

Puntos importantes

Puntos a considerar

1. Trabajo en equipo

- **Comunicación:** En los concursos ICPC, el trabajo en equipo es fundamental ya que compartes un solo ordenador. Es crucial comunicarte de manera clara y efectiva con tus compañeros, para dividir las tareas y evitar malentendidos.
- **Roles definidos:** Usualmente, cada miembro del equipo tiene un rol asignado, ya sea programador principal, analista o buscador de errores. Tener roles bien definidos puede optimizar el tiempo.

2. Manejo del tiempo

- **Priorizar problemas:** Saber priorizar qué problemas intentar resolver primero es esencial. Los problemas más difíciles suelen ser un lastre si les dedicas demasiado tiempo al principio.
- **Tener una estrategia:** Dividir el tiempo de manera eficiente entre leer, analizar, codificar y depurar es esencial. A veces es mejor dejar un problema que está causando demasiados problemas y pasar a otro más fácil.

3. Capacidad para manejar la presión

- **Control emocional:** Los concursos ICPC pueden ser intensos, con muchas emociones. Saber mantener la calma y no dejar que la frustración o los nervios te afecten es vital.
- **Resiliencia:** Si te bloqueas con un problema, es importante no desmotivarse. A veces una pausa breve o un cambio de enfoque ayuda a encontrar la solución.

4. Lectura y análisis rápido

- **Entender enunciados rápidamente:** Los problemas suelen tener descripciones largas, por lo que ser capaz de leer rápidamente y entender las partes clave es importante. Practicar con problemas previos del ICPC te puede ayudar a mejorar esta habilidad.
- **Buscar patrones:** Los problemas a menudo se basan en técnicas o patrones conocidos. Ser capaz de reconocer un patrón rápidamente te permite abordar el problema más

eficientemente.

5. Conocimiento de matemáticas

- **Matemáticas discretas:** Aunque no es programación directa, muchas soluciones de problemas requieren conceptos de combinatoria, teoría de números, álgebra, probabilidad, y geometría.
- **Optimización matemática:** Saber cómo aproximar o reducir la complejidad de un problema usando matemáticas puede marcar una gran diferencia.

6. Gestión de recursos

- **Control del equipo:** En algunos concursos ICPC se comparte un solo computador. Saber gestionar el tiempo en el cual cada miembro del equipo lo usa es fundamental.
- **Organización de código:** Aunque no es estrictamente "no programación", tener un buen sistema para organizar el código, usar plantillas o snippets para las estructuras comunes, y tener un plan claro para implementar soluciones puede ahorrar tiempo.

7. Salud física y mental

- **Descanso adecuado:** Dormir bien antes del concurso y tener un buen manejo del estrés mejora la capacidad de concentración y rendimiento.
- **Hidratación y alimentación:** Estar bien alimentado e hidratado evita que la fatiga mental te afecte a mitad del concurso.

8. Cultura general y reglas del concurso

- **Conocer las reglas del concurso:** Es importante estar al tanto de cómo funcionan las penalizaciones por envíos erróneos, los sistemas de puntuación, y las reglas específicas del ambiente del concurso.
- **Uso adecuado del entorno de desarrollo:** Aunque no es programación directa, estar familiarizado con el entorno de desarrollo y saber usar los atajos del IDE o las herramientas proporcionadas (como los compiladores) puede ser muy útil.

9. Técnicas de pensamiento lógico y creativo

- **Pensamiento lateral:** A veces, los problemas requieren enfoques creativos o soluciones fuera de lo común. Practicar este tipo de pensamiento te puede dar ventaja frente a otros competidores que solo sigan un enfoque mecánico.
- **Pensamiento crítico:** Revisar continuamente si lo que estás haciendo tiene sentido o si estás resolviendo el problema correcto puede ahorrarte tiempo valioso.

En resumen, participar en un concurso como el ICPC requiere no solo habilidades técnicas en programación, sino también habilidades blandas, como trabajo en equipo, manejo de la presión y el tiempo, y estar bien físicamente. Estas cualidades te ayudarán a ser más eficiente y efectivo durante la competencia.### 1. Trabajo en equipo

DFS

```
#include <iostream>
#include <vector>

using namespace std;

typedef pair<int, int> ii; // Par de enteros
typedef vector<ii> vii; // Vector de pares de enteros
vector<vii> AdjList; // Lista de adyacencia
vector<int> dfs_num; // Vector para registrar el estado de cada nodo

const int DFS_WHITE = -1; // Valor que indica no visitado
const int DFS_BLACK = 1; // Valor que indica visitado

void dfs(int u) {
    cout << u << " "; // Vértice visitado
    dfs_num[u] = DFS_BLACK; // Marcar como visitado

    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j]; // Obtener el vecino v de u
        if (dfs_num[v.first] == DFS_WHITE) {
            dfs(v.first); // Realizar DFS en el vecino
        }
    }
}
```

Encontrar componentes conectados}

- Podemos simplemente usar el siguiente código para reiniciar DFS (o BFS) desde uno de los vértices que quedan por visitar para encontrar el siguiente componente.
- Este proceso se repite hasta que todos los vértices hayan sido visitados y tiene una complejidad de $O(V + E)$.

Código para encontrar componentes conectados

```
#include <iostream>
#include <vector>

using namespace std;

const int DFS_WHITE = -1; // Valor que indica no visitado

vector<int> dfs_num; // Vector para registrar el estado de cada nodo
int numCC = 0; // Número de componentes conectadas

void dfs(int u) {
    dfs_num[u] = 1; // Marcar como visitado
    // Aquí iría el código para recorrer los vecinos de `u`
    // Por ejemplo, si tienes un grafo representado con listas de adyacencia
    // recorrerías todos los vecinos de `u`.
}

int main() {
    int V; // Número de vértices
    cin >> V;
    dfs_num.assign(V, DFS_WHITE); // Inicializar todos los vértices como no
    visitados

    for (int i = 0; i < V; i++) {
        if (dfs_num[i] == DFS_WHITE) {
            cout << "Componente " << ++numCC << ":" << endl;
            dfs(i); // Llamar DFS desde el vértice i
    }
}
```

```

        cout << endl;
    }
}

cout << "Hay " << numCC << " componentes conectados" << endl;

return 0;
}

```

Flood filling

- DFS (o BFS) se puede usar también para etiquetar (también conocido en teoría de grafos como “colorear”) cada componente.
- Esta variante se conoce como ‘flood fill’ y es usualmente realizada en grafos implícitos (usualmente retículas 2D).

Código Flood filling en un grafo implícito

```

#include <iostream>
using namespace std;

const int dr[] = {1, 1, 0, -1, -1, -1, 0, 1}; // Movimiento en filas: S, SE,
E, NE, N, NW, W, SW
const int dc[] = {0, 1, 1, 1, 0, -1, -1, -1}; // Movimiento en columnas: S,
SE, E, NE, N, NW, W, SW

char grid[100][100]; // Asumimos una cuadrícula de 100x100 para el ejemplo
int R, C; // Tamaño de la retícula (R filas, C columnas)

// Función floodfill que devuelve el tamaño de la componente conectada (CC)
int floodfill(int r, int c, char c1, char c2) {
    // Si estamos fuera de la retícula
    if (r < 0 || r >= R || c < 0 || c >= C) return 0;

```

```

        if (grid[r][c] != c1) return 0; // Si no es del color c1, no lo contamos

        int res = 1; // Contamos el vértice (r, c)
        grid[r][c] = c2; // Lo coloreamos con c2 para evitar ciclos

        // Exploramos los 8 vecinos alrededor de (r, c)
        for (int d = 0; d < 8; d++) {
            res += floodfill(r + dr[d], c + dc[d], c1, c2);
        }

        return res; // Regresamos el tamaño de la componente conectada
    }

int main() {
    // Aquí puedes inicializar la retícula y probar la función floodfill.
    // Ejemplo:
    R = 5;
    C = 5;
    char c1 = 'A', c2 = 'B';

    // Asigna valores a la retícula 'grid' y llama a floodfill.
    // Por ejemplo, floodfill(2, 2, c1, c2);

    return 0;
}

```

Código Flood filling en un grafo no implícito

```

#include <iostream>
#include <vector>

using namespace std;

vector<vector<pair<int, int>>> AdjList; // Lista de adyacencia
vector<int> dfs_num; // Vector para guardar los colores

const int DFS_WHITE = -1; // Definimos el valor que representa no visitado

void floodfill(int u, int color) {
    dfs_num[u] = color;
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        int v = AdjList[u][j].first;
        if (dfs_num[v] == DFS_WHITE)

```

```

        floodfill(v, color);
    }

}

int main() {
    int V; // Cantidad de vértices
    // Inicializar aquí el grafo (AdjList) con V vértices y sus respectivas
    aristas

    int numCC = 0; // Contador de componentes conexas
    dfs_num.assign(V, DFS_WHITE); // Asigna DFS_WHITE a todos los nodos

    for (int i = 0; i < V; i++) {
        if (dfs_num[i] == DFS_WHITE)
            floodfill(i, ++numCC); // Floodfill para asignar componentes
conexas
    }

    for (int i = 0; i < V; i++) {
        printf("Vértice %d tiene color %d\n", i, dfs_num[i]);
    }

    return 0;
}

```

DFS

```

#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

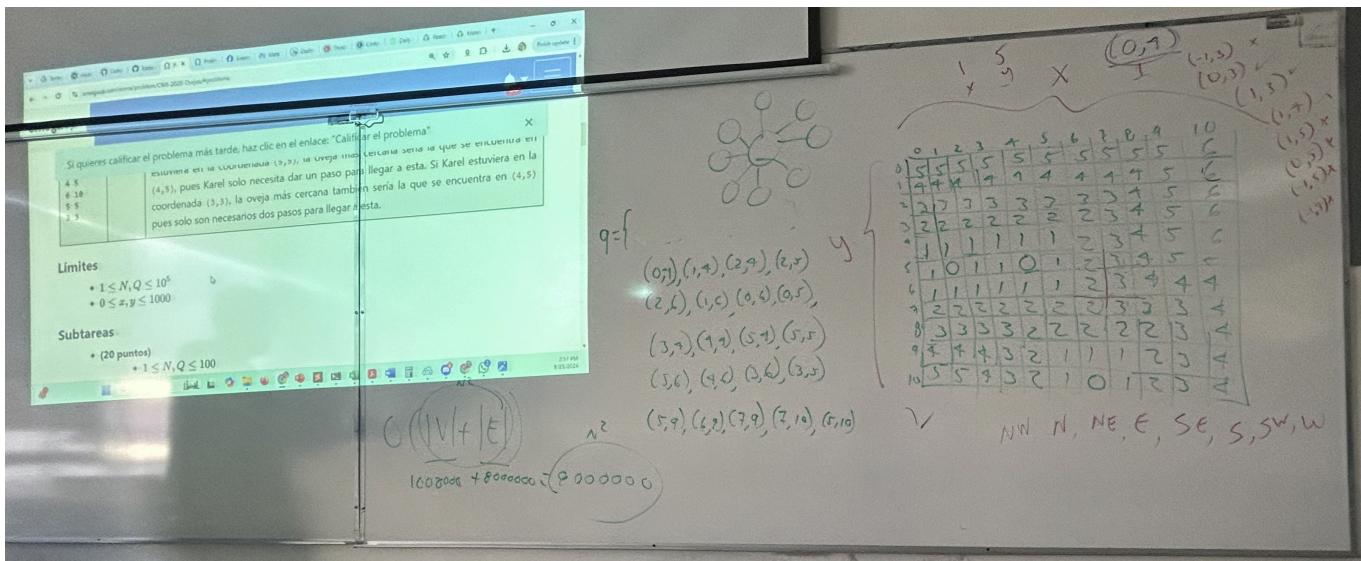
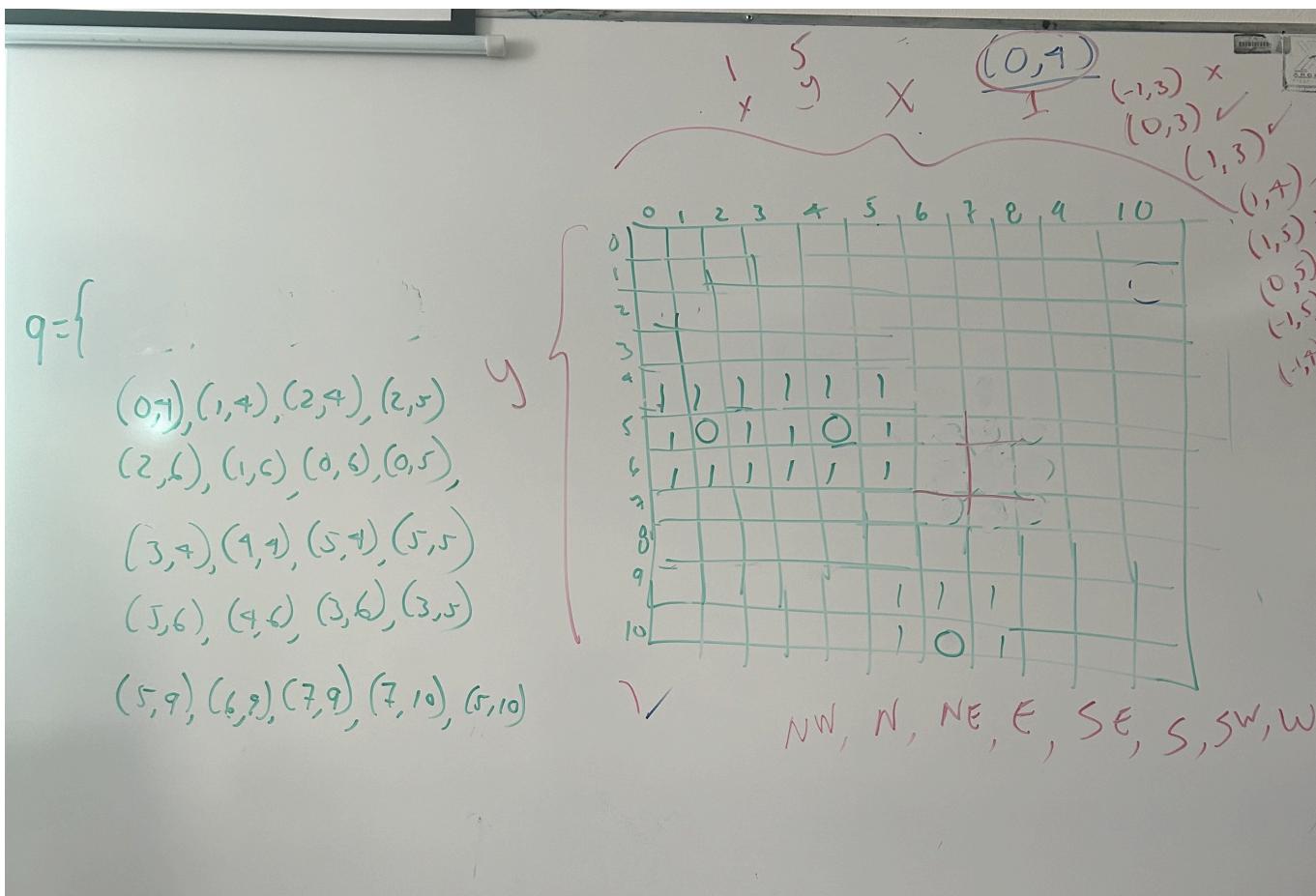
while (!q.empty()) {
    int u = q.front(); q.pop();                                // queue: layer by
layer!
    if (dist[u] != layer) printf("\nLayer %d: ", dist[u]);
    layer = dist[u];
    printf("visit %d, ", u);
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];                                // 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;
}

```

Determinar cuál es la Oveja más cercana a Karel



```
#include <bits/stdc++.h>
using namespace std;
#define MAXX 1000
#define MAXY 1000
#define INF 100000
typedef pair<int, int> ii;

int dist[MAXY+1][MAXX+1];
int q,n;
```

```

int dr[] = {1,1,0,-1,-1,-1, 0, 1}; // vecinos al S,SE,E,NE,N,NW,W,SW
int dc[] = {0,1,1, 1, 0,-1,-1,-1};

void floodfill_bfs(int n) {
    int i,j,x,y,xv,yv,dist_u;
    queue<ii> q;
    for (i=0; i<=MAXY; i++) {
        for (j=0; j<=MAXX; j++) {
            dist[i][j]=INF;
        }
    }
    for (i=0; i<n; i++) {
        scanf("%d %d",&x,&y);
        dist[y][x]=0;
        q.push(make_pair(x,y));
        //printf("Se mete (%d,%d) a la queue con distancia %d\n",x-1,y-1,dist[y-1][x-1]);
    }

    while (!q.empty()) {
        ii u=q.front(); q.pop();
        x = u.first; y=u.second;
        dist_u=dist[y][x];
        //printf("Se saca (%d,%d) a la queue con distancia %d\n",xv,yv,dist_u);
        for (i=0; i<8; i++) {
            xv=x+dc[i]; yv=y+dr[i];
            if (yv<0 || yv>MAXY || xv<0 || xv>MAXX) continue;
            if (dist[yv][xv]==INF) {
                dist[yv][xv]=dist_u+1;
                //printf("\t vecino en (%d, %d) con distancia %d\n",xv,yv,dist[yv][xv]);
                q.push(make_pair(xv,yv));
            }
        }
    }
}

int main() {
    int i;
    int x,y;
    scanf("%d %d", &n, &q);

    floodfill_bfs(n);
}

```

```

/*for (i=0; i<MAXY; i++) {
    for (j=0; j<MAXX; j++) {
        printf("%3d",dist[i][j]);
    }
    printf("\n");
}*/
for (i=0;i<q; i++) {
    scanf("%d %d",&x,&y);
    printf("%d\n",dist[y][x]);
}
//printf("%d\n",cont);
return 0;
}

```

Problema determinar cuántas casillas de la isla son Tierra

w w w w w w w L . /									
w L w w w w L w L . /									
w L w w , / / L L L ;									
w L L / / / / / / /									
w w w , / / / / / / /									
~ ~ ~ ~ ~ ~ ~ ~ ~ ~									
~ ~ ~ ~ ~ ~ ~ ~ ~ ~									
L L L									

```

#include <bits/stdc++.h>
using namespace std;
#define MAXN 100
#define MAXM 100

```

```

char grid[MAXM][MAXN+1];

int m,n;

int dr[] = {1,0,-1, 0}; // vecinos al S,E,N,W
int dc[] = {0,1, 0,-1}; //
void floodfill(int r, int c, char c1, char c2) {
    int i;
    int respuesta;
    if (r<0 || r>=m || c<0 || c>=n) return; // fuera del grid
    if (grid[r][c] != c1) return; // buscamos el color c1
    grid[r][c] = c2; // cambiamos el color a c2
    for (i=0; i<4; i++) {
        floodfill(r + dr[i], c + dc[i], c1, c2);
    }
}

int main() {
    int i,j;
    int cont=0;
    scanf("%d %d", &m, &n);
    for (i=0; i<m; i++) {
        scanf("%s",grid[i]);
    }

    for (i=0; i<m; i++) {
        for (j=0; j<n; j++) {
            if (grid[i][j]=='L') {
                floodfill(i,j,'L','W');
                cont++;
            }
        }
    }
    printf("%d\n",cont);
    return 0;
}

```

Determinar si un grafo es bipartita

```

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

typedef pair<int, int> ii;

```

```

typedef vector<ii> vii;
typedef vector<int> vi;

#define DFS_WHITE -1 // normal DFS
#define DFS_BLACK 1

vector<vii> AdjList;
int V; // numero de vertices

vi color; // Inicializar en DFS_WHITE (-1)
int bipartita(int s) {
    queue<int> q; q.push(s); color[s] = 0; int esBipartita = 1;
    while (!q.empty() && esBipartita) {
        int u = q.front(); q.pop();
        for (int j = 0; j < (int)AdjList[u].size(); j++) {
            ii v = AdjList[u][j];
            if (color[v.first] == DFS_WHITE) {
                color[v.first] = 1-color[u];
                q.push(v.first);
            }
            else if (color[v.first] == color[u]) {
                esBipartita=0; break;
            }
        }
    }
    return esBipartita;
}

int main() {
    int i;
    // lectura de datos ...
    color.assign(V, DFS_WHITE); // V es num de vertices
    for (int i = 0; i < V; i++) {
        if (color[i] == DFS_WHITE) {
            printf("Componente con raiz en %d:", i);
            int esB=bipartita(i);
            printf(", bipartita=%d\n", esB);
        }
    }
    return 0;
}

```

Código de ordenamiento topológico por BFS

```

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

typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;

#define DFS_WHITE -1 // normal DFS
#define DFS_BLACK 1

vector<vii> AdjList;
int V; // numero de vertices

vi topoSort; // vector para almacenar el toposort en orden inverso
vi in_degree;

// toposort con BFS
// Inicializar con la cantidad de aristas que llega a cada nodo

void topo_sort_kahn() {
    queue<int> Q;
    for (int u = 0; u < AdjList.size(); ++u)
        if (in_degree[u] == 0) Q.push(u);
    while (!Q.empty()) {
        char u = Q.front(); Q.pop();
        topoSort.push_back(u);
        for (int j = 0; j < (int)AdjList[u].size(); j++) {
            ii v = AdjList[u][j];
            in_degree[v.first]--;
            if (in_degree[v.first] == 0)
                Q.push(v.first);
        }
    }
}

int main() {
    // lectura de datos
    topo_sort_kahn();
    for (int i = 0; i < (int)topoSort.size(); i++) {
        printf(" %d", topoSort[i]);
    }
    printf("\n");
    return 0;
}

```

Ordenamiento Topológico usando DFS

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

typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;

#define DFS_WHITE -1 // normal DFS
#define DFS_BLACK 1

vector<vii> AdjList;
int V; // numero de vertices

vi topoSort; // vector para almacenar el toposort en orden inverso
vi dfs_num;

// toposort con DFS
void dfs2(int u) {
    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); // este es el unico cambio
}

int main() {
    // lectura de datos
    topoSort.clear();
    dfs_num.assign(V, DFS_WHITE);
    for (int i = 0; i < V; i++) {
        if (dfs_num[i] == DFS_WHITE) {
            dfs2(i);
        }
    }
    reverse(topoSort.begin(), topoSort.end());
    for (int i = 0; i < (int)topoSort.size(); i++) {
        printf(" %d", topoSort[i]);
    }
    printf("\n");
}
```

```
    return 0;  
}
```

Solución Wetlands of Florida

```
#include <bits/stdc++.h>  
using namespace std;  
  
#define MAXN 100  
#define MAXM 100  
char line[MAXN+1], grid[MAXM][MAXN+1];  
  
int m,n;  
  
int dr[] = {1,1,0,-1,-1,-1, 0, 1}; // S,SE,E,NE,N,NW,W,SW  
int dc[] = {0,1,1, 1, 0,-1,-1,-1}; // neighbors  
  
int floodfill(int r, int c, char c1, char c2) {  
    if (r<0 || r>=m || c<0 || c>=n) return 0; // outside  
    if (grid[r][c] != c1) return 0; // we want only c1  
    grid[r][c] = c2; // important step to avoid cycling!  
    int ans = 1; // coloring c1 -> c2, add 1 to answer  
    REP (d, 0, 7) // recurse to neighbors  
        ans += floodfill(r + dr[d], c + dc[d], c1, c2);  
    return ans;  
}  
  
// inside the int main() of the solution for UVa 469 - Wetlands of Florida  
int main() {  
    // read the implicit graph as global 2D array 'grid'/R/C and (row, col)  
    // query coordinate  
    sscanf(gets(line), "%d", &TC);  
    gets(line); // remove dummy line  
  
    while (TC--) {  
        R = 0;  
        while (1) {  
            gets(grid[R]);  
            if (grid[R][0] != 'L' && grid[R][0] != 'W') // start of query  
                break;  
            R++;  
        }  
        C = (int)strlen(grid[0]);
```

```

strcpy(line, grid[R]);
while (1) {
    sscanf(line, "%d %d", &row, &col); row--; col--; // index starts from
0!
    printf("%d\n", floodfill(row, col, 'W', '.')); // change water 'W' to
'.'; count size of this lake
    floodfill(row, col, '.', 'W'); // restore for next query
    gets(line);
    if (strcmp(line, "") == 0 || feof(stdin)) // next test case or last
test case
        break;
}
if (TC)
    printf("\n");
}

return 0;
}

```

DP

Knapsack

```

/* A simple program demonstrating how to solve knapsack problem using
 * bottom up dynamic programming. Simple idea is if the size is manageable
 * then for each item I have the choice to include it or exclude it, keeping
 * a memo of value if I include it and I dont include the current item. */
#include <iostream>
#include <algorithm>
using namespace std;

int solve(int n, int vals[], int weights[], int S) {
    int memo[n+1][S+1];
    for (int i = 0; i <= S; ++i) memo[0][i] = 0;
    for(int i=1; i<=n; ++i) {
        for(int s=0; s <= S; ++s) {
            if(weights[i-1] > s)
                memo[i][s] = memo[i-1][s];
            else {
                memo[i][s] = max(memo[i-1][s], memo[i-1][s-

```

```

weights[i-1]]+vals[i-1]);
        }
    }
    return memo[n][S];
}

int main(int argc, char* argv[]) {
    int n; // the number of items
    cin >> n;
    int vals[n], weights[n];
    for(int i=0; i<n; ++i) {
        cin >> vals[i] >> weights[i];
    }
    int S; // the weight of the knapsack
    cin >> S;
    int max_amount = solve(n, vals, weights, S);
    cout << max_amount << endl;
    return 0;
}

```

Sliding Window

```

#include <bits/stdc++.h>

using namespace std;

int contarSubcadena(string s){
    set<char> chars;
    int cont1 = 0, maxSubstring = 0;
    for (int i=0; i < s.length(); i++){
        if (chars.count(s[i]) == 0) //si no está lo metemos
        {
            chars.insert(s[i]);
            maxSubstring = max(maxSubstring, i - cont1 + 1);
        } else { //si está borramos todo mientras se repita
            while (chars.count(s[i]))
            {
                chars.erase(s[cont1]);
                cont1++;
            }
            chars.insert(s[i]);
        }
    }
}

```

```

        }
    }

    return maxSubstring;
}

int main()
{
    string s; cin>> s;
    int salida = contarSubcadena(s);
    cout << salida << endl;

    return 0;
}

```

Bitwise operations

AND		
A	B	Result
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	Result
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
A	B	Result
0	0	0
0	1	1
1	0	1
1	1	0

NOT	
A	Result
0	1
1	0

Para saber si un bit en específico está encendido:

- num & (1 << bitPosition) != 0

Problema number of 1 bits (cuántos bits están en 1)

```

#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n; cin >> n;
    int cont = 0;

    while (n != 0)

```

```
{  
    cont++;  
    n = n & (n-1);  
}  
  
cout << cont << endl;  
  
return 0;  
}
```