find. Cada nodo es su propio padre.

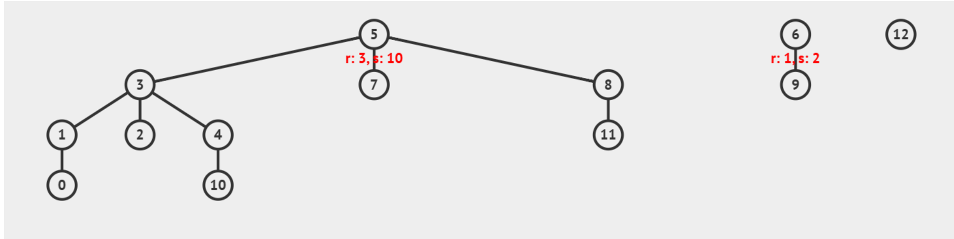

Figure 3: Union-Find despues de unir 3 y 5

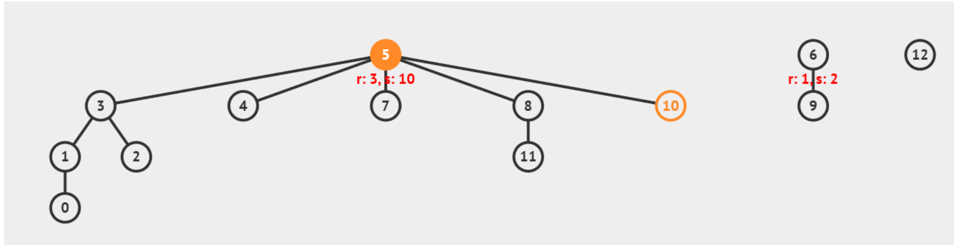

Figure 4: Union-Find despues findSet(10) con Path Compression

```
// Union-Find Disjoint Sets Library written in OOP manner
   , using both path compression and union by rank
   heuristics

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

typedef vector<int> vi;

class UnionFind {                                    // OOP
   style
private:
  vi p, rank, setSize;                               // vi p
     is the key part
  int numSets;
public:
  UnionFind(int N) {
    p.assign(N, 0); for (int i = 0; i < N; ++i) p[i] = i;
    rank.assign(N, 0);                               //
       optional speedup
    setSize.assign(N, 1);                            //
       optional feature
    numSets = N;                                     //
       optional feature
```

```
  }
  int findSet(int i) {
    return (p[i] == i) ? i : (p[i] = findSet(p[i])); //
        Path Compression
  }
  bool isSameSet(int i, int j) { return findSet(i) ==
      findSet(j); }

  int numDisjointSets() { return numSets; }       //
      optional
  int sizeOfSet(int i) { return setSize[findSet(i)]; } //
       optional

  void unionSet(int i, int j) {
    if (isSameSet(i, j)) return;                   // i and
         j are in same set
    int x = findSet(i), y = findSet(j);            // find
        both rep items
    if (rank[x] > rank[y]) swap(x, y);             // keep
        x 'shorter' than y
    p[x] = y;                                      // set x
         under y
    if (rank[x] == rank[y]) ++rank[y];             //
        optional speedup
    setSize[y] += setSize[x];                      //
        combine set sizes at y
    --numSets;                                     // a
        union reduces numSets
  }
};

int main() {
  printf("Assume that there are 5 disjoint sets initially
      \n");
  UnionFind UF(17); // create 5 disjoint sets
  UF.unionSet(1,2);
  UF.unionSet(3,4);
  UF.unionSet(1,3);
  UF.unionSet(5,6);
  UF.unionSet(7,8);
  UF.unionSet(5,7);
  UF.unionSet(1,5);

  UF.unionSet(9,10);
  UF.unionSet(11,12);
  UF.unionSet(9,11);
  UF.unionSet(13,14);
  UF.unionSet(15,16);
  UF.unionSet(13,16);
  UF.unionSet(9,13);

  UF.unionSet(9,1);
  UF.findSet(10);
  UF.findSet(11);

  int a = 1 + 2;
  printf("isSameSet(0, 3) = %d\n", UF.isSameSet(0, 3));
      // will return 0 (false)
  printf("isSameSet(4, 3) = %d\n", UF.isSameSet(4, 3));
      // will return 1 (true)
  for (int i = 0; i < 5; i++) // findSet will return 1
      for {0, 1} and 3 for {2, 3, 4}
    printf("findSet(%d) = %d, sizeOfSet(%d) = %d\n", i,
        UF.findSet(i), i, UF.sizeOfSet(i));
  UF.unionSet(0, 3);
  printf("%d\n", UF.numDisjointSets()); // 1
  for (int i = 0; i < 5; i++) // findSet will return 3
      for {0, 1, 2, 3, 4}
    printf("findSet(%d) = %d, sizeOfSet(%d) = %d\n", i,
        UF.findSet(i), i, UF.sizeOfSet(i));
  return 0;
}
```

3.5 Fenwick Tree - cpbook

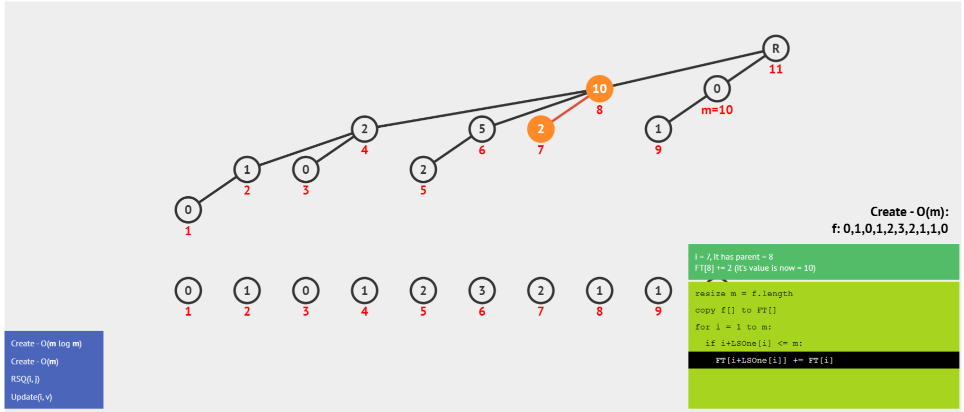

Figure 5: Fenwick Tree (Binary Indexed Tree)

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

#define LSOne(S) ((S) & -(S))                  // the
    key operation

typedef long long ll;                          // for
    extra flexibility
typedef vector<ll> vll;
typedef vector<int> vi;
```

```
class FenwickTree {                                    // index
    0 is not used
private:
  vll ft;                                              //
      internal FT is an array
public:
  FenwickTree(int m) { ft.assign(m+1, 0); }           //
      create an empty FT

  void build(const vll &f) {
    int m = (int)f.size()-1;                           // note
        f[0] is always 0
    ft.assign(m+1, 0);
    for (int i = 1; i <= m; ++i) {                     // O(m)
      ft[i] += f[i];                                   // add
          this value
      if (i+LSOne(i) <= m)                             // i has
          parent
        ft[i+LSOne(i)] += ft[i];                       // add
            to that parent
    }
  }

  FenwickTree(const vll &f) { build(f); }              //
      create FT based on f

  FenwickTree(int m, const vi &s) {                    //
      create FT based on s
    vll f(m+1, 0);
    for (int i = 0; i < (int)s.size(); ++i)           // do
        the conversion first
      ++f[s[i]];                                       // in O(
          n)
    build(f);                                          // in O(
        m)
  }

  ll rsq(int j) {                                      //
      returns RSQ(1, j)
    ll sum = 0;
    for (; j; j -= LSOne(j))
      sum += ft[j];
    return sum;
  }

  ll rsq(int i, int j) { return rsq(j) - rsq(i-1); } //
      inc/exclusion

  // updates value of the i-th element by v (v can be +ve
      /inc or -ve/dec)
  void update(int i, ll v) {
    for (; i < (int)ft.size(); i += LSOne(i))
      ft[i] += v;
  }

  int select(ll k) {                                   // O(log
      m)
    int p = 1;
    while (p*2 < (int)ft.size()) p *= 2;
    int i = 0;
    while (p) {
      if (k > ft[i+p]) {
        k -= ft[i+p];
        i += p;
      }
      p /= 2;
    }
    return i+1;
  }
};

class RUPQ {                                           // RUPQ
    variant
private:
  FenwickTree ft;                                      //
      internally use PURQ FT
public:
  RUPQ(int m) : ft(FenwickTree(m)) {}
  void range_update(int ui, int uj, ll v) {
    ft.update(ui, v);                                  // [ui,
        ui+1, .., m] +v
    ft.update(uj+1, -v);                               // [uj
        +1, uj+2, .., m] -v
  }                                                    // [ui,
      ui+1, .., uj] +v
  ll point_query(int i) { return ft.rsq(i); }          // rsq(i
      ) is sufficient
};

class RURQ  {                                          // RURQ
    variant
private:                                               // needs
     two helper FTs
  RUPQ rupq;                                           // one
      RUPQ and
  FenwickTree purq;                                    // one
      PURQ
public:
  RURQ(int m) : rupq(RUPQ(m)), purq(FenwickTree(m)) {} //
        initialization
  void range_update(int ui, int uj, ll v) {
    rupq.range_update(ui, uj, v);                      // [ui,
        ui+1, .., uj] +v
    purq.update(ui, v*(ui-1));                         // -(ui
        -1)*v before ui
    purq.update(uj+1, -v*uj);                          // +(uj-
        ui+1)*v after uj
  }
  ll rsq(int j) {
    return rupq.point_query(j)*j -                     //
        optimistic calculation
           purq.rsq(j);                                //
```

```
                 cancelation factor
  }
  ll rsq(int i, int j) { return rsq(j) - rsq(i-1); } //
      standard
};

int main() {
  vll f = {0,0,1,0,1,2,3,2,1,1,0};                  // index
       0 is always 0
  FenwickTree ft(f);
  cout << "select:" << ft.select(5);
  printf("%lld\n", ft.rsq(1, 6)); // 7 => ft[6]+ft[4] =
      5+2 = 7
  printf("%d\n", ft.select(7)); // index 6, rsq(1, 6) ==
      7, which is >= 7
  ft.update(5, 1); // update demo
  printf("%lld\n", ft.rsq(1, 10)); // now 12
  printf("=====\n");
  RUPQ rupq(10);
  RURQ rurq(10);
  rupq.range_update(2, 9, 7); // indices in [2, 3, .., 9]
       updated by +7
  rupq.range_update(6, 7, 3); // indices 6&7 are further
      updated by +3 (10)
  rupq.point_query(6);

  rurq.range_update(2, 9, 7); // same as rupq above

  rurq.range_update(6, 7, 3); // same as rupq above
  // idx = 0 (unused) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
       |10
  // val = -          | 0 | 7 | 7 | 7 | 7 |10 |10 | 7 | 7
       | 0
  for (int i = 1; i <= 10; i++)
    printf("%d -> %lld\n", i, rupq.point_query(i));
  printf("RSQ(1, 10) = %lld\n", rurq.rsq(1, 10)); // 62
  printf("RSQ(6, 7) = %lld\n", rurq.rsq(6, 7)); // 20
  return 0;
}
```

3.6 Segment Tree - cpbook

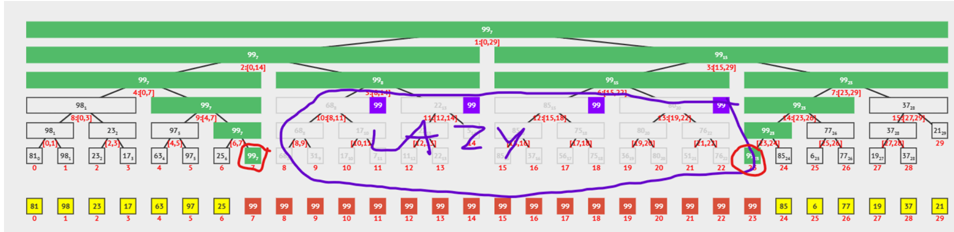

Figure 6: Segment Tree con Lazy Propagation

UPDATE(l=7,r=23,val=99)
Lo que esta en morado no fue realmente actualizado en A pero si sus rangos(nodos superiores), los nodos encerrados en rojo tuvieron que ser actualizados ya que ningun nodo superior cubre su rango la gran ventaja fue que actualizamos 6 nodos en total en vez de los 16 nodos hoja

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

typedef vector<int> vi;

class SegmentTree {                                      // OOP
     style
private:
  int n;                                                 // n = (
       int)A.size()
  vi A, st, lazy;                                        // the
       arrays

  int l(int p) { return  p<<1; }                         // go to
       left child
  int r(int p) { return (p<<1)+1; }                      // go to
       right child

  int conquer(int a, int b) {
    if (a == -1) return b;                               //
         corner case
    if (b == -1) return a;
    return min(a, b);                                    // RMQ
  }

  void build(int p, int L, int R) {                      // O(n)
    if (L == R)
      st[p] = A[L];                                      // base
           case
    else {
      int m = (L+R)/2;
      build(l(p), L  , m);
```

```
      build(r(p), m+1, R);
      st[p] = conquer(st[l(p)], st[r(p)]);
    }
  }

  void propagate(int p, int L, int R) {
    if (lazy[p] != -1) {                              // has a
        lazy flag
      st[p] = lazy[p];                                // [L..R
        ] has same value
      if (L != R)                                     // not a
         leaf
        lazy[l(p)] = lazy[r(p)] = lazy[p];            //
          propagate downwards
      else                                            // L ==
         R, a single index
        A[L] = lazy[p];                               // time
          to update this
      lazy[p] = -1;                                   // erase
         lazy flag
    }
  }

  int RMQ(int p, int L, int R, int i, int j) {   // O(log
      n)
    propagate(p, L, R);                               // lazy
       propagation
    if (i > j)
      return -1;                             // infeasible
    if ((L >= i) && (R <= j))
      return st[p];      // found the segment
    int m = (L+R)/2;
    int left = RMQ(l(p), L  , m, i, min(m, j));
    int right = RMQ(r(p), m+1, R, max(i, m+1), j);
    return conquer(left,right);
  }

  void update(int p, int L, int R, int i, int j, int val)
      { // O(log n)
    propagate(p, L, R);                               // lazy
       propagation
    if (i > j) return;
    if ((L >= i) && (R <= j)) {                       // found
        the segment
      lazy[p] = val;                                  //
         update this
      propagate(p, L, R);                             // lazy
         propagation
    }
    else {
      int m = (L+R)/2;
      update(l(p), L  , m, i          , min(m, j), val);
      update(r(p), m+1, R, max(i, m+1), j        , val);
      int lsubtree = (lazy[l(p)] != -1) ? lazy[l(p)] : st
         [l(p)];
      int rsubtree = (lazy[r(p)] != -1) ? lazy[r(p)] : st
         [r(p)];
      st[p] = conquer(lsubtree, rsubtree);
    }
  }

public:
  SegmentTree(int sz) : n(sz), A(n), st(4*n), lazy(4*n,
      -1) {}

  SegmentTree(const vi &initialA) : SegmentTree((int)
      initialA.size()) {
    A = initialA;
    build(1, 0, n-1);
    true;
  }

  void update(int i, int j, int val) { update(1, 0, n-1,
      i, j, val);  }

  int RMQ(int i, int j) { return RMQ(1, 0, n-1, i, j); }
};

int main() {
  vi A = {18, 17, 13, 19, 15, 11, 20, 99};       // make
      n a power of 2
  SegmentTree st(A);
  st.update(4,7, 2);


  st.RMQ(1,2);
  printf("              idx    0, 1, 2, 3, 4, 5, 6, 7\n")
      ;
  printf("              A is {18,17,13,19,15,11,20,oo}\n"
      );
  printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3));       // 13
  printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7));       // 11
  printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4));       // 15

  st.update(5, 5, 77);                            //
      update A[5] to 77
  printf("              idx    0, 1, 2, 3, 4, 5, 6, 7\n")
      ;
  printf("Now, modify A into {18,17,13,19,15,77,20,oo}\n"
      );
  printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3));       //
      remains 13
  printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7));       // now
      15
  printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4));       //
      remains 15

  st.update(0, 3, 30);                            //
      update A[0..3] to 30
  printf("              idx    0, 1, 2, 3, 4, 5, 6, 7\n")
      ;
  printf("Now, modify A into {30,30,30,30,15,77,20,oo}\n"
      );
```

```
  printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3));       // now
      30
  printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7));       //
      remains 15
  printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4));       //
      remains 15

  st.update(3, 3, 7);                             //
      update A[3] to 7
  printf("              idx    0, 1, 2, 3, 4, 5, 6, 7\n")
      ;
  printf("Now, modify A into {30,30,30, 7,15,77,20,oo}\n"
      );
  printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3));       // now 7
  printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7));       //
      remains 15
  printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4));       // now 7

  return 0;
}
```

3.7 Order Statistics Tree

3.7.1 Quick Selct - cpbook

Ranking(v) = posicion del elemento v si el arreglo estuviese ordenado.

```
int Partition(int A[], int l, int r) {
  int p = A[l];                                   // p is
      the pivot
  int m = l;                                      // S1
      and S2 are empty
  for (int k = l+1; k <= r; ++k) {                //
      explore unknown region
    if (A[k] < p) {                               // case
        2
      ++m;
      swap(A[k], A[m]);
    } // notice that we do nothing in case 1: a[k] >= p
  }
  swap(A[l], A[m]);                               // swap
      pivot with a[m]
  return m;                                       //
      return pivot index
}

int RandPartition(int A[], int l, int r) {
  int p = l + rand() % (r-l+1);                   //
      select a random pivot
  swap(A[l], A[p]);                               // swap
      A[p] with A[l]
  return Partition(A, l, r);
}

int QuickSelect(int A[], int l, int r, int k) {   //
    expected O(n)
  if (l == r) return A[l];
  int q = RandPartition(A, l, r);                 // O(n)
  if (q+1 == k)
    return A[q];
  else if (q+1 > k)
    return QuickSelect(A, l, q-1, k);
  else
    return QuickSelect(A, q+1, r, k);
}

int main() {
  int A[] = { 2, 8, 7, 1, 5, 4, 6, 3 };

  nth_element(A,A+4, A+8);
  printf("%d\n", A[4]);
  //output: 5
  for(auto i:A)
    cout << i << ",";
  //output: [3,2,1,4,5,7,6,8]

  return 0;
}
```

3.8 Ordered Statistics Tree - bits/extc++.h

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

#include <bits/extc++.h>                          // pbds
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
             tree_order_statistics_node_update> ost;

int main() {
  int n = 9;
  int A[] = { 2, 4, 7,10,15,23,50,65,71};         // as in
      Chapter 2
  ost tree;
  for (int i = 0; i < n; ++i)                     // O(n
      log n)
    tree.insert(A[i]);
  // O(log n) select
  cout << *tree.find_by_order(0) << "\n";         // 1-
      smallest = 2
  cout << *tree.find_by_order(n-1) << "\n";       // 9-
      smallest/largest = 71
  cout << *tree.find_by_order(4) << "\n";         // 5-
      smallest = 15
  // O(log n) rank
```

```
    cout << tree.order_of_key(2) << "\n";           // index 0 (rank 1)
    cout << tree.order_of_key(71) << "\n";          // index 8 (rank 9)
    cout << tree.order_of_key(15) << "\n";          // index 4 (rank 5)

    return 0;
}
```

3.9 Priority Queue

```
struct Node {
  int id;
};
// si la funcion devuelve true, a tiene menor prioridad que b
struct cmp {
  bool operator()(const Node& a, const Node& b) const {
    return a.id < b.id;
  }
};

priority_queue<Node, vector<Node>, cmp> pq;// max-heap por id

int main(){
  priority_queue<int, vector<int>> pq_min;// por defecto es max-heap
  priority_queue<int, vector<int>, greater<int>> pq_min;// min-heap
  // Create O(n)
  vector<int> d = {12,3,4,3,3,5,34,343,5325,235,23452,3532};
  priority_queue<int> a (d.begin(), d.end());
  // Create O(n * log n)
  vector<int> d = {12,3,4,3,3,5,34,343,5325,235,23452,3532};
  priority_queue<int> a;
  for(auto i: d)
      a.push(i);
  return 0;
}
```

3.10 Trie-Recursivo-Lucas

```
struct Node
{
  Node* sig_cero = NULL;Node* sig_uno = NULL;
  int cont_cero = 0, cont_uno = 0;
};

class Trie
{
private:
  Node* r;
  void Add(Node*& node, int i)
  {
    if(i > 63) return;
    if(node == NULL) node = new Node();
    if(last_num[i] == '0')
    {
      node->cont_cero++;
      Add(node->sig_cero, i+1);
    }
    else
    {
      node->cont_uno++;
      Add(node->sig_uno, i+1);
    }
  }

  void Delete(Node*& node, int i)
  {
    if(i > 63) return;
    if(node == NULL) node = new Node();
    if(last_num[i] == '0')
    {
      node->cont_cero--;
      Delete(node->sig_cero, i+1);
    }
    else
    {
      node->cont_uno--;
      Delete(node->sig_uno, i+1);
    }
  }

  void Max_xor(Node*& node, int i)
  {
    if(i > 63)
    return;
    if(node == NULL)
    return;
    if(last_num[i] == '0')
    {
      if(node->cont_uno > 0)
      {
        ans.pb('1');
        Max_xor(node->sig_uno, i+1);
      }
      else
      {
        ans.pb('0');
        Max_xor(node->sig_cero, i+1);
      }
    }
    else
    {
```

```
      if(node->cont_cero > 0)
      {
        ans.pb('0');
        Max_xor(node->sig_cero, i+1);
      }
      else
      {
        ans.pb('1');
        Max_xor(node->sig_uno, i+1);
      }
    }
  }
public:
  string ans;
  string last_num;
  Trie()
  {
    r = new Node();
  }

  void add(int x)
  {
    this->last_num = numtobin(x);//128 bits
    Add(r, 0);
  }
  void deletee(int x)
  {
    this->last_num = numtobin(x);//128 bits
    Delete(r, 0);
  }
  void max_xor(int x)
  {
    this->last_num = numtobin(x);//128 bits
    ans.clear();
    Max_xor(r, 0);
  }
};
```

3.11 Trie-Iterativo-Mati

```
struct nodo {
  nodo* hijos[2];
  int cont;

  nodo() {
    hijos[0] = hijos[1] = nullptr;
    cont = 0;
  }
};

class ArbolBin {
private:

nodo* raiz;

void borrarNodo(nodo* n) {
  if(!n) return;

  borrarNodo(n->hijos[0]);
  borrarNodo(n->hijos[1]);
  delete n;
}

public:

ArbolBin() {
  raiz = new nodo();
}

~ArbolBin() {
  borrarNodo(raiz);
}


void insertar(int x) {
  nodo* nodoActual = raiz;
  int bitActual;
  for(int i = MB - 1; i >= 0; i--) {
    bitActual = (x>>i) & 1;

    if(nodoActual->hijos[bitActual] == nullptr) {
      nodoActual->hijos[bitActual] = new nodo();
    }

    nodoActual = nodoActual->hijos[bitActual];

    nodoActual->cont++;
  }
}

void borrar(int x) {
  nodo* nodoActual = raiz;
  int bitActual;
  for(int i = MB - 1; i >= 0; i--) {
    bitActual = (x>>i) & 1;

    nodoActual = nodoActual->hijos[bitActual];

    nodoActual->cont--;
  }
}

int consulta(int x) {
  nodo* nodoActual = raiz;
  int bitDeseado, res = 0;
  for(int i = MB - 1; i >= 0; i--) {
    bitDeseado = ((x>>i) & 1)^1;

    if(nodoActual->hijos[bitDeseado] != nullptr and
        nodoActual->hijos[bitDeseado]->cont > 0) {
      res = res|(1<<i);
    }
    else {
```

```
      bitDeseado = bitDeseado^1;
    }
    nodoActual = nodoActual->hijos[bitDeseado];
  }
  return res;
}
};
```

3.12 Suffix Tree

```
const int N=1000000,    // maximum possible number of nodes in suffix tree
    INF=1000000000; // infinity constant
string a;       // input string for which the suffix tree is being built
int t[N][26],   // array of transitions (state, letter)
    l[N],   // left...
    r[N],   // ...and right boundaries of the substring of a which correspond to incoming edge
    p[N],   // parent of the node
    s[N],   // suffix link
    tv,     // the node of the current suffix (if we're mid-edge, the lower node of the edge)
    tp,     // position in the string which corresponds to the position on the edge (between l[tv] and r[tv], inclusive)
    ts,     // the number of nodes
    la;     // the current character in the string

void ukkadd(int c) { // add character s to the tree
    suff:;      // we'll return here after each transition to the suffix (and will add character again)
    if (r[tv]<tp) { // check whether we're still within the boundaries of the current edge
        // if we're not, find the next edge. If it doesn't exist, create a leaf and add it to the tree
        if (t[tv][c]==-1) {t[tv][c]=ts;l[ts]=la;p[ts++]=tv;tv=s[tv];tp=r[tv]+1;goto suff;}
        tv=t[tv][c];tp=l[tv];
    } // otherwise just proceed to the next edge
    if (tp==-1 || c==a[tp]-'a')
        tp++; // if the letter on the edge equal c, go down that edge
    else {
        // otherwise split the edge in two with middle in node ts
        l[ts]=l[tv];r[ts]=tp-1;p[ts]=p[tv];t[ts][a[tp]-'a']=tv;
        // add leaf ts+1. It corresponds to transition through c.
        t[ts][c]=ts+1;l[ts+1]=la;p[ts+1]=ts;
        // update info for the current node - remember to mark ts as parent of tv
        l[tv]=tp;p[tv]=ts;t[p[ts]][a[l[ts]]-'a']=ts;ts+=2;
        // prepare for descent
        // tp will mark where are we in the current suffix
        tv=s[p[ts-2]];tp=l[ts-2];
        // while the current suffix is not over, descend
        while (tp<=r[ts-2]) {tv=t[tv][a[tp]-'a'];tp+=r[tv]-l[tv]+1;}
        // if we're in a node, add a suffix link to it, otherwise add the link to ts
        // (we'll create ts on next iteration).
        if (tp==r[ts-2]+1) s[ts-2]=tv; else s[ts-2]=ts;
        // add tp to the new edge and return to add letter to suffix
        tp=r[tv]-(tp-r[ts-2])+2;goto suff;
    }
}

void build() {
    ts=2;
    tv=0;
    tp=0;
    fill(r,r+N,(int)a.size()-1);
    // initialize data for the root of the tree
    s[0]=1;
    l[0]=-1;
    r[0]=-1;
    l[1]=-1;
    r[1]=-1;
    memset (t, -1, sizeof t);
    fill(t[1],t[1]+26,0);
    // add the text to the tree, letter by letter
    for (la=0; la<(int)a.size(); ++la)
        ukkadd (a[la]-'a');
}
```

3.13 Custom Hash

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

    size_t operator()(ll x) const {
        static const ll FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
```

```
        }
    };
    unordered_map<ll,int, custom_hash> mapa;
```

4 Math

4.1 Prime Numbers 1-2000

```
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601
607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733
739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997 1009 1013
1019 1021 1031 1033 1039 1049 1051 1061 1063 1069
1087 1091 1093 1097 1103 1109 1117 1123 1129 1151
1153 1163 1167 1181 1187 1193 1201 1213 1217 1223
1229 1231 1237 1249 1259 1277 1279 1283 1289 1291
1297 1301 1303 1307 1319 1321 1327 1361 1367 1373
1381 1399 1409 1423 1427 1429 1433 1439 1447 1451
1453 1459 1471 1481 1483 1487 1489 1493 1499 1511
1523 1531 1543 1549 1553 1559 1567 1571 1579 1583
1597 1601 1607 1609 1613 1619 1621 1627 1637 1657
1663 1667 1669 1693 1699 1709 1721 1723 1733
1741 1747 1753 1759 1777 1783 1787 1789 1801 1811
1823 1831 1847 1861 1867 1871 1873 1877 1879 1889
1901 1907 1913 1931 1933 1949 1951 1973 1979 1987
```

```
 970'997 971'483 921'281'269 999'279'733
 1'000'000'009 1'000'000'021 1'000'000'409
 1'005'012'527
```

4.2 Serie de Fibonacci 0-20

```
Def: F(0)=0  ,  F(1)=1  ,   F(n)=F(n-1)+F(n-2)
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
F(10) = 55
F(11) = 89
F(12) = 144
F(13) = 233
F(14) = 377
F(15) = 610
F(16) = 987
F(17) = 1597
F(18) = 2584
F(19) = 4181
F(20) = 6765
```

4.3 Factorial 0-20

```
Def: n!=n(n-1)!
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

```
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
```

4.4 Numeros Triangulares 1-20

```
Def: T(n)=n(n+1)/2

T(1) = 1
T(2) = 3
T(3) = 6
T(4) = 10
T(5) = 15
T(6) = 21
T(7) = 28
T(8) = 36
T(9) = 45
T(10) = 55
T(11) = 66
T(12) = 78
T(13) = 91
T(14) = 105
T(15) = 120
T(16) = 136
T(17) = 153
T(18) = 171
T(19) = 190
T(20) = 210
```

4.5 Numeros Cuadrados 1-20

```
Def: Q(n)=n^2

Q(1) = 1
Q(2) = 4
Q(3) = 9
Q(4) = 16
Q(5) = 25
Q(6) = 36
Q(7) = 49
Q(8) = 64
Q(9) = 81
Q(10) = 100
Q(11) = 121
Q(12) = 144
Q(13) = 169
Q(14) = 196
Q(15) = 225
Q(16) = 256
Q(17) = 289
Q(18) = 324
Q(19) = 361
Q(20) = 400
```

4.6 Simple Sieve of Eratosthenes O(n*log(log(n))) - con n=1e7 1.25 MB

```
#define tam 1e7
vector < bool > criba(tam , true);

void criba_function()
{
  criba[0]=false;
  criba[1]=false;
  //( i*i < tam) equivalente a (i <= sqrt(tam))
  for(int i = 2; i*i <= tam ; i++)
  {
    if(!criba[i]) continue;
    for(int j = 2; i*j <= tam ; j++)
      criba[i * j] = false;
  }
}
```

4.7 Smallest Prime Factor AND Sieve of Eratosthenes O(n) - con n=1e7 45 MB

```
// O(n)
// pr contains prime numbers
// lp[i] == i if i is prime
// else lp[i] is minimum prime factor of i
const int nax = 1e7;
int lp[nax+1];//because lp is an array nax have to be
   less than 1e7 or change to a vector(nax+1,0)
vector<int> pr; // It can be sped up if change for an
   array

void sieve(){
  form(i,2,nax){
    if (lp[i] == 0) {
      lp[i] = i; pr.pb(i);
    }
    for (int j=0, mult= i*pr[j]; j<sz(pr) && pr[j]<=lp[i]
        && mult<nax; ++j, mult= i*pr[j])
      lp[mult] = pr[j];
  }
}
```

4.8 Smallest Prime Factor Piton++


```
// O(n)
// pr contains prime numbers
// lp[i] == i if i is prime
// else lp[i] is minimum prime factor of i
const int nax = 1e7;
int lp[nax+1];//because lp is an array nax have to be
   less than 1e7 or change to a vector(nax+1,0)
vector<int> pr; // It can be sped up if change for an
   array

void sieve(){
  form(i,2,nax){
    if (lp[i] == 0) {
      lp[i] = i; pr.pb(i);
    }
    for (int j=0, mult= i*pr[j]; j<sz(pr) && pr[j]<=lp[i]
        && mult<nax; ++j, mult= i*pr[j])
      lp[mult] = pr[j];
  }
}
```


4.9 Combinatorics

4.9.1 Next permutation

```
int main() {
  vector<int> perm = {1, 2, 3};
  sort(perm.begin(), perm.end());
  do {
    for (int x : perm)
      cout << x << ' ';
    cout << '\n';
  } while (next_permutation(perm.begin(), perm.end()));

  int arr[] = {2,3,4,5,1,0};
  sort(arr,arr+6);// arr+CANTIDAD DE ELEMENTOS
  do
  {
    for (int x : perm)
        cout << x << ' ';
      cout << '\n';
  } while (next_permutation(arr,arr+6));
  return 0;
}
```

5 Dynamic Programming

6 Otros

6.1 Binary Search

```
vi acu(int(1e5));
int c(int l, int r)
{
  if(l > 0) return acu[r] - acu[l-1];
  return acu[r];
}

int t ;
int bs(int l, int r)
{
  int i = l, j = r;
  int n , mitad;
  while (i != j)
  {
    n = j - i;
    mitad = i + n/2;
    if(c(l,mitad) == t) return mitad - l + 1;
    if(c(l,mitad) > t) j = mitad;
    else i = mitad + 1;
  }
  return c(l,i) <= t ? i - l + 1 : (c(l,i - 1) ? i - l :
     - 1) ;
}

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

  icin(n)
  cin >> t;
```

```
  vi nums(n);
  form(i,0,n) cin >> nums[i];

  acu[0] = nums[0];
  form(i,1,n) acu[i] = acu[i-1] + nums[i];
  int maxi = 0;
  form(i,0,n)
    maxi = max(maxi , bs(i, n-1));
  cout << maxi << endl;
  return 0;
}
```

Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL
1	00000001	001	01	SOH
2	00000010	002	02	STX
3	00000011	003	03	ETX
4	00000100	004	04	EOT
5	00000101	005	05	ENQ
6	00000110	006	06	ACK
7	00000111	007	07	BEL
8	00001000	010	08	BS
9	00001001	011	09	HT
10	00001010	012	0A	LF
11	00001011	013	0B	VT
12	00001100	014	0C	FF
13	00001101	015	0D	CR
14	00001110	016	0E	SO
15	00001111	017	0F	SI
16	00010000	020	10	DLE
17	00010001	021	11	DC1
18	00010010	022	12	DC2
19	00010011	023	13	DC3
20	00010100	024	14	DC4
21	00010101	025	15	NAK
22	00010110	026	16	SYN
23	00010111	027	17	ETB
24	00011000	030	18	CAN
25	00011001	031	19	EM
26	00011010	032	1A	SUB
27	00011011	033	1B	ESC
28	00011100	034	1C	FS
29	00011101	035	1D	GS
30	00011110	036	1E	RS
31	00011111	037	1F	US

Decimal	Binary	Octal	Hex	ASCII
32	00100000	040	20	SP
33	00100001	041	21	!
34	00100010	042	22	"
35	00100011	043	23	#
36	00100100	044	24	$
37	00100101	045	25	%
38	00100110	046	26	&
39	00100111	047	27	'
40	00101000	050	28	(
41	00101001	051	29	)
42	00101010	052	2A	*
43	00101011	053	2B	+
44	00101100	054	2C	,
45	00101101	055	2D	-
46	00101110	056	2E	.
47	00101111	057	2F	/
48	00110000	060	30	0
49	00110001	061	31	1
50	00110010	062	32	2
51	00110011	063	33	3
52	00110100	064	34	4
53	00110101	065	35	5
54	00110110	066	36	6
55	00110111	067	37	7
56	00111000	070	38	8
57	00111001	071	39	9
58	00111010	072	3A	:
59	00111011	073	3B	;
60	00111100	074	3C	<
61	00111101	075	3D	=
62	00111110	076	3E	>
63	00111111	077	3F	?

Decimal	Binary	Octal	Hex	ASCII
64	01000000	100	40	@
65	01000001	101	41	A
66	01000010	102	42	B
67	01000011	103	43	C
68	01000100	104	44	D
69	01000101	105	45	E
70	01000110	106	46	F
71	01000111	107	47	G
72	01001000	110	48	H
73	01001001	111	49	I
74	01001010	112	4A	J
75	01001011	113	4B	K
76	01001100	114	4C	L
77	01001101	115	4D	M
78	01001110	116	4E	N
79	01001111	117	4F	O
80	01010000	120	50	P
81	01010001	121	51	Q
82	01010010	122	52	R
83	01010011	123	53	S
84	01010100	124	54	T
85	01010101	125	55	U
86	01010110	126	56	V
87	01010111	127	57	W
88	01011000	130	58	X
89	01011001	131	59	Y
90	01011010	132	5A	Z
91	01011011	133	5B	[
92	01011100	134	5C	\
93	01011101	135	5D	]
94	01011110	136	5E	^
95	01011111	137	5F	_

Decimal	Binary	Octal	Hex	ASCII
96	01100000	140	60	`
97	01100001	141	61	a
98	01100010	142	62	b
99	01100011	143	63	c
100	01100100	144	64	d
101	01100101	145	65	e
102	01100110	146	66	f
103	01100111	147	67	g
104	01101000	150	68	h
105	01101001	151	69	i
106	01101010	152	6A	j
107	01101011	153	6B	k
108	01101100	154	6C	l
109	01101101	155	6D	m
110	01101110	156	6E	n
111	01101111	157	6F	o
112	01110000	160	70	p
113	01110001	161	71	q
114	01110010	162	72	r
115	01110011	163	73	s
116	01110100	164	74	t
117	01110101	165	75	u
118	01110110	166	76	v
119	01110111	167	77	w
120	01111000	170	78	x
121	01111001	171	79	y
122	01111010	172	7A	z
123	01111011	173	7B	{
124	01111100	174	7C	|
125	01111101	175	7D	}
126	01111110	176	7E	~
127	01111111	177	7F	DEL

Figure 7: Ascii code

Tipo	Tam. Bits	Dígitos de precisión	Rango	
			Min	Max
Bool	8	0	0	1
Char	8	2	-128	127
Signed char	8	2	-128	127
unsigned char	8	2	0	255
short int	16	4	-32,768	32,767
unsigned short int	16	4	0	65,535
Int	32	9	-2,147,483,648	2,147,483,647
unsigned int	32	9	0	4,294,967,295
long int	32	9	-2,147,483,648	2,147,483,647
unsigned long int	32	9	0	4,294,967,295
long long int	64	18	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long int	64	18	0	18,446,744,073,709,551,615
Float	32	6	1.17549e-38	3.40282e+38
Double	64	15	2.22507e-308	1.79769e+308

Figure 8: Data types limits

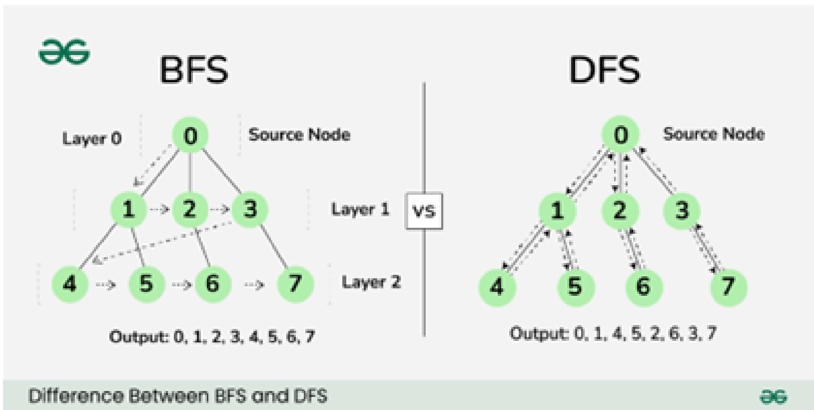

Figure 9: DFS y BFS

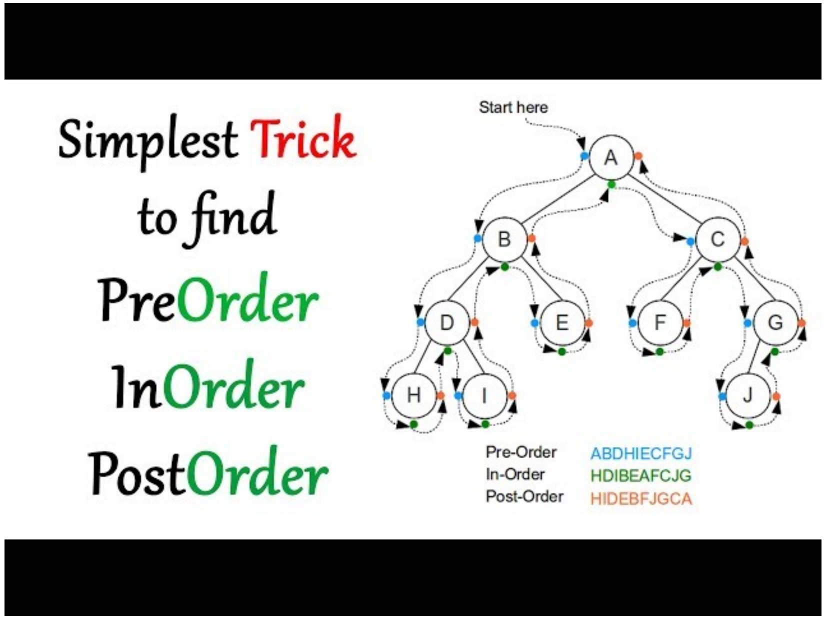

Figure 10: Pre-order, In-order y Post-order

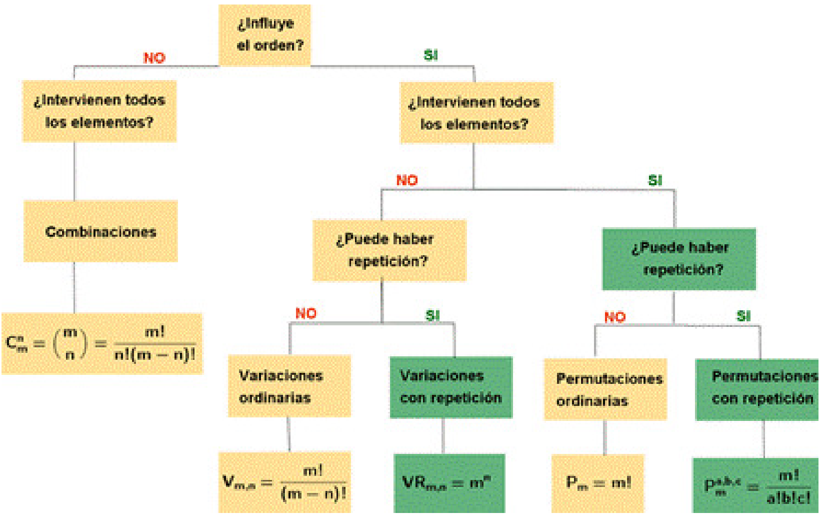

Figure 11: Combinatorics

The following are true involving modular arithmetic:

1. $(a + b) \% m = ((a \% m) + (b \% m)) \% m$
   Example: `(15 + 29) % 8`
   `= ((15 % 8) + (29 % 8)) % 8 = (7 + 5) % 8 = 4`

2. $(a - b) \% m = ((a \% m) - (b \% m)) \% m$
   Example: `(37 - 15) % 6`
   `= ((37 % 6) - (15 % 6)) % 6 = (1 - 3) % 6 = -2` or `4`

3. $(a \times b) \% m = ((a \% m) \times (b \% m)) \% m$
   Example: `(23 × 12) % 5`
   `= ((23 % 5) × (12 % 5)) % 5 = (3 × 2) % 5 = 1`

Figure 12: Modulo properties

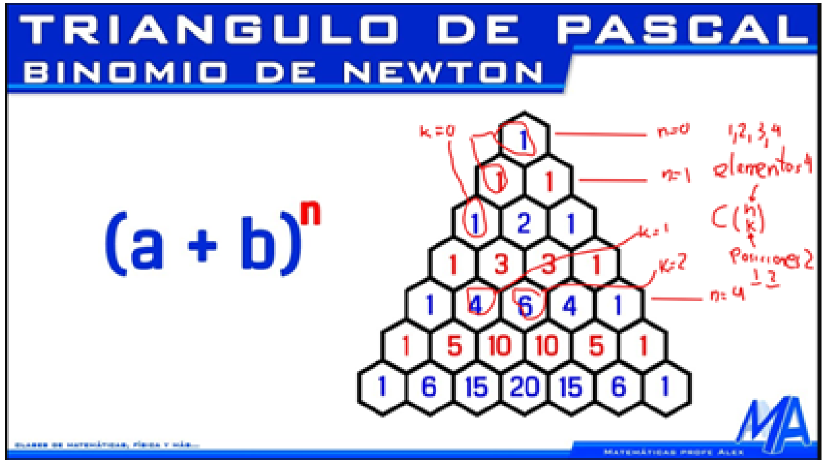

Figure 13: Pascal's triangle