Soluciones OIFem 2020
Entrenamiento 5

Torres de Hanoi

Teoría

- Exponenciación rápida
- Overflow y aritmética modular

Solución

La respuesta para un número n es $2^{n+1} - (2 + n)$. Para evitar overflow, vamos tomando el módulo en todo momento del programa en el que corramos este riesgo.

Código

C++

```cpp
#include <iostream>
using namespace std;
long long int mod = 1e9 + 7;

long long int exponenciacion (long long int base, int potencia) {
    if (potencia == 0)
        return 1;
    long long int x = exponenciacion(base, potencia >> 1);
    if (potencia & 1)
        return (((x*x) % mod)*base)%mod;
    else
        return (x*x)%mod;
}

int main() {
    int t, n;
    cin >> t;
    while(t--) {
        cin >> n;
        cout << (exponenciacion(2, n+1) - (2+n) + mod)%mod << '\n';
    }
    return 0;
}
```

Correo no deseado

Teoría

- Método de bisección
- Ordenamiento
- Algoritmos voraces

Solución

Usamos el método de bisección para encontrar el mínimo tiempo en el que María y sus amigos podrán limpiar Badajoz de correo no deseado. Para ello, usamos la función de bisección para buscar un mínimo aprendida en clase.

Solo necesitamos una función que comprueba si es posible limpiar todo el correo en un tiempo k. Para ello, necesitamos tener las posiciones de los amigos y de los buzones ordenadas. Esto lo haremos dentro de la función que aplica la bisección, ya que así evitamos ordenar las listas con cada comprobación.

Esta comprobación la haremos iterando por las posiciones de los amigos y guardando cuántos buzones han limpiado hasta el momento. De una manera voraz, le asignamos a cada amigo el máximo número de buzones en orden que puede limpiar en el tiempo k. Una vez hayamos terminado de iterar por los amigos, retornamos `true` en caso de que los N buzones hayan sido limpiados.

Código

C++

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

typedef long long int ll; // incluimos esta línea opcional para poder escribir ll en vez de long long int, algo útil en concursos para ganar tiempo

bool posible(vector<ll> & posicionBuzones, vector<ll> & posicionAmigos, ll k) {
    ll posicionAmigo;
    int N = (int) posicionBuzones.size(), M = (int) posicionAmigos.size(), alcanzado = 0, ultimoAlcanzado;
    for (int p = 0; p < M; p++) {
         posicionAmigo = posicionAmigos[p];
         ultimoAlcanzado = alcanzado;
         while(alcanzado < N && posicionBuzones[alcanzado] - posicionBuzones[ultimoAlcanzado] + min(abs(posicionBuzones[alcanzado] - posicionAmigo), abs(posicionBuzones[ultimoAlcanzado] - posicionAmigo)) <= k)
             alcanzado++;
    }
    return alcanzado == N;
}
```

```cpp
ll minimoK(vector<ll> & posicionBuzones, vector<ll> & posicionAmigos) {
    ll lo, hi, mid, respuesta;
    sort(posicionBuzones.begin(), posicionBuzones.end());
    sort(posicionAmigos.begin(), posicionAmigos.end());
    lo = 0;
    hi = 2e18;
    respuesta = 2e18;
    while(lo <= hi) {
        mid = lo + (hi-lo)/2;
        if (posible(posicionBuzones, posicionAmigos, mid)) {
            hi = mid-1;
            respuesta = mid;
        }
        else lo = mid+1;
    }
    return respuesta;
}

int main() {
    int T, N, M;
    cin >> T;
    vector<ll> posicionBuzones, posicionAmigos;
    while(T--) {
        cin >> N >> M;
        posicionBuzones.assign(N, 0);
        for (int n = 0; n < N; n++)
            cin >> posicionBuzones[n];
        posicionAmigos.assign(M, 0);
        for (int m = 0; m < M; m++)
            cin >> posicionAmigos[m];
        cout << minimoK(posicionBuzones, posicionAmigos) << '\n';
    }
    return 0;
}
```

Farolas y ladrones

Teoría

- Algoritmos voraces
- Ordenamiento
- Método rápido de entrada y salida de datos

Solución

Guardamos de cada farola la coordenada en la que empieza su rango de iluminación y en la que termina, que calculamos a partir de su radio y posición. Ordenamos estas coordenadas de izquierda a derecha, al igual que las posiciones de los vecinos. Vamos vecino a vecino de izquierda a derecha viendo si las farolas ya encendidas cubren su coordenada y, si no, encendemos la farola que termine más a la derecha dentro de las que lo cubren, ya que así maximizamos la probabilidad de que esta farola cubra a los siguientes vecinos. Vamos actualizando el número de farolas encendidas y la máxima coordenada cubierta a la derecha con cada farola que encendemos. Si hay un vecino al que no se lo puede iluminar con ninguna farola, retornamos -1 como forma de decir que es imposible cubrirlos a todos.

Código

C++

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

int farolasEncendidas(vector<int> & posiciones, vector<pair<int, int>> &
↪  coordenadas) {
    sort(coordenadas.begin(), coordenadas.end());
    sort(posiciones.begin(), posiciones.end());
    int vecinos = (int) posiciones.size();
    int farolas = (int) coordenadas.size();
    int respuesta = 0, cubierto = -1, coordenadaMaxima;
    for (int i = 0; i < vecinos; i++) {
        if (posiciones[i] <= cubierto) {
            // ya hay una farola que cubre al vecino
            continue;
        }
        coordenadaMaxima = -1;

        for (int j = 0; j < farolas; j++) {
            if (coordenadas[j].first <= posiciones[i] && posiciones[i] <=
            ↪  coordenadas[j].second) {
                // la farola j cubre al vecino i
                coordenadaMaxima = max(coordenadaMaxima, coordenadas[j].second);
            }
        }
```

```
        // si es imposible cubrir a este vecino, retornamos -1
        if (coordenadaMaxima == -1)
            return -1;

        // si no, incrementamos el contador y actualizamos el terreno cubierto
        respuesta++;
        cubierto = coordenadaMaxima;
    }
    return respuesta;
}

int main() {
    // estas dos líneas sirven para hacer I/O más rápido con cin y cout
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int t, farolas, respuesta, vecinos;
    vector<int> posiciones, posFarola, radioFarola;
    vector<pair<int, int>> coordenadas;
    cin >> t;
    while(t--) {
        cin >> vecinos >> farolas;
        posiciones = vector<int>(vecinos);
        for (int i = 0; i < vecinos; i++)
            cin >> posiciones[i];
        posFarola = vector<int>(farolas);
        radioFarola = vector<int>(farolas);
        coordenadas = vector<pair<int, int>>(farolas);
        for (int i = 0; i < farolas; i++) {
            cin >> posFarola[i] >> radioFarola[i];
            coordenadas[i] = make_pair(posFarola[i] - radioFarola[i], posFarola[i]
            ↪  + radioFarola[i]);
        }
        respuesta = farolasEncendidas(posiciones, coordenadas);
        if (respuesta != -1) cout << respuesta << '\n';
        else cout << "imposible\n";
    }
    return 0;
}
```

Pintando piedras

Teoría

- Programación dinámica
- Recursión múltiple
- Combinatoria
- Aritmética modular

Solución

Para elaborar la solución, hay que tener en cuenta tres casos distintos:

- Hay dos piedras seguidas ya pintadas del mismo color (respuesta = 0).
- Las N piedras están sin pintar (respuesta = $colores * colores^{N-1}$).
- El resto de casos, en los que hay al menos una piedra pintada y una posibilidad de colorear.

En el tercer caso, separaremos las piedras en tres grupos. La estructura de la secuencia es (0..0) (piedras intermedias) (0..0). Si hay x ceros al principio, hay $(colores - 1)^x$ formas de pintarlos. Lo mismo para los ceros del final. Por tanto, nuestra respuesta inicial es el producto de estas dos formas de colorear el principio y el final.

Si la secuencia de piedras intermedias tiene longitud 1, retornamos este valor como respuesta final, ya que solo hay una forma de pintar la secuencia intermedia.

Si no, vamos piedra a piedra de la secuencia intermedia. Si es el final de una secuencia de ceros, calculamos las formas que hay de pintarla. Si no, incrementamos el contador de ceros.

El número de formas de colorear una secuencia de ceros depende de si las dos piedras que la rodean son del mismo color o no. Creamos dos funciones para esto, donde `mismas()` retorna las formas de colorear una secuencia si los dos bordes son iguales y `diferentes()` en caso contrario.

Programaremos estas dos funciones con recursión múltiple. Empecemos con los casos base. Si hay un solo cero en la secuencia y las piedras son las mismas, este cero se puede pintar con cualquiera de los colores menos el del borde. Por lo tanto, la respuesta es $colores - 1$. Si las piedras son de colores diferentes, este cero se puede pintar con cualquiera de los colores menos los colores del borde. Por lo tanto, la respuesta es $colores - 2$.

Si hay x ceros consecutivos y las piedras del borde son las mismas, podemos pintar el último cero con cualquier color menos el del borde, dejando $x - 1$ ceros con un borde distinto. Por lo tanto, `mismas(x) = diferentes(x-1)*(colores-1)`.

Si hay x ceros consecutivos y las piedras del borde son diferentes, podemos pintar la última piedra del color de la primera, por lo que habría `mismas(x-1)` combinaciones, o de uno que no sea de ninguno de los bordes, por lo que habría `diferentes(x-1)*(colores-2)` combinaciones. Por lo tanto, `diferentes(x) = diferentes(x-1)*(colores-2)+mismas(x-1)`.

Finalmente, solo nos faltaría incluir módulos y calcular las potencias con el método rápido.

Código

C++

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
typedef long long int ll; // así nos ahorramos escribir long long int

ll mod = 1e9 + 7;

ll exponenciacion (ll base, int potencia) {
    if (potencia == 0)
        return 1;
    ll x = exponenciacion(base, potencia >> 1);
    if (potencia & 1)
        return (((x*x) % mod)*base)%mod;
    else
        return (x*x)%mod;
}

ll diferentes(int numeroCeros, ll colores, vector<ll> & DPdiferentes, vector<ll> &
↪ DPmismas); // para poder hacer la recusión múltiple

ll mismas(int numeroCeros, int colores, vector<ll> & DPmismas, vector<ll> &
↪ DPdiferentes) {
    if (DPmismas[numeroCeros-1] != -1)
        return DPmismas[numeroCeros-1];
    return DPmismas[numeroCeros-1] = (diferentes(numeroCeros-1, colores,
    ↪ DPdiferentes, DPmismas)*(colores-1))%mod;
}

ll diferentes(int numeroCeros, ll colores, vector<ll> & DPdiferentes, vector<ll> &
↪ DPmismas) {
    if (DPdiferentes[numeroCeros-1] != -1)
        return DPdiferentes[numeroCeros-1];
    return DPdiferentes[numeroCeros-1] = (diferentes(numeroCeros-1, colores,
    ↪ DPdiferentes, DPmismas)*(colores-2)
            + mismas(numeroCeros-1, colores, DPmismas, DPdiferentes))%mod;
}

ll formasDePintar(vector<int> & piedras, int colores) {
    int N = (int) piedras.size();
    // comprobamos que no haya dos piedras seguidas del mismo color (no igual a 0)
    ↪ -> respuesta = 0
    for (int i = 1; i < N; i++) {
        if (piedras[i] == piedras[i-1] && piedras[i])
            return 0;
    }
```

```cpp
    // encontramos la primera piedra que no sea un cero empezando por la izquierda
    int primera = -1;
    for (int i = 0; i < N; i++) {
        if (piedras[i]) {
            primera = i;
            break;
        }
    }

    if (primera == -1) {
        // comprobamos si son todo ceros las piedras -> respuesta =
        ↪  colores*((colores-1)^(n-1))
        return (exponenciacion((ll) (colores-1), ((ll) (N-1)))*colores) % mod;
    }

    // encontramos la primera piedra que no sea un cero empezando por la derecha
    int ultima = -1;

    for (int i = N-1; i >= 0; i--) {
        if (piedras[i]) {
            ultima = i;
            break;
        }
    }

    ll respuesta = 1;

    respuesta *= exponenciacion((ll) (colores-1), (ll) primera); // formas de
    ↪  colorear los ceros del principio
    respuesta %= mod;
    respuesta *= exponenciacion((ll) (colores-1), (ll) N-ultima-1); // formas de
    ↪  colorear los ceros del final
    respuesta %= mod;

    if (primera == ultima) {
        // solo hay una piedra aparte de los ceros del principio y final
        return respuesta;
    }

    // las piedras del medio se componen de una secuencia de colores ya puestos y
    ↪  de grupos de ceros
    // solo hay una manera de colorear las piedras ya coloreadas (el color en el
    ↪  que vienen)
    // sin embargo, las formas de colorear una secuencia de ceros depende de si
    ↪  las piedras de color que la rodean
    // son del mismo color o colores diferentes

    int ultimoColorVisto = piedras[primera], cerosSeguidos = 0;
    vector<ll> DPmismas(N, -1), DPdiferentes(N, -1);
    DPmismas[0] = (ll) (colores-1); // caso base mismas
    DPdiferentes[0] = (ll) (colores-2); // caso base diferentes
```

```
    for (int i = primera; i <= ultima; i++) {
        if (piedras[i]) {
            if (cerosSeguidos) {
                if (piedras[i] == ultimoColorVisto)
                    respuesta *= mismas(cerosSeguidos, colores, DPmismas,
                    ↪ DPdiferentes);
                else
                    respuesta *= diferentes(cerosSeguidos, colores, DPdiferentes,
                    ↪ DPmismas);
                respuesta %= mod;
                cerosSeguidos = 0;
            }
            ultimoColorVisto = piedras[i];
        } else cerosSeguidos++;
    }

    return respuesta;
}


int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int t, numeroPiedras, colores;
    vector<int> piedras;
    cin >> t;
    while(t--) {
        cin >> numeroPiedras >> colores;
        piedras = vector<int>(numeroPiedras);
        for (int i = 0; i < numeroPiedras; i++)
            cin >> piedras[i];
        cout << formasDePintar(piedras, colores) << '\n';
    }
    return 0;
}
```

Cave

Teoría

- Bisección

Solución

Para este problema, iteraremos sobre las N puertas, primero encontrando la dirección correcta del interruptor y luego el interruptor que abre la i-ésima puerta.

Para eso, primero probaremos con todos los interruptores de las puertas restantes colocados hacia abajo. Si con esta combinación se abre la i-ésima puerta, sabremos que su interruptor va hacia abajo. Si no, sabremos que va hacia arriba.

Sabiendo esto, hacemos bisección para encontrar cuál de los interruptores fue el que abrió la puerta. Para cada intervalo lo a hi, encendemos los interruptores entre lo y mid cuya dirección final aún no conozcamos. Probamos esta combinación. Comparamos esto con lo que pasó al dejar apagados todos los del intervalo. Si no hay diferencia respecto a la i-ésima puerta, sabemos que el interruptor se encuentra en la segunda mitad del intervalo, ya que estos se mantuvieron apagados. Por el contrario, si hay diferencia, sabemos que el interruptor se encuentra en la primera mitad y su dirección corresponderá a la combinación que no bloqueara la i-ésima puerta.

Hacemos esto para todas las puertas hasta lograr abrirlas todas. Entonces sabremos la combinación final y qué puerta corresponde con cada interruptor.

Código

C++

```cpp
#include "cave.h"
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

void flip(int & a) {
        // flip enciende un interruptor apagado y apaga un interruptor encendido
        a = 1 - a;
}
```

```cpp
void exploreCave(int N) {
        int direccionFinal[N], conexionFinal[N], direccionInterruptores[N];
        memset(direccionFinal, -1, sizeof(direccionFinal)); // asignamos -1 a todas las posiciones
        memset(conexionFinal, -1, sizeof(conexionFinal)); // asignamos -1 a todas las posiciones
        memset(direccionInterruptores, 0, sizeof(direccionInterruptores)); // asignamos 0 a todas las posiciones
        int puertasAbiertas1, puertasAbiertas2, lo, hi, mid;
        if (N == 1) {
                // solo hay una puerta -> hacemos una llamada para ver la dirección del interruptor
                puertasAbiertas1 = tryCombination(direccionInterruptores);
                if (puertasAbiertas1 == 0) direccionFinal[0] = 1;
                else direccionFinal[0] = 0;
                conexionFinal[0] = 0;
                answer(direccionFinal, conexionFinal);
                return;
        }

        pair<int, int> respuesta = make_pair(0, 0);
        for (int i = 0; i < N; i++) {
                // primero, averiguaremos la dirección del interruptor que abre la puerta i
                puertasAbiertas1 = tryCombination(direccionInterruptores);

                // nos aseguramos de que la variable guarde el número de puertas abiertas
                if (puertasAbiertas1 == -1)
                        puertasAbiertas1 = N;

                // hacemos bisección para encontrar qué interruptor abre la i'ésima puerta
                lo = 0;
                hi = N-1;
                while(lo < hi) {
                        mid = lo + (hi-lo)/2;
                        for (int j = lo; j <= mid; j++) {
                                // alteramos la dirección de la primera mitad de interruptores cuya dirección final no es conocida
                                if (conexionFinal[j] == -1)
                                        flip(direccionInterruptores[j]);
                        }
                        puertasAbiertas2 = tryCombination(direccionInterruptores);

                        for (int j = lo; j <= mid; j++) {
                                // los devolvemos a su estado original tras la prueba
                                if (conexionFinal[j] == -1)
                                        flip(direccionInterruptores[j]);
                        }
```

```
                        // nos aseguramos de que la variable guarde el número de
                        ↪  puertas abiertas
                        if (puertasAbiertas2 == -1)
                                puertasAbiertas2 = N;

                        if (puertasAbiertas2 > i) {
                                // la puerta i'ésima se abrió con la nueva
                                ↪  combinación
                                if (puertasAbiertas1 <= i) {
                                        // la puerta i'ésima no se abrió con la
                                        ↪  vieja combinación
                                        // el interruptor está en la primera mitad
                                        // hay que cambiar su dirección
                                        hi = mid;
                                        respuesta = make_pair(mid,
                                        ↪  1-direccionInterruptores[mid]);
                                } else {
                                        // la puerta i'ésima se abrió con la vieja
                                        ↪  combinación
                                        // el interruptor está en la segunda mitad
                                        // no hay que cambiar la dirección
                                        lo = mid+1;
                                        respuesta = make_pair(mid+1,
                                        ↪  direccionInterruptores[mid+1]);
                                }
                        } else {
                                // la puerta i'ésima no abrió con la nueva
                                ↪  combinación
                                if (puertasAbiertas1 <= i) {
                                        // la puerta i'ésima se abrió con la vieja
                                        ↪  combinación
                                        // el interruptor está en la primera mitad
                                        // hay que cambiar su dirección
                                        lo = mid+1;
                                        respuesta = make_pair(mid+1,
                                        ↪  1-direccionInterruptores[mid+1]);
                                }
                                else {
                                        // la puerta i'ésima no se abrió con la
                                        ↪  vieja combinación
                                        // el interruptor está en la segunda mitad
                                        // no hay que cambiar la dirección
                                        hi = mid;
                                        respuesta = make_pair(mid,
                                        ↪  direccionInterruptores[mid]);
                                }
                        }
                }
                conexionFinal[respuesta.first] = i;
                direccionFinal[respuesta.first] = respuesta.second;
                direccionInterruptores[respuesta.first] = respuesta.second;
        }
        answer(direccionFinal, conexionFinal);
}
```

Scales

Teoría

- Lógica
- Árboles de decisión

Solución

La solución consiste en un árbol de decisión algo tedioso de programar. Por no ocupar medio documento aquí con un programa, os pongo un link a la solución de Abhinav Kulshrestha. La función que el juez os pide que implementéis va de la línea 130 a la 375.

Esta solución se basa en que $6! = 720$ y $3^6 = 729$. Por lo tanto, hacen falta 6 preguntas que reduzcan entre 3 las posibilidades para resolver el problema.

https://pastebin.com/bS3HiBsL

Sabotage

Teoría

- Método de bisección
- Programación dinámica; algoritmo de Kadane (problema clásico: max 1D subarray sum)

Solución

Para resolver este problema, usaremos el método de bisección que encuentra mínimos. Trataremos de encontrar la media más baja posible de esta forma.

Por cómo se calcula la media de una lista de números (la suma dividida entre la longitud de la lista), podemos observar que los signos de la suma y la media siempre coinciden. Para probar un número avg y ver si se puede quitar elementos de la lista de tal manera que quede una media igual o inferior a avg, restamos avg de todos los elementos de la lista y quitamos la ventana con la máxima suma. Es posible dejar una media igual o inferior a avg si la media de los elementos restantes es menor o igual a 0, que es lo que comprueba la condición final.

Código

C++

```cpp
#include <fstream>
#include <vector>
#include <iomanip>
using namespace std;

int round3dp(double num) {
        return (int) (1000*num + 0.5);
}

bool posible(double avg, vector<int> & milk, int sumaTotal) {
        double sumaMayor, sumaActual;
        sumaMayor = milk[1] - avg;
        sumaActual = max(milk[1] - avg, 0.0);
        for (int i = 2; i < (int) milk.size()-1; i++) {
                sumaActual += milk[i];
                sumaActual -= avg;
                sumaMayor = max(sumaMayor, sumaActual);
                sumaActual = max(sumaActual, 0.0);
        }
        return sumaMayor >= sumaTotal - avg * ((int) milk.size());
}
```

```cpp
double minimoK(vector<int> & milk) {
        double lo = 0, hi = 10000, mid, total = 0, k;
        for (int i = 0; i < (int) milk.size(); i++)
                total += milk[i];
        while(round3dp(lo) < round3dp(hi)) {
                mid = (lo+hi)/2;
                if (posible(mid, milk, total))
                        hi = k = mid;
                else
                        lo = mid;
        }
        return k;
}

int main() {
        ifstream fin;
        fin.open("sabotage.in");
        ofstream fout;
        fout.open("sabotage.out");
        int N;
        fin >> N;
        vector<int> milk(N);
        for (int i = 0; i < N; i++) {
                fin >> milk[i];
        }
        fin.close();
        fout << fixed;
        fout << setprecision(3) << minimoK(milk) << '\n';
        fout.close();
        return 0;
}
```

Pintando piedras 2

Teoría

- Combinatoria
- Programación dinámica
- Aritmética modular

Solución

Conseguimos la solución de forma recursiva. Hay tres secciones dentro del programa. Usaremos i para referirnos a la i-ésima piedra por la izquierda ($i = 0$ significa el caso base de aún no haber llegado a la primera piedra).

- $0 \leq i < k$: no puede haber k piedras seguidas. Hay una sola respuesta posible cuando $i = 0$ y las formas de pintar i piedras con c colores son c multiplicado por las formas de pintar $i - 1$ piedras, ya que hay c colores con los que pintar esta última.
- $i = k$: hay c^k formas de pintar las piedras. Sin embargo, para cada uno de los c colores, hay una combinación de k piedras de ese color seguidas. Por lo tanto, hay $c^k - c$ formas de pintar las piedras.
- $i > k$: para cada combinación con $i - 1$ piedras, hay c colores que podemos añadir, dando un total de $c * formas[i - 1]$. A este total hay que restarle el número de combinaciones que terminan con k piedras iguales. Este número de combinaciones es igual a $formas[i - k] * (c - 1)$. Por lo tanto, $formas[i] = c * formas[i - 1] - formas[i - k] * (c - 1)$.

Durante todo el programa, iremos añadiendo módulos para evitar overflow. Además guardaremos las soluciones de los subproblemas para no calcular lo mismo varias veces.

Código

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;
typedef long long int ll;

ll respuesta (int n, ll colores, int k, ll mod) {
    vector<ll> DP(n+1);
    DP[0] = 1;
    for (int i = 1; i < k; i++)
        DP[i] = (DP[i-1]*colores)%mod;

    DP[k] = (((DP[k-1]*colores)%mod) - (colores % mod) + mod)%mod;

    for (int i = k+1; i <= n; i++)
        DP[i] = (((DP[i-1]*colores)%mod) - ((DP[i-k]*(colores-1))%mod) + mod)%mod;

    return DP[n];
}
```

```cpp
int main() {
    int t, n, c, k, p;
    cin >> t;
    while(t--) {
        cin >> n >> c >> k >> p;
        cout << respuesta(n, c, k, (ll) p) << '\n';
    }
    return 0;
}
```