# Manual de CP

ErrorByNight

## Índice


**1. Plantilla** **1**

**2. Teoria de numeros** **3**
2.1. Criba de Eratóstenes . . . . . 3
2.2. Big Mod . . . . . 3

**3. Grafos** **3**
3.1. DFS . . . . . 3
3.2. BFS . . . . . 3
3.3. Topological Sort . . . . . 4
3.4. Dijkstra Algorithm . . . . . 4
3.5. Kruskal . . . . . 4
3.6. Puntos de articulacion . . . . . 5

**4. Estructuras** **5**
4.1. Trie Tree . . . . . 5
4.2. Segment Tree . . . . . 5
4.2.1. Iterativo . . . . . 5
4.2.2. Recursivo . . . . . 6
4.2.3. Persistente . . . . . 6


## 1. Plantilla

```
/*
   TEMPLATE
*/

#include <bits/stdc++.h>
#define D(x) cout << #x << ": " << x << endl;
#define CYN(x) cout << (x ? "YES" : "NO" ) <<endl;
#define forn(i,n) for(int i=0; i< (int)n; i++)
#define for1(i,n) for(int i=1; i<= (int)n; i++)
#define all(v) v.begin(),v.end()
#define precision(x) cout<< setprecision(20)<< fixed
#define cin_pro ios::sync_with_stdio(0); cin.tie(NULL); cout.tie(NULL)
#define pb push_back
#define F first
#define S second
#define pf push_front
#define mp make_pair
#define rall(v) v.rbegin(), v.rend()
#define cases(t) while(t--)
#define rforn(i,n) for(int i = n - 1; i >= 0; i--)
#define rfor1(i,n) for(int i = n; i >= 1; i--)
#define foreach(it, v) for(auto it: v)
#define mem(v, val) memset(v, (val), sizeof(v))
#define inf (int) 1e9
#define pi 3.14159265358979323846264338327950

#define vi vector<int>
#define pii pair<int,int>
#define vii vector<pii>
#define vvi vector<vi>
#define mpii map<int,int>
#define umpii unordered_map<int,int>
#define seti set<int>
#define pqi priority_queue<int>

//Operaciones | Sumatorias | Otros
#define sumn(n) n*(n+1)/2
```

```cpp
#define sumevens(n) n*(n+1)
#define sumodds(n) n*n
#define sumsquares(n) (n*(n+1)*(2*n+1))/6
#define sumcubes(n) sumn(n)*sumn(n)

int dr[] = {1,-1,0, 0,1,-1,-1, 1};
int dc[] = {0, 0,1,-1,1, 1,-1,-1};

using namespace std;

typedef long long ll;

template <typename T> void amax(T &a, const T &b){ if( a < b) a = b; }
template <typename T> void amin(T &a, const T &b){ if( b < a) a = b; }
template <typename T> T gcd(T a, T b){
    if(a==0 || b==0)return max(a,b);
    else return gcd(b,a%b);
}

template <typename T> inline void prefix_sum(T arr, T& res){
   if(arr.size() > 0) res[0]=arr[0];
   for(int i = 1; i < arr.size(); i++) res[i] += res[i-1] + arr[i];
}

template <typename T> inline void sufix_sum(T arr, T& res){
   if(arr.size() > 0) res[arr.size()-1]=arr[arr.size()-1];
   for(int i = arr.size()-2; i >= 0; i--) res[i] += res[i+1] + arr[i];
}

void read_fast(){
   cin_pro;
   //#ifdef ONLINE_JUDGE
   #ifdef LOCAL
        freopen("input.txt", "r", stdin);
   #else
        #define endl '\n'
    #endif
}

/*
   END OF TEMPLATE
*/

void solve(){

}

int main(){
   read_fast();
   int t = 1;
   cases(t){
      solve();
   }
   return 0;
}
```

# 2. Teoria de numeros

## 2.1. Criba de Eratóstenes

```
const int SIZE = 1000000;
//criba[i] = false si i es primo
bool criba[SIZE+1];
void buildCriba(){
    memset(criba, false, sizeof(criba));
    criba[0] = criba[1] = true;
    for (int i=4; i<=SIZE; i += 2){
        criba[i] = true;
    }
    for (int i=3; i*i<=SIZE; i += 2){
    if (!criba[i]){
        for (int j=i*i; j<=SIZE; j += i){
            criba[j] = true;
        }
    }
    }
}
```

## 2.2. Big Mod

```
//retorna (b^p)mod(m)
int bigmod(int b, int p, int m){
    int mask = 1;
    int pow2 = b % m;
    int r = 1;
    while (mask){
        if (p & mask) r = (r * pow2) % m;
        pow2 = (pow2 * pow2) % m;
        mask <<= 1;
    }
    return r;
}
```

# 3. Grafos

## 3.1. DFS

Complejidad: O(n+m) donde n es el numero de nodos y m es el numero de aristas

```
vector <int> g[MAXN];    // La lista de adyacencia
int color[MAXN];         // El arreglo de visitados
enum {WHITE, GRAY, BLACK}; // WHITE = 1, GRAY = 2, BLACK = 3

// Visita el nodo u y todos sus vecinos empezando por
// los mas profundos
void dfs(int u){
   color[u] = GRAY; // Marcar el nodo como semi-visitado
   for (int i = 0; i < g[u].size(); ++i){
     int v = g[u][i];
     if (color[v] == WHITE) dfs(v); // Visitar los vecinos
   }
   color[u] = BLACK; // Marcar el nodo como visitado
}

// Llama la funcion dfs para los nodos 0 a n-1
void call_dfs(int n){
   for (int u = 0; u < n; ++u) color[u] = WHITE;
   for (int u = 0; u < n; ++u)
     if (color[u] == WHITE) dfs(u);
}
```

## 3.2. BFS

Complejidad: O(n+m) donde n es el numero de nodos y m es el numero de aristas

```
vector <int> g[MAXN]; // La lista de adyacencia
int d[MAXN];          // Distancia de la fuente a cada nodo

void bfs(int s, int n){ // s = fuente, n = numero de nodos
   for (int i = 0; i < n; ++i) d[i] = -1;

   queue <int> q;
   q.push(s);
   d[s] = 0;
   while (q.size() > 0){
      int cur = q.front();
      q.pop();
      for (int i = 0; i < g[cur].size(); ++i){
         int next = g[cur][i];
         if (d[next] == -1){
            d[next] = d[cur] + 1;
            q.push(next);
         }
      }
   }
}
```

## 3.3. Topological Sort

Complejidad: O(n+m) donde n es el numero de nodos y m es el numero de aristas

```
int n; // numero de vertices
vector<vector<int>> adj; // Lista de adyacencias del grafo
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
    reverse(ans.begin(), ans.end());
}
```

## 3.4. Dijkstra Algorithm

Complejidad: $O(n^2 + m)$ donde n es el numero de nodos y m es el numero de aristas

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}
```

## 3.5. Kruskal

Complejidad: $O(nlog(m))$ donde n es el numero de nodos y m es el numero de aristas

```
const int MAX = 1e6-1;
int root[MAX];
const int nodes = 4, edges = 5;
pair <long long, pair<int, int> > p[MAX];

int parent(int a)                                    //Buscar el
    padre del nodo
{
    while(root[a] != a)
    {
        root[a] = root[root[a]];
        a = root[a];
    }
    return a;
}

void union_find(int a, int b)                        //Verificar
    si dos nodos tienen una misma union
{
    int d = parent(a);
    int e = parent(b);
    root[d] = root[e];
}

long long kruskal()
```

```
{
    int a, b;
    long long cost, minCost = 0;
    for(int i = 0 ; i < edges ; ++i)
    {
        a = p[i].second.first;
        b = p[i].second.second;
        cost = p[i].first;
        if(parent(a) != parent(b))                    //Tener en
            cuenta si no genera un ciclo
        {
            minCost += cost;
            union_find(a, b);
        }
    }
    return minCost;
}
```

### 3.6. Puntos de articulacion

Complejidad: $O(nlog(m))$ donde n es el numero de nodos y m es el numero de aristas

```
int MAX = 1000000;
vector<int> desc[MAX]; // Inicializado arreglo
vector<vector<int>> g(MAX);
int cont = 1;
int DFS_PA (int node){
    desc[node] = ++cont;
    int menor = cont;
    for(auto u : g[node]){
        if desc[u] = 0:
            int min_m = DFS_PA(u);
            if(min_m < menor)
                menor = min_m;
            if (min_m >= desc[node])
                // n es un punto de articulacion
                cout<<n<<" ";
        else if(desc[u] < menor)
            menor = desc[u];
    }
    return menor;
}
```

# 4. Estructuras

## 4.1. Trie Tree

```
struct TrieNode{
    TrieNode* children[26];

    TrieNode(){
        for(int i=0;i<26;++i)
            children[i]=NULL;
    }
    void insert(string key){
        struct TrieNode* current = this;
        for(int i=0;i<key.length();++i){
            int index= key[i]-'a';
            if(!current->children[index]){
                current->children[index] = new TrieNode();
            }
            current = current->children[index];
        }
    }
    int search(string key){
        struct TrieNode* current = this;
        for(int i=0;i<key.length();++i){
            int index= key[i]-'a';
            if(!current->children[index])
                return 0;
            current = current->children[index];
        }
        return 1;
    }
};
```

## 4.2. Segment Tree

### 4.2.1. Iterativo

```
const int N = 1e5; // limit for array size
int n; // array size
int t[2 * N];

void build() { // build the tree
  for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void modify(int p, int value) { // set value at position p
  for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
```

```
}

int query(int l, int r) { // sum on interval [l, r)
  int res = 0;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) res += t[l++];
    if (r&1) res += t[--r];
  }
  return res;
}

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) scanf("%d", t + n + i);
  build();
  modify(0, 1);
  printf("%d\n", query(3, 11));
  return 0;
}
```

### 4.2.2. Recursivo

```
int n, t[4*MAXN];
template <typename T> inline T fun(T a, T b){ return a + b; }
void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = fun(t[v*2] , t[v*2+1]);
    }
}

int sum(int v, int tl, int tr, int l, int r) {
    if (l > r)
        return 0;
    if (l == tl && r == tr) {
        return t[v];
    }
    int tm = (tl + tr) / 2;
    return fun(sum(v*2, tl, tm, l, min(r, tm))
           , sum(v*2+1, tm+1, tr, max(l, tm+1), r));
}

void update(int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr) {
        t[v] = new_val;
    } else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update(v*2, tl, tm, pos, new_val);
        else
            update(v*2+1, tm+1, tr, pos, new_val);
        t[v] = fun(t[v*2] , t[v*2+1]);
    }
}
```

### 4.2.3. Persistente

```
// C++ program to implement persistent segment
// tree.
#include "bits/stdc++.h"
using namespace std;

#define MAXN 100

/* data type for individual
 * node in the segment tree */
struct node
{
    // stores sum of the elements in node
    int val;

    // pointer to left and right children
    node* left, *right;

    // required constructors........
    node() {}
    node(node* l, node* r, int v)
    {
        left = l;
        right = r;
        val = v;
    }
};

// input array
int arr[MAXN];

// root pointers for all versions
node* version[MAXN];
```

```
// Constructs Version-0
// Time Complexity : O(nlogn)
void build(node* n,int low,int high)
{
    if (low==high)
    {
        n->val = arr[low];
        return;
    }
    int mid = (low+high) / 2;
    n->left = new node(NULL, NULL, 0);
    n->right = new node(NULL, NULL, 0);
    build(n->left, low, mid);
    build(n->right, mid+1, high);
    n->val = n->left->val + n->right->val;
}

/**
 * Upgrades to new Version
 * @param prev : points to node of previous version
 * @param cur : points to node of current version
 * Time Complexity : O(logn)
 * Space Complexity : O(logn) */
void upgrade(node* prev, node* cur, int low, int high,
                                int idx, int value)
{
    if (idx > high or idx < low or low > high)
        return;

    if (low == high)
    {
        // modification in new version
        cur->val = value;
        return;
    }
    int mid = (low+high) / 2;
    if (idx <= mid)
    {
        // link to right child of previous version
        cur->right = prev->right;

        // create new node in current version
        cur->left = new node(NULL, NULL, 0);

        upgrade(prev->left,cur->left, low, mid, idx, value);
    }
    else
    {
        // link to left child of previous version
        cur->left = prev->left;

        // create new node for current version
        cur->right = new node(NULL, NULL, 0);

        upgrade(prev->right, cur->right, mid+1, high, idx, value);
    }

    // calculating data for current version
    // by combining previous version and current
    // modification
    cur->val = cur->left->val + cur->right->val;
}

int query(node* n, int low, int high, int l, int r)
{
    if (l > high or r < low or low > high)
      return 0;
    if (l <= low and high <= r)
      return n->val;
    int mid = (low+high) / 2;
    int p1 = query(n->left,low,mid,l,r);
    int p2 = query(n->right,mid+1,high,l,r);
    return p1+p2;
}

int main(int argc, char const *argv[])
{
    int A[] = {1,2,3,4,5};
    int n = sizeof(A)/sizeof(int);

    for (int i=0; i<n; i++)
      arr[i] = A[i];

    // creating Version-0
    node* root = new node(NULL, NULL, 0);
    build(root, 0, n-1);

    // storing root node for version-0
    version[0] = root;

    // upgrading to version-1
    version[1] = new node(NULL, NULL, 0);
    upgrade(version[0], version[1], 0, n-1, 4, 1);

    // upgrading to version-2
    version[2] = new node(NULL, NULL, 0);
    upgrade(version[1],version[2], 0, n-1, 2, 10);
```

```
    cout << "In version 1 , query(0,4) : ";
    cout << query(version[1], 0, n-1, 0, 4) << endl;

    cout << "In version 2 , query(3,4) : ";
    cout << query(version[2], 0, n-1, 3, 4) << endl;

    cout << "In version 0 , query(0,3) : ";
    cout << query(version[0], 0, n-1, 0, 3) << endl;
    return 0;
}
```