Mazmorra

Después de una ardua jornada de entrenar tus habilidades de sigilo al nivel 100, decides entrenar tu fuerza. Para ello, acabas de entrar en una mazmorra que contiene n monstruos. Inicialmente, tu nivel de fuerza es 0 y cada monstruo tiene un nivel f_i. Puedes derrotar a un monstruo si tu nivel de fuerza es mayor o igual al suyo. Cuando derrotas a un monstruo, este desaparece y tu nivel de fuerza aumenta en 1. Como eres muy sigiloso, puedes atacar a los monstruos en el orden que quieras sin que se den cuenta. No es necesario acabar con todos los monstruos.

¿Cuál es el máximo nivel de fuerza que puedes alcanzar?

Estrategia de la solución

En este problema, la observación clave es que siempre conviene atacar a los monstruos de menor a mayor nivel. En efecto, si es posible atacar a un monstruo de nivel mayor antes que otro de nivel menor, también será posible hacerlo al revés ya que nuestro nivel solo puede aumentar después de acabar con un monstruo.

Por lo tanto, la solución esperada consiste en ordenar los niveles de los monstruos de menor a mayor y simular las batallas en este orden. Esto ya es suficiente para pasar los casos de prueba, pero una optimización posible es parar la simulación después del primer monstruo que no puedes derrotar, ya que tampoco podrás derrotar los siguientes.

Solución en Python

```python
ncasos = int(input())
for caso in range(ncasos):

    num_enemigos = int(input())
    enemigos = []

    for i in range(num_enemigos):
        # anado el siguiente enemigo a la lista
        enemigos.append(int(input()))

    # ordeno la lista enemigos
    enemigos = sorted(enemigos)

    # lvl es mi nivel actual
    lvl = 0
    for i, enemigo in enumerate(enemigos):
        # si puedo derrotar al enemigo, subo de nivel
        if lvl >= enemigo:
                lvl = lvl+1

    print(lvl)
```

Autor de la solución: Javier López-Contreras (Miembro del comité organizador)

Solución en C++

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        vector<int> V(n);
        for (int i = 0; i < n; ++i) cin >> V[i];
        sort(V.begin(), V.end());
        int l = 0;
        for (int i = 0; i < n; ++i) {
               if (V[i] <= l) ++l;
        }
        cout << l << endl;
    }
}
```

Autor de la solución: Jan Olivetti (Miembro del comité organizador)

TurboFlex

En el año 2023, los empleos fijos son cosa del pasado y TuboFlex™ ha instalado tubos en cada puesto de trabajo. Con ellos, puedes transportarte fácilmente de un puesto a otro para completar tareas donde sea.

Tu día laborable consiste en *s* segundos, y hay *n* tareas a completar cada día. La tarea *i*-ésima empieza en el segundo e_i, termina en el segundo f_i y te pagan p_i euros por completarla. Solo puedes completar una tarea a la vez y tienes que permanecer la duración entera de esta para completarla, pero una vez terminada puedes inmediatamente empezar otra tarea, ya que los tubos son un método de transporte muy rápido.

Quieres elegir las tareas de modo que maximices tus ganancias. ¿Cuál es la cantidad máxima de dinero que puedes obtener?

Estrategia de la solución

Sea *G(i)* la cantidad máxima de dinero que se puede obtener del segundo *0* al segundo *i*. Entonces, *G(0) = 0* y *G(i)* será el máximo entre *G(i-1)* (no coger ninguna tarea en el segundo *i-1*) y $G(e_j) + p_j$ (coger la tarea *j*, solo se puede si esta termina en el segundo *i*). Hacer un backtracking superaría el límite de tiempo, pero este problema se puede solucionar con *programación dinámica*. El truco está en guardarse en un vector todas las *G(i)* que ya hayamos calculado, para no tenerlas que calcular otra vez si volvemos a llamar *G(i)*. La respuesta al problema será *G(s)*. Este truco se llama *memorización* y la programación dinámica resultante, *top-down*, porque empiezas con el valor de *i* más grande y vas bajando a los más pequeños. Con esto era suficiente para pasar los casos de prueba.

También vamos a discutir como resolver este problema de forma iterativa, utilizando una programación dinámica *bottom-up*, es decir, empezar con la *i* más pequeña y propagar hacia arriba. Guardamos en la posición *i* de un vector todas las tareas *j* con $e_j = i$, y creamos otro vector para guardar los *G(i)* ya calculados. Empezamos con el caso base, *G(0) = 0*. Iteramos sobre todas las *i* en orden ascendente y las propagamos, teniendo en cuenta el caso en que no tomamos ninguna tarea y el caso donde tomamos alguna de las tareas que empieza en el segundo *i* (aquí conviene, antes de empezar el programa, guardar en la posición *i* de un vector todas las tareas *j* con $e_j = i$). Si llegamos a una misma *G(i)* con distintos valores nos quedamos con el máximo. La solución continúa siendo *G(s)*. El código en C++ muestra la implementación de la programación dinámica *bottom-up*.

Solución en Python

```
ncasos = int(input())
for caso in range(ncasos):

    line = input().split(' ')
    s = int(line[0])
    n = int(line[1])
```

```
    # guardo el input en la lista jobs
    jobs = []

    for i in range(n):
        line = input().split(' ')
        jobs.append( (int(line[0]), int(line[1]), int(line[2])) )

    jobs = sorted(jobs)
    lastjob = len(jobs)-1

    # la dp responde la pregunta: cuanto dinero puedes conseguir si empiezas
    # a trabajar en el segundo i-esimo?
    dp = [0 for i in range(s+1)]

    i = s-1
    while i >= 0:
        dp[i] = max(dp[i], dp[i+1])

        if lastjob >= 0 and i == jobs[lastjob][0]:
                # si hay un trabajo, escojo lo que me de mas
                # dinero entre trabajar y no trabajar
                dp[i] = max(dp[i], dp[ jobs[lastjob][1] ] + jobs[lastjob][2])

                lastjob = lastjob-1;
                i = i+1
        i = i-1

    print(dp[0]);
```

Autor de la solución: Javier López-Contreras (Miembro del comité organizador)

Solución en C++

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

typedef long long ll;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int t;
    cin >> t;
    while (t--) {
        int s, n;
        cin >> s >> n;
        vector<ll> G(s + 1, 0);
        //guardamos los f_i en F y los p_i en P
        vector< vector<ll> > F(s + 1);
        vector< vector<ll> > P(s + 1);
        for (int i = 0; i < n; ++i) {
                int e, f, p;
                cin >> e >> f >> p;
```

```
                //guardamos las tareas en la posicion e_i
                F[e].push_back(f);
                P[e].push_back(p);
        }
        for (int i = 0; i < s; ++i) {
                //consideramos las tareas que empiezan en el segundo i
                for (int j = 0; j < F[i].size(); ++j) {
                        int f = F[i][j];
                        if (G[f] < G[i] + P[i][j]) G[f] = G[i] + P[i][j];
                }
                //esperar un segundo
                if (G[i + 1] < G[i]) G[i + 1] = G[i];
        }
        cout << G[s] << endl;
    }
}
```

Autor de la solución: Jan Olivetti (Miembro del comité organizador)

Perdido

Estás perdido y sediento en el desierto. Además, el viento sopla muy fuerte. Si no encuentras un lugar seguro pronto, morirás.

El desierto se puede representar como un tablero de *n-1* filas y columnas. Las filas están numeradas de *0* a *n-1* de arriba a abajo, y las columnas están numeradas de *0* a *m-1* de izquierda a derecha. Inicialmente, tienes *u* unidades de agua, estás en la columna *e*-ésima de la fila *0* y tu destino es la columna *f*-ésima de la fila *n-1*. Hay tres tipos de celdas: '.' son las celdas normales, '#' son dunas por las cuales no puedes pasar y 'O' son oasis.

Como el viento sopla hacia abajo muy fuerte, tus movimientos solo pueden ser hacia abajo, abajo a la izquierda o abajo a la derecha, sin salir del desierto. Cada movimiento te hace perder una unidad de agua, y llegar a un oasis te devuelve a *u* unidades de agua. Si llegas a *0* unidades de agua sin llegar a un oasis, mueres.

¿Cuántos caminos existen que te lleven con vida a tu destino? Escribe este número módulo 10^9+7.

Estrategia de la solución

Este problema también se resuelve utilizando programación dinámica. La función es más complicada ya que depende de tres variables: *DP(i, j, k)* es el número de maneras de llegar a la fila *i*, columna *j* con exactamente *k* unidades de agua restantes. Es posible pasar los casos de prueba con una solución recursiva *top-down*, pero es más sencillo pensarlo de forma iterativa *bottom-up*.

El caso base es *DP(0, j, k) = 1* si *i = 0, j = e, k = u* y igual a *0* en cualquier otro caso. Como que no podemos retroceder a una fila anterior, conviene iterar primero sobre las filas *i* de *0* a *n-1* en este orden y después sobre las columnas y unidades de agua restantes. Si empezamos con *DP(i, j, k)*, propagaremos hacia *DP(i+1, j-1, k-1)*, *DP(i+1, j, k-1*, *DP(i+1, j+1, k-1)*, excepto si la casilla destino es una duna, en cuyo caso no propagamos porque no se puede pasar por ahí, y si es un oasis en vez de poner *k-1* ponemos *u*. Como que nos interesa el número total de caminos, y no encontrar un óptimo, al propagar sumamos en vez de quedarnos con el máximo. Al implementarlo, tenemos que tener cuidado de no propagar a un estado inválido (fuera del tablero o con *0* unidades de agua) y tenemos que utilizar módulos para evitar *overflow*.

La solución es la suma de *DP(n-1, f, k)* para todos los *k* entre *1* y *u*.

Solución en C++

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

typedef long long ll;
```

```cpp
typedef vector<ll> VI;
typedef vector<VI> VVI;
typedef vector<VVI> VVVI;
const ll MOD = 1e9 + 7;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int t;
    cin >> t;
    while (t--) {
        int n, m, u, e, f;
        cin >> n >> m >> u >> e >> f;
        //mapa del desierto
        vector< vector<char> > B(n, vector<char>(m));
        for (int i = 0; i < n; ++i) {
               for (int j = 0; j < m; ++j) {
                      cin >> B[i][j];
               }
        }
        VVVI DP(n, VVI(m, VI(u + 1, 0)));
        DP[0][e][u] = 1;
        for (int i = 0; i < n - 1; ++i) {
               for (int j = 0; j < m; ++j) {
                      for (int k = 1; k <= u; ++k) {
                             for (int dx = -1; dx <= 1; ++dx) {
                                    if (j + dx >= 0 and j + dx < m) {
                                           //nivel de agua despues de moverse
                                           int nk = k - 1;
                                           //si acabamos en un oasis el nivel de
                                           //agua es u
                                           if (B[i + 1][j + dx] == 'O')
                                                  nk = u;
                                           //solo propagamos si no acabamos en
                                           //una duna
                                           if (B[i + 1][j + dx] != '#')
                                                  DP[i + 1][j + dx][nk] += DP[i][j][k];
                                           DP[i + 1][j + dx][nk] %= MOD;
                                    }
                             }
                      }
               }
        }
        ll count = 0;
        for (int k = 1; k <= u; ++k) {
               count = count + DP[n - 1][f][k];
               count %= MOD;
        }
        cout << count << endl;

    }
}
```

Autor de la solución: Jan Olivetti (Miembro del comité organizador)