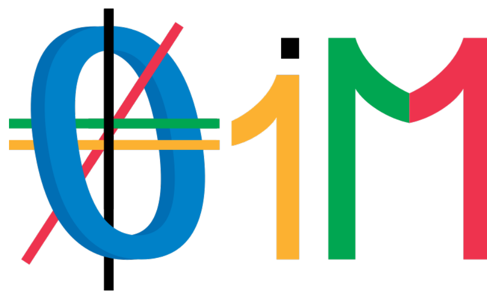
Olimpiada Informática Española

Regional de Madrid
2021



Explicación de las soluciones

12 de febrero de 2021

Those hours of practice, and failure, are a necessary part of the learning process
Gina Sipley

12 de febrero de 2021

Listado de problemas

A ¿En qué volumen? **3**

B Codificación límite **5**

C ¡Se ha colado! **7**

D Comienza la temporada **9**

E Pepe Casanova **13**

F Puntos de silla **21**

G Ratones en un laberinto **23**

H ¿Cuándo seré rico? **27**

I El carpintero Ebanisto **29**

Autor de las explicaciones:

- Alberto Maurel Serrano

This work is licensed under a Creative Commons “Attribution-NonCommercial-ShareAlike 4.0 International” license.

I Olimpiada Informática de Madrid

Soluciones

A - ¿En qué volumen?

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=595

Explicación

Para cada problema, tenemos que ver a qué volumen pertenece. Para ello, simplemente tenemos que dividir el número de problema entre 100, y quedarnos con la parte entera. Por ejemplo:

$$306 \;/\; 100 = 3,06 \Rightarrow \text{volumen } 3$$

La opción más sencilla para implementar esto es usar la división entera. En C++, al dividir dos enteros nos sale simplemente la parte entera del resultado. En Python, el operador para la división entera es //.

Código

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

int main() {
    int nc;
    cin >> nc;
    for (int z = 0; z < nc; ++z) {
        int n;
        cin >> n;
        cout << n / 100 << '\n';
    }
    return 0;
}
```

I Olimpiada Informática de Madrid

Soluciones

B - Codificación límite

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=596

Explicación

Vamos a fijarnos en los distintos tipos de nodos que hay en el árbol. Utilizamos la siguiente notación:

- r: el nodo en el que nos encontramos
- i: el subárbol izquierdo (hijo izquierdo)
- d: el subárbol derecho (hijo derecho)
- . para el subárbol vacío
- x para elementos indeterminados **distintos** del subárbol vacío
- y para elementos indeterminados cualesquiera

Tenemos 4 casos:

- Tenemos un árbol con **dos hijos no vacíos**. En este caso, si estamos en la letra r, el mensaje será de la forma:

  $$r\ i\ d$$

  donde, al ser el hijo izquierdo no vacío, no contendrá un punto en la primera posición. Por lo tanto, será algo de la forma:

  $$r\ xyyyyy$$

- Tenemos un árbol con **el hijo izquierdo vacío y el derecho no**. En este caso, si estamos en la letra r, el mensaje será de la forma:

  $$r\ .\ d$$

  Al ser el hijo derecho no vacío, su primer elemento será distinto al árbol vacío. Por lo tanto, será algo de la forma:

  $$r\ .\ xyyyyy$$

- Tenemos un árbol con **el hijo izquierdo no vacío y el derecho sí**. El mensaje será de la forma:

  $$r\ i\ .$$

  De forma análoga al caso anterior, la representación será algo de la forma:

  $$r\ xyyyyy\ .$$

- Tenemos un árbol con **ambos hijos vacíos**. En este caso, se trata de un nodo hoja, los que estamos buscando. Será algo de la forma:

  $$r\ .\ .$$

Por lo tanto, simplemente tenemos que buscar en nuestra cadena una letra seguida de dos puntos: estos serán los nodos hoja.

Código

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

bool res() {
    string msgCifrado;
    cin >> msgCifrado;
    if (!cin) return false;

    int idx = 0;

    while (idx + 2 < msgCifrado.size()) {
        if (msgCifrado[idx] != '.' && msgCifrado[idx + 1] == '.' &&
            msgCifrado[idx + 2] == '.') {

            cout << msgCifrado[idx];
            idx += 2;
        }
        else ++idx;
    }

    cout << '\n';
    return true;
}

int main() {
    while (res()) {}
    return 0;
}
```

I Olimpiada Informática de Madrid
Soluciones

C - ¡Se ha colado!

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=597

Explicación

En primer lugar, vamos a ver qué significa "haberse colado". Formalmente, si la secuencia de números que representan los tickets es a_1, a_2, a_3, ... , a_n queremos encontrar aquellos números tales que $a_i > a_j$, siendo $i < j$. Esto significará que en la cola encontramos antes a alguien que ha llegado después (tiene un ticket mayor), y por lo tanto se ha colado.

Además, las constantes del enunciado ($n \leq 500.000$) nos indican que la complejidad esperada de nuestra solución está en $O(n)$. Es decir, el número de operaciones que realizamos es lineal (un múltiplo) respecto al número n. No nos vale con, para cada ticket, comprobar con todos los sucesivos si es menor que ellos.

Para comprobar esto, vamos a recorrer el vector de derecha a izquierda. El último elemento no puede haberse colado. Para los siguientes elementos, tendremos la certeza de que se han colado si hay a su derecha algún número menor. Por ello, es suficiente con almacenar el menor encontrado hasta el momento. El algoritmo será por tanto:

- Recorremos los números de derecha a izquierda
- Si el número actual es menor a los encontrados hasta el momento, no se ha colado. Actualizamos el mínimo encontrado.
- Si el número actual es mayor al mínimo encontrado hasta el momento, se ha colado. Incrementamos en 1 el número de personas expulsadas.

Código

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

int main() {
    int nc;
    cin >> nc;
    for (int z = 0; z < nc; ++z) {
        int n;
        cin >> n;

        vector<int> v(n);
        for (int& x : v) cin >> x;

        int minimo = v[n - 1];
        int expulsados = 0;
```

```
1            for (int i = n - 2; i >= 0; --i) {
2                if (v[i] < minimo)
3                    minimo = v[i];
4                else
5                    ++expulsados;
6            }
7            cout << expulsados << '\n';
8        }
9
10       return 0;
11   }
```

D - Comienza la temporada

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=598

Explicación

Vamos a resolver este problema mediante un **algoritmo voraz**. Los algoritmos voraces son algoritmos que en cada paso hacen una elección que se mantendrá el resto de la ejecución, de forma que dicha elección conduce a una solución óptima.

En primer lugar, ordenamos a los jugadores y a las equipaciones de menor a mayor por talla. Para cada jugador, comenzando por el de talla menor, hacemos:

- Si hay una equipación de su misma talla libre, se la asignamos.
- Si no hay una equipación libre de su misma talla, pero sí de la talla siguiente, se la asignamos.
- Si no hay ninguna equipación que cumpla esto, le compramos una equipación.

Este algoritmo funciona porque en cada paso, como decíamos antes, hacemos una elección óptima. Esto se debe a que:

- Si hay una equipación de su misma talla libre (t), puede que no haya ningún jugador posterior que tenga la misma talla que este. Por ello, si no se la asignamos estaríamos desperdiciando esta equipación. Y si hay otro jugador con la misma talla, podemos intercambiar con él la equipación que tenga sin problemas, por lo que decidimos dársela a este (y ya veremos luego si queda alguna equipación para el otro).

  Aún en el caso de tener una equipación de la talla siguiente ($t + 1$), preferimos asignarle una equipación de la talla t. La razón es que la equipación de la talla t solo nos vale para los jugadores con talla t, puesto que al estar procesando a los jugadores por orden de talla creciente, ya hemos terminado con los de talla $t - 1$. La equipación de talla $t + 1$ sin embargo nos vale tanto para jugadores de talla t como de talla $t + 1$, por lo que preferimos reservárnoslas.
- Si no hay una equipación libre de su misma talla, pero sí una de la talla siguiente ($t + 1$), se la asignamos. Puede que esta equipación la pudiésemos utilizar con un jugador de talla $t+1$. Pero, ese jugador podrá también utilizar equipaciones de talla $t + 2$. Y en el caso de que el segundo jugador se quede sin equipación, tendríamos que comprar igualmente una.
- Si no hay ninguna equipación que cumpla esto, no le podemos asignar una equipación de ninguna forma.

La primera solución es una simple aplicación de esto. Como ordenamos las listas de tallas y las recorremos, tiene complejidad temporal en $O(s \log s)$, siendo $s = \max(m, n)$.

Pero podemos “evitar” realizar la ordenación. El motivo es que se nos dice que las tallas de las equipaciones están entre 1 y 100. Por ello, podemos “comprimir” la lista de jugadores y equipaciones, indicando para cada una de las tallas cuántos tenemos de cada uno. Posteriormente ejecutamos el algoritmo anterior, pero ahora asignando las equipaciones en bloque, no de una en una. Esto es lo que hace el segundo código, con complejidad en $O(m + n + 100)$.

Nota: en este segundo caso, también estamos ordenando las tallas, aunque sin usar operaciones de comparación. Esto es posible gracias a que el rango de tallas está lo suficientemente acotado. Esta técnica de ordenación se llama *counting sort*.

I Olimpiada Informática de Madrid

Soluciones

Código Solución 1

```cpp
#include <iostream>
#include <vector>
#include <algorithm> //Para el sort
using namespace std;

bool res() {
    int n, m;
    cin >> n >> m;
    if (!cin) return false;

    vector<int> jugadores(n), equipaciones(m);
    for (int& x : jugadores) cin >> x;
    for (int& x : equipaciones) cin >> x;
    //Ordenamos las equipaciones y los jugadores por tamaño, de menor a mayor
    sort(jugadores.begin(), jugadores.end());
    sort(equipaciones.begin(), equipaciones.end());

    int idxJugador = 0, idxEquipacion = 0, emparejados = 0;

    while(idxJugador < jugadores.size()&&idxEquipacion < equipaciones.size()){
        //Podemos darle a este jugador esta equipación, se la damos
        if (jugadores[idxJugador] == equipaciones[idxEquipacion] ||
            jugadores[idxJugador] + 1 == equipaciones[idxEquipacion]) {
            ++idxJugador;
            ++idxEquipacion;
            ++emparejados;
        }

        //Tenemos que comprarle una equipación más pequeña
        else if (jugadores[idxJugador] + 1 < equipaciones[idxEquipacion])
            ++idxJugador;

        //Esta equipación no nos valdrá para ningún jugador
        else
            ++idxEquipacion;
    }

    //Compramos equipaciones para los que no tienen
    cout << n - emparejados << '\n';
    return true;
}

int main() {
    while (res()) {}
    return 0;
}
```

I Olimpiada Informática de Madrid

Soluciones

Código Solución 2

```
#include <iostream>
#include <vector>
#include <algorithm> //Para el sort
using namespace std;

bool res() {
    int n, m;
    cin >> n >> m;
    if (!cin) return false;
    vector<int> jugadores(102, 0), equipaciones(102, 0);

    for (int i = 0; i < n; ++i) {
        int talla;
        cin >> talla;
        ++jugadores[talla];
    }
    for (int i = 0; i < m; ++i) {
        int talla;
        cin >> talla;
        ++equipaciones[talla];
    }

    int tAct = 1;
    int noEmparejados = 0;

    while (tAct <= 100) {
        int equipAsignadas = min(jugadores[tAct], equipaciones[tAct]);
        jugadores[tAct] -= equipAsignadas;
        equipaciones[tAct] -= equipAsignadas;

        if (jugadores[tAct] > 0) {
            equipAsignadas = min(jugadores[tAct], equipaciones[tAct + 1]);
            jugadores[tAct] -= equipAsignadas;
            equipaciones[tAct + 1] -= equipAsignadas;
        }
        noEmparejados += jugadores[tAct];
        ++tallaActual;
    }
    cout << noEmparejados << '\n';
    return true;
}

int main() {
    while (res()) {}
    return 0;
}
```

I Olimpiada Informática de Madrid

Soluciones

E - Pepe Casanova

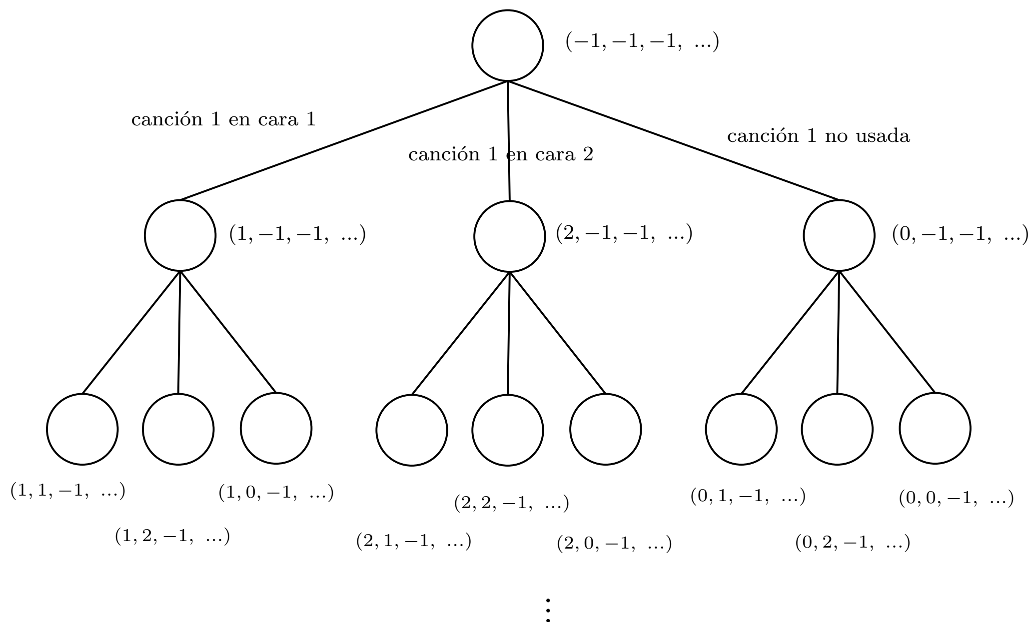
Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=599

Explicación

Para cada una de las canciones que tenemos, tenemos que decidir si:

- La introducimos en la cara 1 de la cinta.
- La introducimos en la cara 2 de la cinta.
- No la introducimos en la cinta.

Si resolvemos el problema probando todas las combinaciones distintas, podemos ver el espacio de soluciones como un árbol. Al principio, tenemos un único nodo, la solución vacía. Cuando consideramos las tres posibilidades para la primera canción, vemos que aparecen 3 ramas: en la que introducimos la canción en la cara 1, en la que lo hacemos en la cara 2 y en la que no la grabamos. Y así sucesivamente. Si representamos con 1 cuando metemos la canción i en la cara 1, 2 cuando la metemos en la cara 2, 0 cuando no la metemos y -1 cuando aún no hemos decidido tenemos:



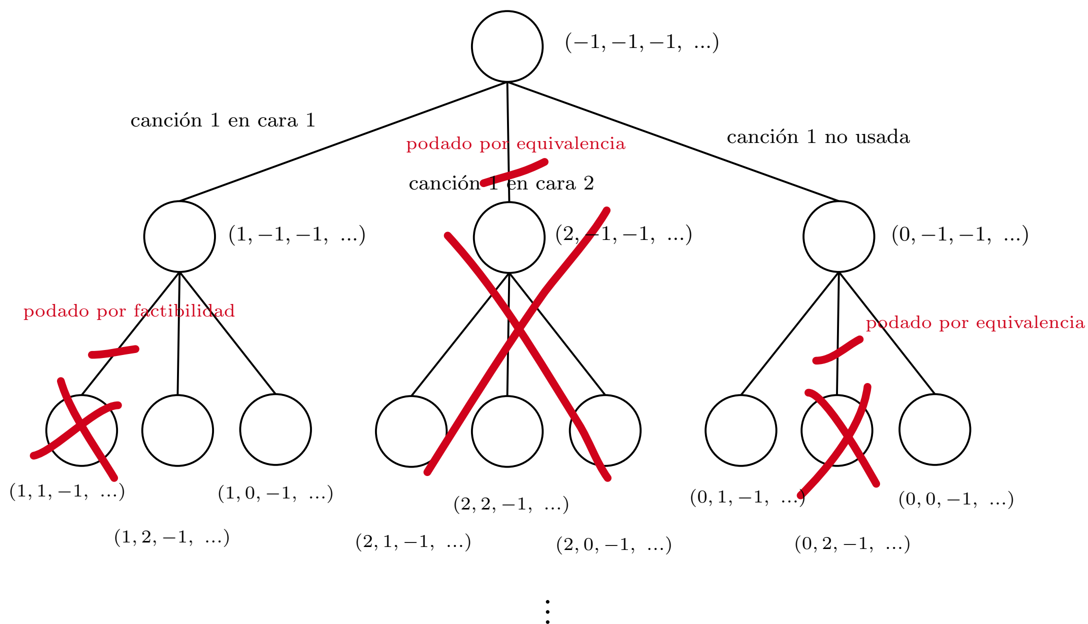
Si probamos todas las posibilidades directamente, tendremos que probar 3^n combinaciones distintas. Dado que $n \leq 30$, obtendremos un TLE.

Para evitar esto, vamos a utilizar la técnica de **vuelta atrás**. Esta técnica consiste en ir recorriendo el árbol de todas las combinaciones distintas, y si en un momento vemos que no va a ser posible que ninguna solución de la rama actual mejore la mejor solución encontrada hasta el momento, podamos la rama y no la exploramos. De esta forma, si la poda es buena, reducimos en gran medida el número de combinaciones exploradas.

Hay numerosas ramas que no necesitamos recorrer, y que podemos “podar” en base a la información que tenemos. Por ejemplo:

- **Por factibilidad**: en una cara no podremos introducir una canción si no cabe en el hueco restante. Si mantenemos los minutos que hemos ocupado en cada cara, podremos podar aquellas ramas en las que al ir a introducir la canción en una cara no nos quepa.
- **Por soluciones equivalentes:** al inicio, cuando aún no hemos introducido ninguna canción, ambas caras son equivalentes. Y en general, si ambas caras tienen el mismo hueco restante, son equivalentes, y meter la canción en una u otra cara dará lugar a las mismas soluciones, con la cara 1 y 2 intercambiadas.
- **Por optimalidad:** si ya hemos encontrado una solución, puede darse el caso de que ninguna de las soluciones de la rama que estamos explorando, a pesar de ser factibles y diferentes a las anteriores, vaya a mejorar a la mejor solución ya encontrada. Si obtenemos una cota de la máxima puntuación que podemos obtener por esta rama, podremos podarla si es inferior a la mejor solución encontrada.

Con estas mejoras, nos quedará algo como:



⋮

Solo nos queda ahora ver cómo hacemos el cálculo de la cota. Cuanto más ajustada sea la cota, mejor podremos podar, pero evidentemente, **la cota ha de ser siempre mayor o igual que la máxima puntuación que podemos obtener en la rama**. En caso contrario, estaríamos podando soluciones que quizás sí que mejorarían la que tenemos.

Además, existe un *tradeoff* entre ajustar más la cota y calcularla más rápidamente (tenemos que calcularla de forma eficiente, si no el tiempo que nos ahorramos explorando el árbol de posibilidades lo gastaremos en calcular la cota y no mejoraremos nada).

Tenemos 2 propuestas de cotas. La **primera** consiste simplemente en ver cuál es la puntuación que nos pueden aportar todas las canciones restantes. En el mejor de los casos, cabrán todas las canciones en la cinta, y la cota será igual a la puntuación máxima obtenida en la rama. Y en caso de que no quepan, la cota será superior a la solución máxima, por lo que será una cota válida.

Para calcular esta cota de forma eficiente, simplemente tendremos que precalcular la suma de las puntuaciones de las canciones en el intervalo $[i, n]$. De esta forma, para la rama actual que está decidiendo sobre la canción i, podremos calcular la cota como :

$$cota = puntuación\ actual + puntuación\ en\ [i, n]$$

en $O(1)$, con un precálculo en $O(n)$.

La **segunda** cota es un poco más ajustada, pero a cambio es menos eficiente de calcular. Este problema en realidad es un "*pseudoproblema* de la mochila", en el que en vez de tener una mochila tenemos dos.

Una versión relajada del problema de la mochila consiste en considerar que se pueden partir los objetos. De esta forma, el objeto partido aportará un valor proporcional a la porción partida. En este caso, lo que hacemos es que se puedan partir canciones. Si solo nos cabe un tercio de canción, esta nos aportará un tercio de la puntuación.

Esta versión relajada del problema de la mochila sí que tiene una solución óptima eficiente de calcular. Se basa en ordenar los objetos por "densidad de valor", es decir, en orden decreciente de v_i/p_i, siendo v_i el valor del objeto i y p_i el peso del objeto i.

En este problema, lo óptimo es introducir los objetos con mayor densidad de valor hasta que no nos quepa más, posiblemente cortando el último. Es sencillo tener una intuición de por qué esto funciona: por cada unidad de peso gastada, aportará el máximo valor posible.

Es también sencillo ver por qué esta técnica no funciona en el caso de que no se puedan partir los objetos: pueden quedar espacios libres que no podamos aprovechar. Por ejemplo, si podemos cargar 8 kilos, y tenemos un objeto con peso 5 y valor 6, y otros dos con peso 4 y valor 4, nos conviene introducir los dos objetos con peso 4 para obtener un valor total de 8.

¿Cómo podemos convertir esto en una cota?

Lo que vamos a hacer es, llegado a un cierto punto de exploración del árbol, vamos a calcular cuál es la máxima puntuación que podríamos lograr metiendo los objetos con mayor densidad de valor entre los que nos quedan. No tenemos en cuenta que no se pueden partir los objetos, por lo que en caso de que no quepan enteros, la puntuación final será menor. Pero no puede ser mayor, puesto que estamos rellenando todo el hueco que nos queda con los objetos que aportan más valor. Y en el peor de los casos, sí que cabrán dichos objetos sin cortar y la mejor solución y la cota coincidirán. Además, es una cota más ajustada que la anterior porque aquí sí que tenemos en cuenta el hueco que nos queda libre.

Por último, para implementar esto de forma eficiente conviene ordenar los objetos desde el principio por densidad de valor. Así no habrá que, en cada cálculo de la cota, ordenar los que nos quedan.

Nota 1: durante el concurso, se aceptaban ambas soluciones. Sin embargo, **en Acepta el Reto solo se acepta la segunda solución**.

Nota 2: el problema de la mochila es un problema ampliamente conocido. Se puede leer más sobre él en https://es.wikipedia.org/wiki/Problema_de_la_mochila

Nota 3: tampoco es correcto ejecutar el algoritmo de la mochila una vez y sobre los objetos no escogidos, volver a ejecutar el algoritmo de la mochila. Esto se debe a que puede ser que nos convenga perder un poco de valor en la primera mochila a cambio de incrementar el valor de la segunda mochila. Por ejemplo, si tenemos los siguientes 4 objetos:

i	1	2	3	4
v_i	20	15	15	10
p_i	4	5	5	6

Al ejecutar 2 veces el algoritmo de la mochila por separado obtenemos que en la primera mochila tenemos los objetos 1 y 2, y en la segunda el objeto 3. Sin embargo, nos caben todos los objetos en ambas mochilas, si metemos el 1 y 4 en la primera y el 2 y el 3 en la segunda.

Código Solución 1

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
using vi = vector<int>;
using ii = pair<int, int>;
using vii = vector<ii>;

int duracion;
vii canciones;

//Solución 1: usamos como cota la máxima puntuación que podemos conseguir
// con las canciones restantes
void vuelta_atras(int cancionAct, int puntuacionAct, int ocupadoCara1,
                  int ocupadoCara2, int acumulado, int & mejorRes) {
    if (cancionAct == canciones.size()) {
        mejorRes = max(puntuacionAct, mejorRes);
    }
    else {
        //Realizamos la poda. Para ello, comprobamos si con la puntuación
        // que nos pueden aportar las canciones restantes permitiría superar
        // a la mejor puntuación obtenida hasta el momento. Para ello usamos
        // la suma de las puntuaciones de las canciones restantes
        if (puntuacionAct + acumulado <= mejorRes)
            return;

        //Decidimos si metemos o no la canción
        //Meterla en la cara 1 (si es posible)
        if (ocupadoCara1 + canciones[cancionAct].first <= duracion) {
            vuelta_atras(cancionAct + 1, puntuacionAct +
                  canciones[cancionAct].second, ocupadoCara1 +
                  canciones[cancionAct].first, ocupadoCara2,
                  acumulado - canciones[cancionAct].second, mejorRes);
        }
```

```cpp
        //Meterla en la cara 2 (si es posible)
        if (ocupadoCara2 != ocupadoCara1 &&
            ocupadoCara2 + canciones[cancionAct].first <= duracion) {
            vuelta_atras(cancionAct + 1, puntuacionAct +
                  canciones[cancionAct].second, ocupadoCara1,
                  ocupadoCara2 + canciones[cancionAct].first,
                  acumulado - canciones[cancionAct].second, mejorRes);
        }


        //No meterla
        vuelta_atras(cancionAct + 1, puntuacionAct, ocupadoCara1, ocupadoCara2,
              acumulado - canciones[cancionAct].second, mejorRes);
    }
}

int main() {
    int n;
    cin >> n;

    while (n != 0) {
        cin >> duracion;

        //Para cada canción, pares (duración, puntuación)
        canciones.resize(n);
        int acum = 0;

        for (ii& cancion : canciones) {
            cin >> cancion.first >> cancion.second;
            acum += cancion.second;
        }

        int sol = 0;
        vuelta_atras(0, 0, 0, 0, acum, sol);

        cout << sol << '\n';
        cin >> n;
    }

    return 0;
}
```

I Olimpiada Informática de Madrid

Soluciones

Código Solución 2

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

struct tCancion {
    int dur,
        punt;
};

//Operador custom para ordenar tipos definidos por nosotros
struct cmp {
    bool operator()(tCancion const& c1, tCancion const& c2) {
        return c1.punt * c2.dur > c2.punt * c1.dur;
    }
};

int duracion;
vector<tCancion> canciones;
//Solución 2: usamos como cota la de un problema de la mochila voraz (rellenamos
// el tiempo restante con las canciones que tengan un mayor ratio de
//puntuacion/duracion, suponiendo que las canciones se pueden cortar)
void vuelta_atrás(int cancionAct, int puntuacionAct, int ocupadoCara1,
                  int ocupadoCara2, int& mejorRes) {
    if (cancionAct == canciones.size()) {
        mejorRes = max(puntuacionAct, mejorRes);
    }

    else {
        //Realizamos la poda.
        int mejorPuntuacionPosible = puntuacionAct;
        int huecoAct = 2 * duracion - ocupadoCara1 - ocupadoCara2;
        for (int i = cancionAct; i < canciones.size(); ++i) {
            if (canciones[i].dur >= huecoAct) {
                double densidad = ((double) canciones[i].punt) /
                                  canciones[i].dur;
                mejorPuntuacionPosible += ceil(huecoAct * densidad);
                huecoAct = 0;
                break;
            }

            else {
                mejorPuntuacionPosible += canciones[i].punt;
                huecoAct -= canciones[i].dur;
            }
        }
```

```
        if (mejorPuntuacionPosible <= mejorRes)
            return;

        //Decidimos si metemos o no la canción
        //Meterla en la cara 1 (si es posible)
        if (ocupadoCara1 + canciones[cancionAct].dur <= duracion) {
            vuelta_atrás(cancionAct + 1, puntuacionAct +
              canciones[cancionAct].punt, ocupadoCara1 +
              canciones[cancionAct].dur, ocupadoCara2, mejorRes);
        }

        //Meterla en la cara 2 (si es posible)
        if (ocupadoCara2 != ocupadoCara1 &&
            ocupadoCara2 + canciones[cancionAct].dur <= duracion) {
            vuelta_atrás(cancionAct + 1, puntuacionAct +
              canciones[cancionAct].punt, ocupadoCara1,
              ocupadoCara2 + canciones[cancionAct].dur, mejorRes);
        }

        //No meterla
        vuelta_atrás(cancionAct + 1, puntuacionAct, ocupadoCara1,
                     ocupadoCara2, mejorRes);
    }
}

int main() {
    int n;
    cin >> n;

    while (n != 0) {
        cin >> duracion;

        //Para cada canción, pares (duración, puntuación)
        canciones.resize(n);

        for (tCancion& cancion : canciones)
            cin >> cancion.dur >> cancion.punt;

        //Ordenamos las canciones según su ratio puntuación/duración
        sort(canciones.begin(), canciones.end(), cmp());

        int sol = 0;
        vuelta_atrás(0, 0, 0, 0, sol);

        cout << sol << '\n';
        cin >> n;
    }

    return 0;
}
```

I Olimpiada Informática de Madrid

Soluciones

F - Puntos de silla

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=600

Explicación

Tenemos que comprobar para cada punto si es o no un punto de silla, siguiendo las indicaciones del enunciado, de una forma razonablemente eficiente.

Si miramos para cada punto de la matriz en su fila y en su columna si es el mayor o el menor, estaremos empleando un algoritmo con complejidad $O(fcm)$, siendo $m = \max(f,c)$. Esta solución debería entrar en tiempo, dado que $f,\ c \leq 300$.

Sin embargo, podemos hacerlo mejor. Para ello, en primer lugar guardamos cuál es el máximo y mínimo de cada fila y cada columna. Esto podemos hacerlo con complejidad $O(fc)$, puesto que recorremos cada fila y cada columna una vez.

Tras ello, recorremos la matriz otra vez (de nuevo con complejidad $O(fc)$), comprobando si hay algún punto que sea de silla con los datos precalculados. La complejidad por lo tanto nos quedaría $O(fc+fc) = O(fc)$

Código

```
1  #include <iostream>
2  #include <vector>
3  #include <climits>   // Límites de los enteros
4  #include <algorithm> // min/max
5  using namespace std;
6
7  using vi = vector<int>;
8  using vvi = vector<vi>;
9  using ii = pair<int, int>;
10 using vii = vector<ii>;
11
12 int main() {
13     int f, c;
14     cin >> f >> c;
15
16     while (f != 0 || c != 0) {
17         vvi matriz(f, vi(c));
18
19         //Para cada fila, fila[i].first contendrá el mínimo y fila[i].second
20         // el máximo. Ídem para las columnas.
21         //INT_MAX e INT_MIN son los elementos neutros para el mínimo y el
22         // máximo respectivamente
23         vii filas(f, {INT_MAX, INT_MIN}), columnas(c, { INT_MAX, INT_MIN });
24
25         for (vi & fila: matriz)
26             for (int & x: fila)
27                 cin >> x;
```

```
        //Marcamos los elementos que son los menores/mayores de la fila y la
        ↪  columnaº
        for (int i = 0; i < f; ++i) {
            for (int j = 0; j < c; ++j) {
                filas[i].first = min(filas[i].first, matriz[i][j]);
                filas[i].second = max(filas[i].second, matriz[i][j]);
                columnas[j].first = min(columnas[j].first, matriz[i][j]);
                columnas[j].second = max(columnas[j].second, matriz[i][j]);
            }
        }

        bool puntoSilla = false;
        for (int i = 0; i < f && !puntoSilla; ++i) {
            for (int j = 0; j < c; ++j) {
                if (matriz[i][j] == filas[i].first &&
                    matriz[i][j] == columnas[j].second ||
                    matriz[i][j] == filas[i].second &&
                    matriz[i][j] == columnas[j].first) {

                    puntoSilla = true;
                    break;
                }
            }
        }

        if (puntoSilla) cout << "SI\n";
        else cout << "NO\n";

        cin >> f >> c;
    }

    return 0;
}
```

I Olimpiada Informática de Madrid

Soluciones

G - Ratones en un laberinto

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=601

Explicación

La idea de este problema es calcular la menor distancia desde todos los nodos del grafo a uno solo (la salida).

Un truco para hacer esto es invertir las aristas del grafo y calcular la distancia desde la salida a todos los nodos (equivalente porque hemos invertido el grafo).

Calcular la distancia de un nodo del grafo a todos es un problema bien conocido, el SSSP (Single Source Shortest Paths), que puede ser resuelto con el algoritmo de Dijkstra.

Nota: este es un truco que puede resultar un poco artificial la primera vez, pero es bastante conocido. Otro problema que utiliza este principio es el problema: Acepta el Reto 292 - Repartiendo paquetes

Enlace: https://www.aceptaelreto.com/problem/statement.php?id=292

Código

```
#include <iostream>
#include <vector>
#include <queue>      //Priority-queue
#include <functional> //Greater
using namespace std;

using vi = vector<int>;

using ii = pair<int, int>;
using vii = vector<ii>;

struct tArista {
    int fin,
        coste;
};

const int INF = 1000 * 1000 * 1000;

//Función que computa el algoritmo de Dijkstra sobre el grafo representado por
// listaAdy a partir del nodo salida
vi dijkstra(int salida, vector<vector<tArista>> const& listaAdy) {
    vi distancia(listaAdy.size(), INF);
    distancia[salida] = 0;

    //En la cola de prioridad introducimos pares (distancia, nodo)
    priority_queue<ii, vii, greater<ii>> pq;
    pq.push({0, salida});
```

I Olimpiada Informática de Madrid

Soluciones

Código

```
    while (!pq.empty()) {
        int nodoAct = pq.top().second;
        int distAct = pq.top().first;
        pq.pop();

        //Nodo residual, lo ignoramos
        if (distancia[nodoAct] < distAct) continue;

        //Si es realmente el nodo que toca procesar en el Dijkstra,
        // tratamos de añadir las aristas que parten de él
        for (tArista arista : listaAdy[nodoAct]) {
            int nodoSig = arista.fin;
            int coste = arista.coste;

            //Si la arista considerada reduce la distancia encontrada
            // hasta el momento para llegar a un nodo, reducimos dicha
            // distancia
            if (distancia[nodoSig] > distAct + coste) {
                distancia[nodoSig] = distAct + coste;
                pq.push({ distAct + coste, nodoSig });
            }
        }
    }

    return distancia;
}

bool res() {
    int n, s, t, p;
    cin >> n >> s >> t >> p;
    --s;
    if (!cin) return false;

    vector<vector<tArista>> listaAdy(n);
    for (int i = 0; i < p; ++i) {
        int ini, fin, coste;
        cin >> ini >> fin >> coste;
        --ini; --fin;

        //Introducimos las aristas INVERTIDAS (comienzan en fin y
        // terminan en ini)
        listaAdy[fin].push_back({ini, coste});
    }

    vi distancia = dijkstra(s, listaAdy);
```

```
1       int sol = 0;
2       //Para cada ratón (uno en cada nodo) vemos si le da tiempo a salir
3       for (int i = 0; i < distancia.size(); ++i) {
4           if (i != s && distancia[i] <= t)
5               ++sol;
6       }
7       cout << sol << '\n';
8
9       return true;
10  }
11
12  int main() {
13      while (res()) {}
14      return 0;
15  }
```

I Olimpiada Informática de Madrid

Soluciones

H - ¿Cuándo seré rico?

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=602

Explicación

Simplemente tenemos que simular el proceso hasta llegar a un cierto día en el que la cantidad de dinero obtenida sea superior a la cantidad de dinero deseada.

$$\begin{cases} \text{paga}(1) = 1 \\ \text{paga}(2) = 1 \\ \text{paga}(i) = \text{paga}(i-1) + 2\,\text{paga}(i-2) \end{cases}$$

Calcular esto de forma directa empleando la definición recursiva nos conduciría a un TLE, puesto que repetiríamos múltiples veces la llamada a $\text{paga}(i)$ para cada valor de i (se explica esto más en detalle en el siguiente problema). La mejor manera de calcular esto es de forma iterativa, de forma que si tenemos almacenado $\text{paga}(i-2)$ y $\text{paga}(i-1)$ podemos calcular fácilmente $\text{paga}(i)$.

Dada esta recurrencia, vemos que la paga obtenida en el día i es aproximadamente el doble que la obtenida en el día $i-1$. Por ello, la complejidad de la solución está en $O(\log n)$

Código

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

int main() {
    int nc;
    cin >> nc;
    for (int z = 0; z < nc; ++z) {
        int paga;
        cin >> paga;
        if (paga == 1) {
            cout << 1 << '\n';
            continue;
        }
        int diaAct = 2;
        int dineroTotal = 2, dineroAct = 1, dineroAnt = 1;
        while (dineroTotal < paga) {
            ++diaAct;
            int aux = dineroAct;
            dineroAct = dineroAct + 2 * dineroAnt;
            dineroAnt = aux;
            dineroTotal += dineroAct;
        }
        cout << diaAct << '\n';
    }
    return 0;
}
```

I Olimpiada Informática de Madrid

Soluciones

I - El carpintero Ebanisto

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=603

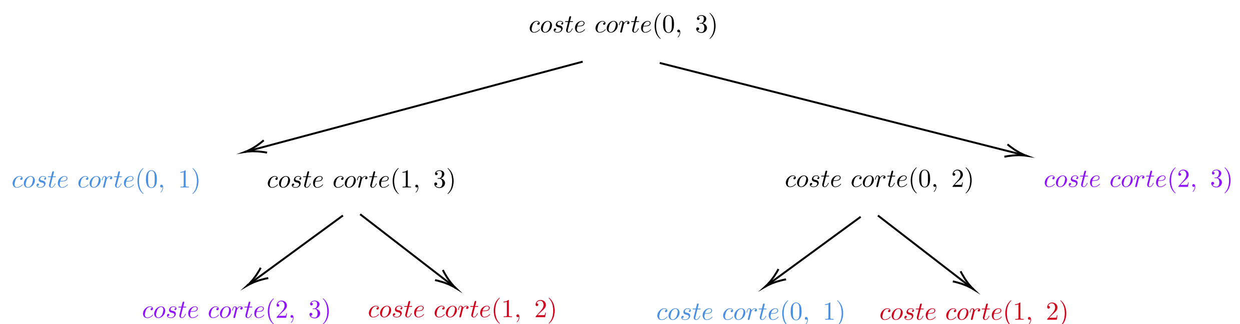
Explicación

Una primera observación que tenemos que hacer es que, dado un tablón, independientemente de por dónde cortemos, el coste del primer corte será el mismo. Por ello, cuando hacemos un corte, lo que queremos es minimizar la suma de los costes de hacer los cortes en los 2 listones que nos quedan.

En lo que nos vamos a fijar ahora es en los puntos en los que tenemos que hacer los cortes. Supongamos que el 0 es el punto inicial del tablón, el $n+1$ el punto final y los puntos del 1 al n los puntos en los que hay que efectuar un corte. Entonces, el coste buscado será:

$$\begin{cases} \text{coste_corte}(i,\, i+1) = 0 \\ \text{coste_corte}(i,\, j) = 2 * \text{longitud} + \min\limits_{i<k<j} (\text{coste_corte}(i,\, k) + \text{coste_corte}(k,\, j)) & i+1 \ < \ j \end{cases}$$

De esta forma, una llamada a coste_corte$(0,\, n+1)$ nos devolverá la solución. Sin embargo, una implementación directa de estas fórmulas no es eficiente. El problema reside en que haremos un gran número de llamadas a la función con los mismos argumentos. Por ejemplo:



Implementándolo directamente, recalcularemos en cada llamada el valor de la función, obteniendo TLE. Para evitar esto utilizamos **programación dinámica**: llevamos una tabla en la que para cada inicio y cada fin del tablón almacenamos el valor de *coste_corte*. De esta forma, en la primera llamada lo calcularemos y en las subsecuentes la tabla nos dará el valor.

Por lo tanto, el coste en memoria es $O(n^2)$, puesto que tenemos que almacenar el valor de *coste_corte* para cada posible inicio y fin, y en tiempo en $O(n^3)$, puesto que para cada posible inicio y fin tenemos que iterar por los distintos valores de k (cantidad lineal respecto al número de cortes que hay que realizar).

Código

```
#include <iostream>
#include <vector>
#include <climits> //INT_MAX
#include <algorithm>
using namespace std;

using vi = vector<int>;
using vvi = vector<vi>;

int dp(int iniTablon, int finTablon, vi const& cortes, vvi & marcaje) {
    if (iniTablon + 1 == finTablon) return 0;
    else if (marcaje[iniTablon][finTablon] != -1)
        return marcaje[iniTablon][finTablon];
    else {
        int minCoste = INT_MAX;

        for (int corte = iniTablon + 1; corte < finTablon; ++corte) {
            ll costeAct = dp(iniTablon, corte, cortes, marcaje) +
                          dp(corte, finTablon, cortes, marcaje);
            minCoste = min(minCoste, costeAct);
        }

        marcaje[iniTablon][finTablon] = minCoste + cortes[finTablon]
                                                 - cortes[iniTablon];
        return marcaje[iniTablon][finTablon];
    }
}

int main() {
    int l, n;
    cin >> l >> n;

    while (l != 0 || n != 0) {
        vi cortes(n + 2);
        cortes[0] = 0;
        cortes[n + 1] = l;
        for (int i = 1; i <= n; ++i)
            cin >> cortes[i];

        vvi marcaje(n + 2, vi(n + 2, -1));
        cout << 2 * dp(0, n + 1, cortes, marcaje) << '\n';

        cin >> l >> n;
    }
    return 0;
}
```