Soluciones OIFem III

Prueba de Nivel

¡Hola mundo!

Teoría

- Entrada y salida de datos
- Bucles

Solución

Usando `cin` o `scanf`, leemos el número de la entrada. Iteramos ese número de veces usando un bucle, en cada iteración imprimiendo la frase dada con `cout` o `printf`.

Código

```cpp
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    while(n--) {
        cout << "Hola mundo.\n";
    }
    return 0;
}
```

Rebotando en el parchís

Teoría

- Entrada y salida de datos
- Aritmética

Solución

Usando `cin` o `scanf`, leemos los números de la entrada. Dividimos las posibilidades en dos: nuestra ficha no llegará a la ficha final, en cuyo caso nos bastará con imprimir la suma de la posición actual y el número de posiciones que nos hemos de mover, y el caso donde la ficha sí llegará a la casilla final, en cuyo caso restamos de esta casilla los movimientos sobrantes.

Código

```cpp
#include <iostream>

using namespace std;

int main() {
    int c, p, t;
    while(true) {
        cin >> c >> p >> t;
        if (!(c || p || t)) break;
        if (p + t <= c) cout << p + t << '\n';
        else cout << c - (p + t - c) << '\n';
    }
    return 0;
}
```

Constante de Kaprekar

Teoría

- Strings
- Conversión entre tipos de datos

Solución

Para resolver este problema, simulamos el proceso dado, convirtiendo los números a strings en las operaciones para las que las strings son una estructura de datos más cómoda (por ejemplo, ordenar los caracteres o escribir el número al revés) y usando ints para las operaciones para las que son necesarios, como las restas.

Código

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int t;
    string s1, s2;
    cin >> t;
    while(t--) {
        cin >> s1;
        while (s1.length() < 4) {
            s1 = "0" + s1;
        }
        if (stoi(s1) % 1111 == 0) {
            cout << "8\n";
            continue;
        }
        int it = 0;
        while(s1 != "6174") {
            sort(s1.begin(), s1.end());
            s2 = s1;
            reverse(s2.begin(), s2.end());
            s1 = to_string(stoi(s2) - stoi(s1));
            while (s1.length() < 4) {
                s1 = "0" + s1;
            }
            it++;
        }
        cout << it << '\n';
    }
}
```

Tableros de ajedrez

Teoría

- Bucles

Solución

Este era un problema de implementación. Esto quiere decir que, en vez de tener que idear un algoritmo creativo para su resolución, debemos centrarnos más en cómo escribir el código para asegurarnos de que se cumple exactamente lo que dice el enunciado.

Una forma sencilla de hacer esto es crear una string de fila oscura y otra de fila clara e ir alternándolas para imprimir el tablero en cuestión.

Código

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int tamano;
    string caracter;
    while(true) {
        cin >> tamano;
        if (!tamano) break;
        cin >> caracter;
        string filaClara = "|", filaOscura = "|";
        for (int i = 0; i < 8; i++) {
            if (i & 1) {
                for (int j = 0; j < tamano; j++) {
                    filaClara += caracter;
                    filaOscura += " ";
                }
            } else {
                for (int j = 0; j < tamano; j++) {
                    filaClara += " ";
                    filaOscura += caracter;
                }
            }
        }
        filaClara += "|";
        filaOscura += "|";
        string filaBorde = "|";
        while(filaBorde.length() + 1 < filaClara.length()) {
            filaBorde += "-";
        }
        filaBorde += "|";
        cout << filaBorde << '\n';
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < tamano; j++) {
                if (i&1) cout << filaOscura << '\n';
                else cout << filaClara << '\n';
            }
        }
```

```
40            cout << filaBorde << '\n';
41        }
42    }
```

Poniendo la mesa

Teoría

- Recursión

Solución

Para resolver este problema, la manera más sencilla era usar recursión, teniendo en cuenta en todo momento el número de copas restantes y quién ha de realizar el viaje. En este caso, lo hemos programado con una técnica llamada *recursión múltiple*, que consiste en diseñar dos funciones que se llamen mutuamente. Pueden ambas tener un caso base (o varios) o que solo lo tenga una. También podría resolverse con una única función con un *flag binario*, pero es una práctica menos limpia.

Código

```cpp
#include <iostream>
using namespace std;

int viajesYo(int copas); // para poder hacer la recursión múltiple

int viajesHermano(int copas) {
    if (copas <= 0) {
        // si no hay un número positivo de copas, no hacen falta viajes
        return 0;
    }
    return 1 + viajesYo(copas-1);
}

int viajesYo(int copas) {
    if (copas <= 0) {
        // si no hay un número positivo de copas, no hacen falta viajes
        return 0;
    }
    if (copas % 2 == 0) {
        // si es mi turno y hay un número par, llevo dos copas
        return 1 + viajesHermano(copas-2);
    } else {
        // si no, llevo una copa
        return 1 + viajesHermano(copas-1);
    }
}

int main() {
    int copas;
    while(true) {
        cin >> copas;
        if (!copas) break;
        cout << viajesHermano(copas) << "\n";
    }
    return 0;
}
```

Por 3 o más 5

Teoría

- Montículos (priority queues)
- Conjuntos desordenados (hash/unordered sets)

Solución

Para resolver este problema, hemos utilizado un montículo. Por él pasarán todos los números no mayores de 20.000 que puedan formarse con una sucesión de operaciones del tipo $\times 3$ o $+5$ partiendo de 1. Nos aseguraremos de que pasen todos empezando únicamente con un 1 en el montículo y, hasta que este esté vacío, retirando el menor número del montículo (llamémosle k) y añadiendo $3k$ y $k+5$ al montículo si no están ya en este y su valor no supera 20.000. Iremos guardando todos estos números en un conjunto desordenado, para que así podamos procesar cada número de la entrada en $O(1)$.

Código

```cpp
#include <iostream>
#include <queue>
#include <unordered_set>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    priority_queue<int> pq;
    unordered_set<int> possible;
    possible.reserve(4096);
    pq.push(-1);
    possible.insert(1);
    while(pq.size()) {
        int k = -pq.top();
        pq.pop();
        if (3*k <= 20000 && possible.find(3*k) == possible.end()) {
            pq.push(-3*k);
            possible.insert(3*k);
        }
        if (k+5 <= 20000 && possible.find(k+5) == possible.end()) {
            pq.push(-k-5);
            possible.insert(k+5);
        }
    }
    int n;
    while(true) {
        cin >> n;
        if (!n) break;
        if (possible.find(n) != possible.end()) cout << "SI\n";
        else cout << "NO\n";
    }
}
```

Los amigos de mis amigos son mis amigos

Teoría

- Grafos
- Componentes conexas/ union-find disjoint sets

Solución

Si consideramos a los amigos nodos y a las amistades aristas, este problema se reduce a encontrar el tamaño de la mayor componente conexa. Esto lo podemos resolver aplicando de forma muy directa DFS, BFS o UFDS. En el siguiente código lo haremos con DFS.

Código

```cpp
#include <iostream>
#include <vector>
using namespace std;

int dfs(int nodo, vector<bool> & visited, vector<vector<int>> & listaAdy) {
    visited[nodo] = true;
    int ret = 1;
    for (int vecino: listaAdy[nodo]) {
        if (!visited[vecino]) ret += dfs(vecino, visited, listaAdy);
    }
    return ret;
}

int mayorComponenteConexa(vector<vector<int>> & listaAdy) {
    int N = (int) listaAdy.size();
    vector<bool> visited(N, false);
    int mayorCC = 0;
    for (int i = 0; i < N; i++) {
        if (!visited[i]) {
            mayorCC = max(mayorCC, dfs(i, visited, listaAdy));
        }
    }
    return mayorCC;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int t, N, M, A, B;
    cin >> t;
    while(t--) {
        cin >> N >> M;
        vector<vector<int>> listaAdy(N, vector<int>());
        while(M--) {
            cin >> A >> B;
            listaAdy[A-1].push_back(B-1);
            listaAdy[B-1].push_back(A-1);
        }
        cout << mayorComponenteConexa(listaAdy) << '\n';
    }
    return 0;
```

```
42 }
```

Ratones en un laberinto

Teoría

- Dijkstra

Solución

Este problema fue extraído de la Olimpiada Informática de Madrid de 2021. Os remitimos, por lo tanto, a las soluciones oficiales del concurso.

DNI incompleto

Teoría

- Búsqueda completa

Solución

Este problema se resolvía aplicando una búsqueda completa, cuidando algo los detalles de la implementación para evitar un TLE.

Código

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

const string letras = "TRWAGMYFPDXBNJZSQVHLCKE";
string dni;
int total;

void bt(int i, int num) {
    if (i == 8) {
        if (letras[num % 23] == dni[8]) ++total;
    }
    else {
        if (dni[i] == '?') {
            for (int digito = 0; digito < 10; ++digito) {
                bt(i+1, num*10 + digito);
            }
        }
        else {
            int digito = dni[i] - '0';
            bt(i+1, num*10 + digito);
        }
    }
}

int main() {
    int N;
    cin >> N;
    while (N--) {
        cin >> dni;
        total = 0;
        bt(0, 0);
        cout << total << endl;
    }
}
```

La ardilla viajera

Teoría

- Union-find disjoint sets

Solución

Para resolver este problema, la clave era, en lugar de construir el bosque e ir talando los árboles, simular este proceso al revés. Es decir, empezar por un bosque ya talado e ir "destalando" árboles en el orden inverso en el que fueron talados. Así, podremos obtener una imagen del bosque tras cada árbol talado, pero de forma mucho más rápida. Para ir añadiendo aristas a un grafo de forma constructiva, preservando su estructura de componentes conexas y pudiendo revisarla rápido, usamos la estructura de datos UFDS.

Código

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> parent;
vector<int> rnk;
vector<bool> hasTree;
vector<pair<int, int>> trees;
int N, M;

int root(int a) {
    if (parent[a] != a) parent[a] = root(parent[a]);
    return parent[a];
}

int pointToVal(pair<int, int> p) {
    return p.first*(M+1) + p.second;
}

void mergeByRank(int a, int b) {
    int r_a, r_b;
    r_a = root(a);
    r_b = root(b);
    if (r_a == r_b) return;
    if (rnk[r_a] > rnk[r_b]) {
        parent[r_b] = r_a;
    } else if (rnk[r_b] > rnk[r_a]) {
        parent[r_a] = r_b;
    } else {
        parent[r_a] = r_b;
        rnk[r_b]++;
    }
}

bool outside(int i, int j) {
    return i > N || j > M || i < 0 || j < 0;
}

void fillNearby(pair<int, int> p, int k) {
    int col;
```

```
    int dist_sqd = k*k;
    for (int i = -k; i <= k; i++) {
        for (int j = -k; j <= k; j++) {
            if (i*i + j*j > k*k) continue;
            if (outside(p.first + i, p.second + j)) continue;
            if (hasTree[pointToVal({p.first + i, p.second + j})])
                mergeByRank(root(pointToVal(p)), root(pointToVal({p.first + i, p.second +
↪   j})));
        }
    }
}

int main() {
    int k, n, x, y, fil, col;
    pair<int, int> empty_pair = {0, 0};
    vector<pair<int, int>> empty_row;
    vector<int> empty_r;
    bool found;
    while (cin >> N) {
        if (!cin) break;
        cin >> M >> k >> n;
        parent.assign((N+1)*(M+1), 0);
        rnk.assign((N+1)*(M+1), 0);
        hasTree.assign((N+1)*(M+1), false);
        hasTree[0] = true;
        hasTree[pointToVal({N, M})] = true;
        for (int i = 0; i <= N; i++) {
            for (int j = 0; j <= M; j++) {
                parent[pointToVal({i, j})] = pointToVal({i, j});
            }
        }

        trees.assign(n, empty_pair);
        for (int i = 0; i < n; i++) {
            cin >> x >> y;
            trees[i].first = x;
            trees[i].second = y;
        }
        found = false;

        for (int i = n-1; i >= 0; i--) {
            fil = trees[i].first;
            col = trees[i].second;
            if (found) continue;
            else {
                hasTree[pointToVal({fil, col})] = true;
                fillNearby(trees[i], k);
            }
            if (root(0) == root(pointToVal({N, M}))) {
                found = true;
                cout << fil << " " << col << endl;
            }
        }

        if (!found) {
            cout << "NUNCA SE PUDO\n";
        }
    }
    return 0;
}
```

Pintando la pared

Teoría

- Divide y vencerás (skyline)
- Ordenamiento y simulación

Solución

Este problema se podía resolver de dos formas: usando el algoritmo de skyline, una técnica conocida del paradigma *divide y vencerás*, o a través de una simulación inteligente del proceso (incluida aquí). Esta consiste en tratar el principio de un rectángulo de pintura como un evento y su final como otro. Ordenamos los eventos en función de su distancia del origen y su altura y vamos calculando el área total iterando sobre los eventos en este orden.

Código

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

const int ACABA = 0;
const int EMPIEZA = 1;

struct Evento {
    int x, y, tipo;
    Evento(int x_, int y_, int tipo_) {
        x = x_;
        y = y_;
        tipo = tipo_;
    }
};

bool operator<(Evento lhs, Evento rhs) {
    if (lhs.x != rhs.x)
        return lhs.x < rhs.x;
    return lhs.tipo < rhs.tipo;
}

int main() {
    int a, n;
    while ((cin >> a) and a != 0) {
        cin >> n;
        vector<Evento> eventos;
        for (int i = 0; i < n; ++i) {
            int x, y;
            cin >> x >> y;
            eventos.push_back(Evento(x, y, EMPIEZA));
            eventos.push_back(Evento(x+a, y, ACABA));
        }
        sort(eventos.begin(), eventos.end());
        multiset<int> alturas_activas;
        alturas_activas.insert(0);
        int prev_x = 0;
        long long total = 0;
        for (Evento e : eventos) {
            int altura_max = *alturas_activas.rbegin();
```

```
            total += (long long)(e.x - prev_x)*altura_max;
            prev_x = e.x;
            if (e.tipo == EMPIEZA) {
                alturas_activas.insert(e.y);
            }
            else { // e.tipo == ACABA
                alturas_activas.erase(alturas_activas.find(e.y));
            }
        }
        cout << total << '\n';
    }
}
```