ALGORITMOS Y ESTRUCTURAS DE DATOS II GRADO INGENIERÍA INFORMÁTICA, CURSO 24/25 PRÁCTICA DE AVANCE RÁPIDO Y BACKTRACKING

Eduardo Terry Gavilá 24420735W

Aaron Atonga Ruiz 54961208V

Correos:

eduardo.terryg@um.es

a.atongaruiz@um.es

Usuario Mooshak: G3_75

Lista de envíos al juez online:

Avance rápido:

Hemos realizado el problema E. Este problema consiste en asignar averías a los mecánicos disponibles.

Envíos realizados:

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
4	177:47:05		G3 G3 75	E_AR	C++	3 Accepted	final
3	177:39:43		G3 G3 75	E_AR	C++	1 Wrong Answer	final
2	177:38:50		G3 G3 75	A_AR	C++	0 Wrong Answer	final
1	177:37:08		G3 G3 75	E_AR	C++	1 Wrong Answer	final

Se puede observar que hay 3 “Wrong Answer” y finalmente el “Accepted”.

El segundo “Wrong Answer” es porque hubo un error al seleccionar el problema y como se puede observar enviamos el problema E al juez del problema A.

Los otros dos son porque se envió un código el cual también funciona correctamente, pero en la función “Seleccionar” se escogía al mecánico con menos tareas disponibles ya que es más probable que luego no se le pueda asignar otra avería de manera que de mejores soluciones. Tenemos la sospecha de que es porque es más costoso, pero como el resultado es “Wrong Answer” y no Time “Limit Exceeded” pensamos que puede ser porque el juez no está programado para aceptar esa respuesta o que algo se nos escapa. Este código correspone al fichero “ARAnt.cpp” de la carpeta “AR”.

El “Accepted” corresponde al fichero “AR.cpp” también de la carpeta “AR” el cual tiene una función “Seleccionar” en el que simplemente se le asigna la avería al primer mecánico que pueda resolverla.

Backtracking:

Hemos realizado el problema F. Este problema trata de comprar una prenda de cada tipo y que la suma del precio total sea igual al presupuesto o lo más alta posible sin pasarse de este.

Envíos realizados:

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
81	462:00:59		G3 G3 75	F_Ba	C++	3 Accepted	final
68	450:20:45		G3 G3 75	F_Ba	C++	0 Time Limit Exceeded	final
67	450:14:17		G3 G3 75	F_Ba	C++	0 Time Limit Exceeded	final
66	450:05:33		G3 G3 75	F_Ba	C++	0 Time Limit Exceeded	final
65	449:38:51		G3 G3 75	F_Ba	C++	0 Time Limit Exceeded	final
60	440:47:11		G3 G3 75	F_Ba	C++	0 Time Limit Exceeded	final
50	434:28:29		G3 G3 75	F_Ba	C++	0 Time Limit Exceeded	final

Se puede observar que hay 6 “Time Limit Exceeded” y el “Accepted”.

Los “Time Limit Exceeded” se deben a complicaciones con las podas. Se ve en la imagen que muchas de ellas son el mismo día o incluso la misma hora ya que no dábamos con la solución. El problema era que no teniamos poda y en los cuatro ultimos el problema fue una mala condicion en el if de la funcion “Generar” ya que era:

if(nivel == 0 and S[nivel] - 1 > 0 and prendas[nivel][S[nivel]] <= prendas[nivel][S[nivel] - 1] and S[nivel] < int(prendas[nivel].size()) - 1)

La primer condición nivel == 0 estaba limitando la poda. Este código corresponde al fichero “BTSinPoda.cpp” de la carpeta “BT”.

El “Accepted” corresponde al fichero “BT.cpp” de la carpeta “BT” en el que se quita esa condición consiguiendo un rendimiento increiblemente alto en ejercicios en los que debido a que no se encuentra solucion habia que recorrer el árbol completamente lo cual podía suponer minutos de diferencia cuando el árbol tiene alrededor de 10 niveles.

Resolución de problemas:

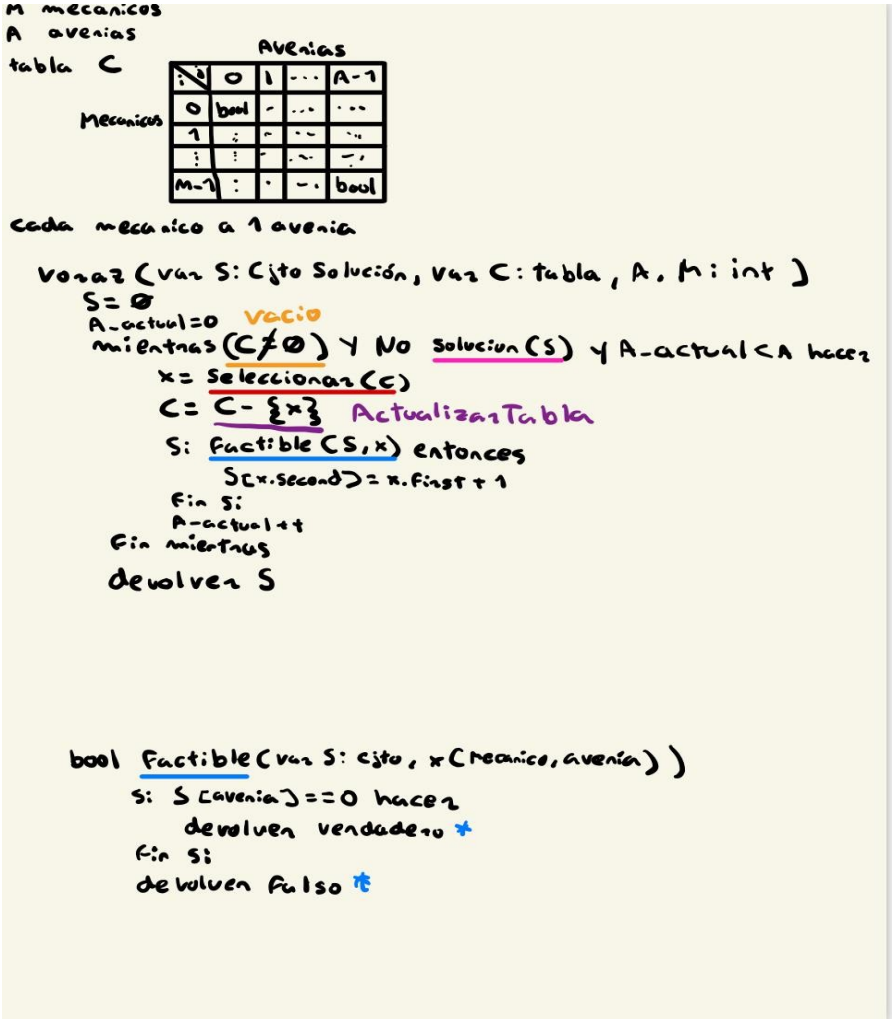
En este apartado se resolverán los apartados correspondientes a cada problema.

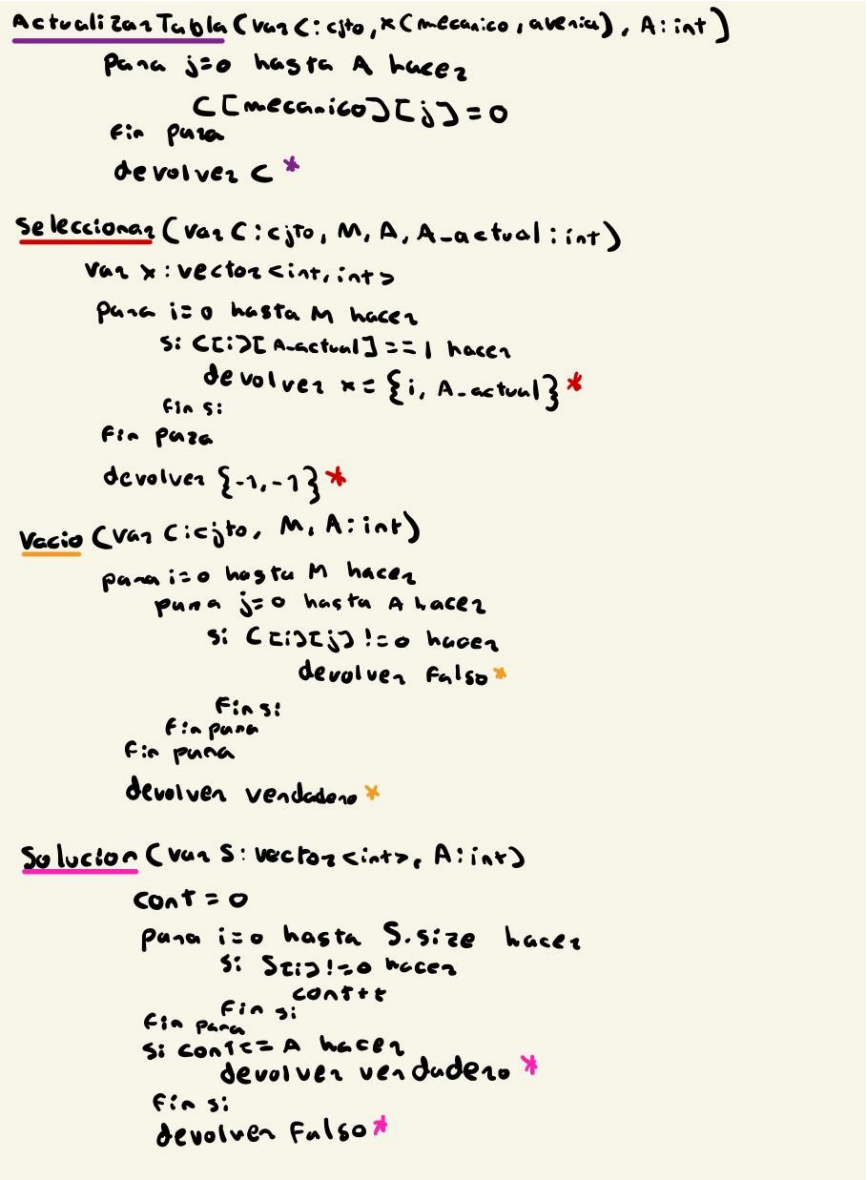
Avance rápido:

Introducción:

Como ya hemos dicho hemos realizado el problema E. En este problema tenemos M mecánicos y A averías que solucionar. También tenemos una tabla bidimensional C en la cual se nos muestra si un mecánico i puede reparar la avería j. Tenemos que reparar el número máximo de averías teniendo en cuenta que solo se puede asignar un mecánico a una avería al día.

Diseño en pseudocódigo:





Esta implementación se ha realizado siguiendo el esquema general de teoría, pero con leves modificaciones. En el programa tenemos un conjunto solución S en el que vamos a guardar para cada avería que mecánico la repara. Ejemplo: S(1, 4, 0). Esto quiere decir que la avería 1 la resuelve el mecánico 1, la avería 2 el mecánico 4 y no hay mecánico disponible para la avería 3. También se usa la tabla C y las variables A y M antes mencionadas. Al empezar la función “voraz” inicializamos S con conjunto vacío y creamos la variable A_actual y la ponemos a 0. La variable A_actual guarda el número de avería que estamos analizando en ese momento.

A continuación, nos encontramos con un bucle while en el cual primero se comprueba que C no sea un conjunto vacío. Para comprobarlo se usa la función “vacio” la cual recorre la tabla C y si esta tiene algún elemento distinto a 0 devuelve falso. Después se comprueba que no se tenga una solución. Para ello, se usa la función “solucion” la cual recorre el vector solución procurando que este relleno con un contador. Si el contador es igual al número de

averías significará que hay solución y se devolverá verdadero. Finalmente, se comprueba que A_actual sea menor que A para comprobar que no nos pasamos.

Una vez dentro del while guardamos en una variable x la tupla mecánico-avería seleccionadas por la función “seleccionar”. Esta función recorre para la columna A_actual el primer mecánico que pueda arreglarla. Si lo encuentra devuelve la tupla mecánico-avería.

Después se actualiza la tabla C usando la función “actualizarTabla”. Esta funciona poniendo la fila del mecánico seleccionado en x a 0 ya que como ya tiene una avería asignada ya no se le puede asignar a otra.

Tras esto hay que comprobar que la tupla seleccionada en x es factible. Para ello hemos implementado la función “factible”. Esta función simplemente comprueba si en la posición avería del vector solución S había ya un mecánico asignado. Si no lo había se devuelve verdadero. Si se obtiene el verdadero se guarda el mecánico en la posición avería de S.

Finalmente, justo antes de que finalice cada iteración del while actualizamos la avería actual poniendo en A_actual la siguiente avería. Al terminar el while se devuelve el conjunto solución S.

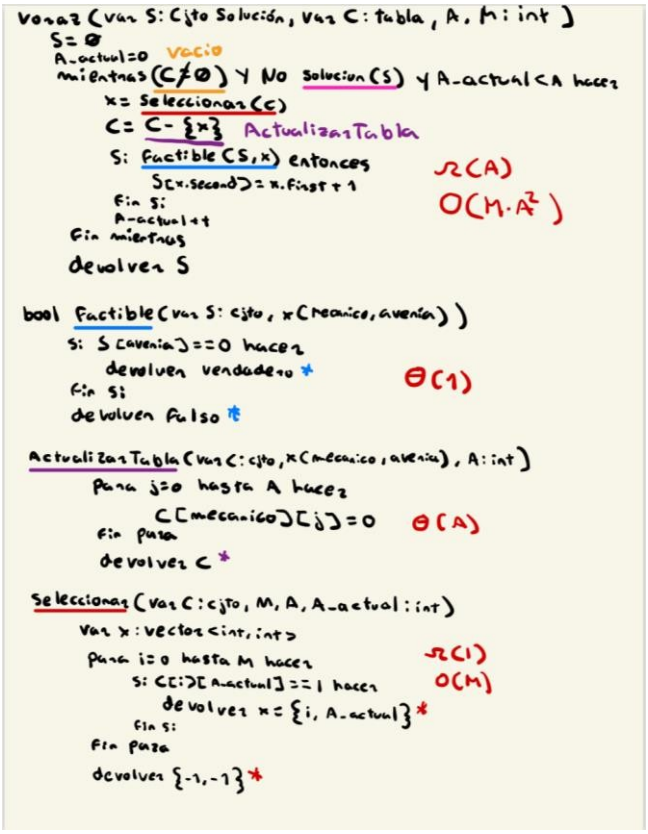
Implementación:

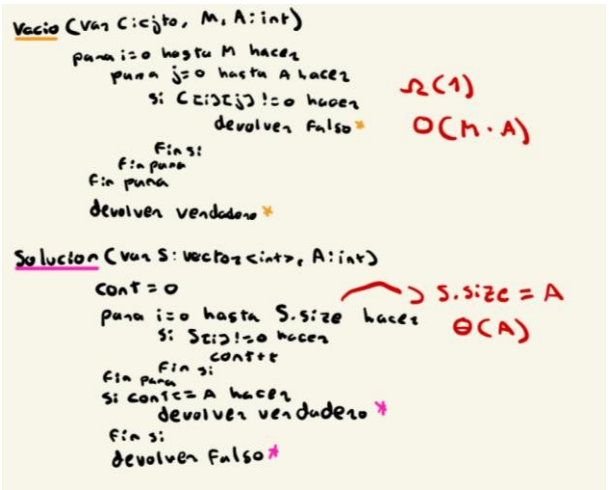
```cpp
#include <iostream>
#include <vector>

using namespace std;

bool Solucion(vector<int> S, int A){

    int cont = 0;

    for(int i = 0; i < int(S.size()); i++){ //Para cada valor del vector solucion se comprueba si la averia tiene mecanico asignado
        if(S[i] != 0){
            cont ++;
        }
    }

    if(cont == A){  //Si hay tantos mecanicos asignados como averias significan que todas estan cubiertas por lo que hay solucion
        return true;
    }
    return false;
}

bool Vacio(vector<vector<int>> C, int M, int A){

    for(int i = 0; i < M; i++){
        for(int j = 0; j < A; j++){
            if(C[i][j] != 0){
                return false;       //Se recorre la tabla C de manera que si esta a 0 significa que C esta vacio
            }
        }
    }

    return true;
}

pair<int, int> Seleccionar(vector<vector<int>> C, int M, int A, int A_actual){

    pair<int, int> x;   //Variable encargada de devolver la pareja mecanico-averia

    for(int i = 0; i < M; i++){
        if(C[i][A_actual] == 1){    //Recorremos la tabla comprobando que mecanicos pueden arreglar la averia actual
            return x = {i, A_actual};
        }
    }

    return {-1, -1};    //Se devuelve la decision voraz
}
```

```cpp
vector<vector<int>> ActualizarTabla(vector<vector<int>> C, pair<int, int> x, int A){

    for(int j = 0; j < A; j++){ //Ponemos las averias a 0 del mecanico asignado de manera que ya no se puede usar
        C[x.first][j] = 0;
    }

    return C;   //Devolvemos la tabla actualizada
}

bool Factible(vector<int> S, pair<int, int> x){

    if(S[x.second] == 0){   //Se comprueba si en la averia j-esima hay un mecanico asignado y si no lo hay se devuelve verdadero
        return true;
    }
    return false;
}

vector<int> Voraz(vector<vector<int>> C, int M, int A){

    vector<int> S(A, 0);    //Vector solucion de tamaño A (averias) el cual esta inicializado a 0

    pair<int, int> x = {0, 0};  //Pareja mecanico-averia incializados a 0

    int A_actual = 0;   //A_actual sirve para ver que averia es a la que le buscamos un mecanico disponible en cada momento

    while(Solucion(S, A) == false && Vacio(C,M,A) == false && A_actual < A){    //Se comprueba que no haya solucion aun en S, la tabla c no este vacia y que A_actual no sobrepasa a A
        x = Seleccionar(C,M,A,A_actual);   //Se guarda en x la pareja mecanico-averia seleccionada
        if(x.first != -1 && x.second != -1){   //Se comprueba que no se trate de un caso en el que no se ha encontrado ningun mecanico disponible
        C = ActualizarTabla(C,x,A);        //Se actualiza la tabla C con lo obtenido en x

        if(Factible(S,x) == true){     //Se comprueba que la pareja obtenida es adecuada para guardarla en la solucion
            S[x.second] = x.first + 1; //Se guarda en la solucion el mecanico obtenido anteriormente sumandole 1 ya que los arrays comienzan en 0
        }
        }
        A_actual++;    //Se le suma uno a A_actual para pasar a analizar la siguiente averia en la proxima iteracion
    }

    return S;
}
```

```cpp
int main(){

    int P;
    cin >> P;   //Se lee el numero de casos de prueba

    cout << endl << P << endl;

    for (int caso = 0; caso < P; caso++) {  //Se resuelve el problema una vez para cada caso

        int M, A;
        cin >> M >> A;  //Se leen los mecanicos y averias de este caso

        vector<vector<int>> C(M, vector<int>(A, 0));    //Declaracion de la tabla

        for (int i = 0; i < M; i++) {
            for (int j = 0; j < A; j++) {
                cin >> C[i][j]; //Se leen los valores de cada tabla de entrada
            }
        }

        vector<int> S = Voraz(C, M, A); //Se guarda en S la solucion correspondiente

        int reparadas = 0;
        for (int i = 0; i < A; i++) {   //Se cuentan las averias reparadas para imprimirlas
            if (S[i] != 0)
                reparadas++;
        }

        cout << reparadas << endl;

        for (int i = 0; i < A; i++) {
            cout << S[i] << (i < A - 1 ? " " : ""); //Se imprime la solucion con espacios entre caracteres excepto la ultima que no se imprime nada
        }

        cout << endl;
    }

    return 0;
}
```

Análisis teórico:





Se puede observar que el algoritmo tiene un O(M * A^2) y un Ω(A). Esto se debe a que si la tabla tiene solo una fila solo se recorrerá una vez el while y se recorrerá A en la función AcutalizarTabla. Para el peor caso hay que recorrer A veces M * A debido a que habría que recorrer toda la tabla en la función “vacio”.

Análisis práctico:

Hemos hecho unas pruebas empíricas las cuales consisten en probar primero para valores iguales y luego para A constante y M variable y viceversa. Se ha usado el programa "PruebaAr.cpp".

Aquí los valores:

M = 100 y A = 100 t = 0,0045s

M = 500 y A = 500 t = 0,2564s

M = 1000 y A = 1000 t = 2,2783s

M = 2000 y A = 2000 t = 25,9325s

M = 3000 y A = 3000 t = 1m y 31,251s

M = 4000 y A = 4000 t = 3m y 50,4384s

M = 5000 y A = 5000 t = 7m y 21,5432s

M = 1000 y A = 100 t = 0,038s

M = 1000 y A = 500 t = 0,5473s

M = 1000 y A = 1000 t = 2,2783s

M = 1000 y A = 2000 t = 9,2352s

M = 1000 y A = 3000 t = 16,8432s

M = 1000 y A = 5000 t = 21,4134s

M = 2000 y A = 100 t = 0,079s

M = 2000 y A = 500 t = 1,9531s

M = 2000 y A = 1000 t = 5,2242s

M = 2000 y A = 2000 t = 25,9325s

M = 2000 y A = 3000 t = 50,3453s

M = 2000 y A = 5000 t = 1m y 24,1395s

M = 100 y A = 1000 t = 0,065s

M = 500 y A = 1000 t = 0,9123s

M = 1000 y A = 1000 t = 2,2783s

M = 2000 y A = 1000 t = 5,6743s

M = 3000 y A = 1000 t = 8,6234s

M = 5000 y A = 1000 t = 14,6233s

M = 100 y A = 2000 t = 0,187s

M = 500 y A = 2000 t = 2,2562s

M = 1000 y A = 2000 t = 9,2352s

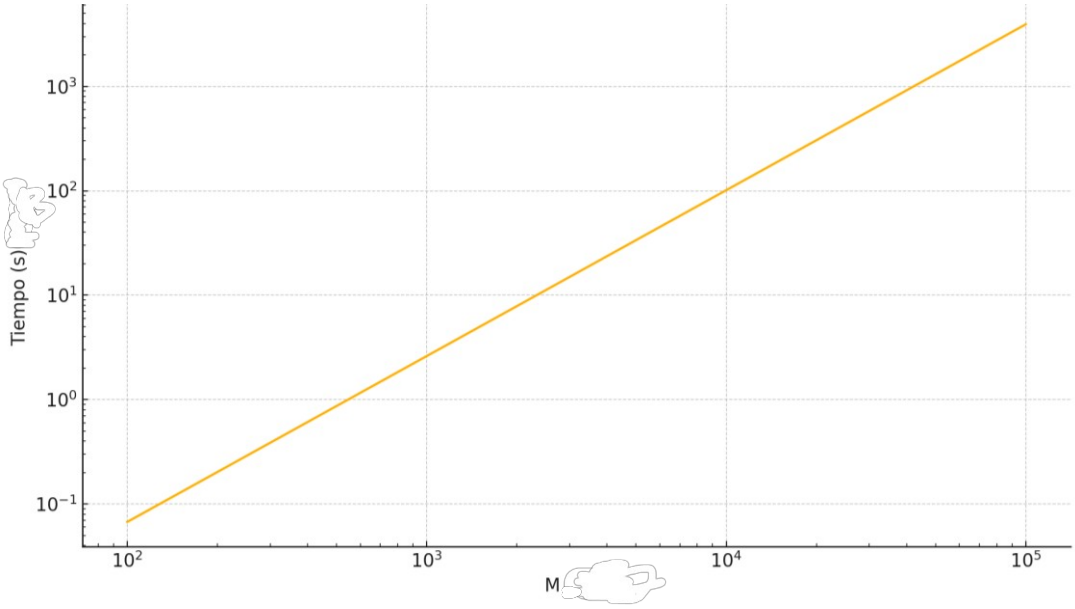
M = 2000 y A = 2000 t = 25,9325s

M = 3000 y A = 2000 t = 37,7435s

M = 5000 y A = 2000 t = 1m y 0,4493s

M = 5000 y A = 100 t = 0,2031s

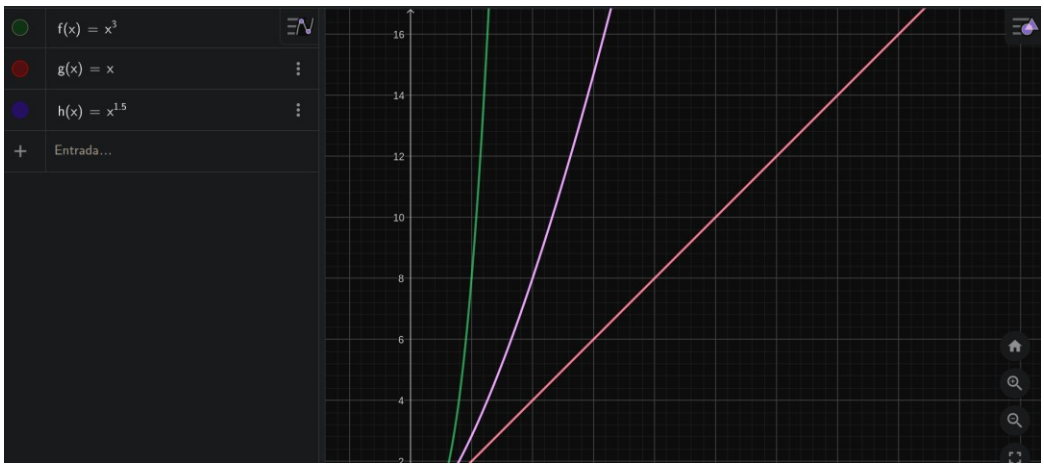
M = 100 y A = 5000 t = 0,4743s



Se puede aproximar n^1.5 más o menos.

Contraste:

Imagen de las gráficas:



Podemos observar que x^1.5 que es lo que nos salía en el análisis practico y lo cual representa un promedio aproximado del tiempo de ejecución se encuentra acotado superiormente por x^3 que es una aproximación de M * A^2 y x que es una aproximación de A lo cual nos da a entender que es un resultado satisfactorio.

Este es un resultado razonable en un voraz ya que suelen tener tiempo polinomial y x^1.5 se aproxima mucho a un tiempo polinomial.

Backtracking:

Introducción:

Hemos realizado el problema F. En este problema tenemos que comprar un modelo de cada prenda de manera que el precio total de estas no se pase de nuestro presupuesto, pero se pide que cuanto más alto sea mejor. Disponemos de un presupuesto M y una cantidad de prendas C con K modelos cada una. El valor de K puede variar para cada C.

Diseño en pseudocódigo:

```
Backtracking(var S: TuplaSolución)
 nivel = 0
 S = (-1, -1, -1 ...)
 Voa = -∞
 Pa = (0, 0, ...)
 Pact = 0
 repetir
   Si (MasHermanos(S, nivel, prendas))
     Genera(nivel, S, Pact, prendas, Pa)
     Si Solucion(nivel, C, M, Pact) AND Pact > Voa
        Voa = Pact
     Si Criterio(nivel, C, M, Pact) entonces
        nivel = nivel + 1
     Sino
        mientras NOT masHermanos(S, nivel, prendas) AND (nivel > 0) hacer
           Retroceden(Pact, S, nivel, prendas, Pa)
     Fin Si
   Sino
     Retroceden(Pact, S, nivel, prendas, Pa)
   Fin Si
 hasta nivel < 0 o Voa == M
 devolver Voa

 Genera(nivel, S, Pact, prendas, Pa)
       S[nivel] = S[nivel] + 1
       Si prendas[nivel][S[nivel]] <= prendas[nivel][S[nivel] - 1]
          S[nivel] = S[nivel] + 1
       Fin Si
       Pact = Pact + prendas[nivel][S[nivel]]
       Pa[nivel] = Pact

Solucion(nivel, C, M, Pact)
   devolver (nivel == C - 1) AND (Pact ≤ M)
Criterio(nivel, C, M, Pact)
   devolver (nivel <= C - 1) AND (Pact ≤ M)
```

```
MasHermanos(S, nivel, prendas)
     devolver S[nivel] < prendas[nivel].size() - 1

Retroceden(Pact, S, nivel, prendas, Pa)
     S[nivel] = -1
     nivel = nivel - 1
     Pact = Pa[nivel - 1]
```

Para esta implementación también se ha seguido el esquema general de teoría. Como se ha mencionado hay una variable M y una tabla C. Al empezar el algoritmo se inicializa una variable nivel a 0. Esta variable informa en qué nivel del árbol se encuentra el programa en cada momento. Despues se inicializa un vector solución S a –1. Este no se va a devolver en esta ocasión, pero va almacenando la solución parcial. Empieza a –1 ya que al generar hermanos se le suma uno a la posición adecuada para saber en qué hermano estamos. Además, se inicializa una variable voa a -∞Esta variable va a guardar la mejor solución que se haya encontrado y será lo que finalmente devuelva el algoritmo. A continuación, hay un vector Pn el cual se inicializa a 0 y va a guardar resultados anteriores para que se puedan usar más tarde. Por último, se inicializa una variable pact a 0 la cual va a guardar el resultado que se esté obteniendo en la rama actual y se comparara con voa para ver si es el mejor resultado.

Tras todas las anteriores inicializaciones, el algoritmo comienza con un bucle “repetir” en el cual comenzaremos generando un nodo. Siempre se comprueba si hay más hermanos antes de generar otro. La función MasHermanos comprueba si el nodo actual tiene más hermanos con una simple comparación la cual consiste en confirmar si S[nivel] que es el hermano del nivel actual es menor que la anchura de ese nivel – 1. Esto significa por ejemplo que si estamos en el hermano 2 (que es el tercero) y la anchura del nivel es 3 pues no podríamos generar más porque ya estamos en el último hermano, no se cumple 2 < 3 – 1.

Tras comprobar lo anterior, como ya hemos dicho se genera un nodo. Esto se realiza con la función Generar. En esta función primero se le suma uno al hermano actual de manera que se crea otro. A continuación, nos encontramos con una poda, esta consiste en comprobar si el valor del nodo actual es menor igual que el del anterior. Teniendo en cuenta que, si el hermano anterior vale 30 por ejemplo, y se ha combinado con todas las posibilidades, y el actual vale 20 si o si va a ser menor el resultado por lo que no interesa ni mirarlo y se coge el siguiente. Después, se actualiza pact sumándole el precio del modelo escogido en el nodo actual y se actualiza también Pn guardando pact en él.

Después de general un nodo y actualizar pact se comprueba si hay una nueva solución. La función Solución simplemente comprueba si se está en el último nivel y pact es menor o igual que M de manera que se devolvería verdadero al ser una solución válida.

Tras comprobar si hay una solución, si es verdadero, se comprueba si pact es mayor que voa verificando asi si la solución actual es mejor que la mejor que se tenía antes. Si es mejor, se actualiza voa con el valor de pact.

Justo después de comprobar lo anterior, el programa se asegura de que se puede subir de nivel usando la función Criterio para así continuar obteniendo una solución. Se asegura comprobando que no estamos ya en el nivel más alto y que pact no supera al presupuesto. Si devuelve verdadero aumentamos el nivel.

Si Criterio devuelve falso entonces significa que o estamos en el nivel más alto o nos pasamos de presupuesto. Antes de nada, se guarda en Pn el pact por si se usa la función Retroceder que se explicara más tarde. Ahora, se comprueba si podemos retroceder, para ello, se verifica que no haya más hermanos y que no estemos en el nivel más bajo. Mientras se cumpla lo anterior se retrocederá. Esto se consigue con la función Retroceder la cual en primer lugar pone a –1 el hermano actual de manera que se invalida para la siguiente búsqueda. Después, se baja el nivel restándole uno a nivel y actualizamos pact con el valor de Pn adecuado.

En este punto, termina el if inicial en el que se comprobaba que hay más hermanos. Si no hay más hermanos en un principio, directamente se retrocede de la misma manera que hemos hecho antes. Alguien se puede cuestionar por qué antes de retroceder se comprobaban los hermanos como ahora y además que no nos pasásemos de nivel y ahora no. Esto se explica a continuación.

Aquí termina el “repetir” inicial. Este dejara de repetirse cuando el nivel sea menor que cero o cuando voa sea igual a M. Esa es la explicación de porque no se comprobaba el nivel, ya que si estábamos ya en el menor nivel (el 0) y bajábamos a un nivel no existente significaría que ya hemos explorado todo el árbol y por ende el algoritmo terminaría. Por otra parte, como se ha mencionado comprueba que si voa es igual a M ya que si fuese afirmativo no tendría sentido seguir explorando ya que no existe mejor solución posible.

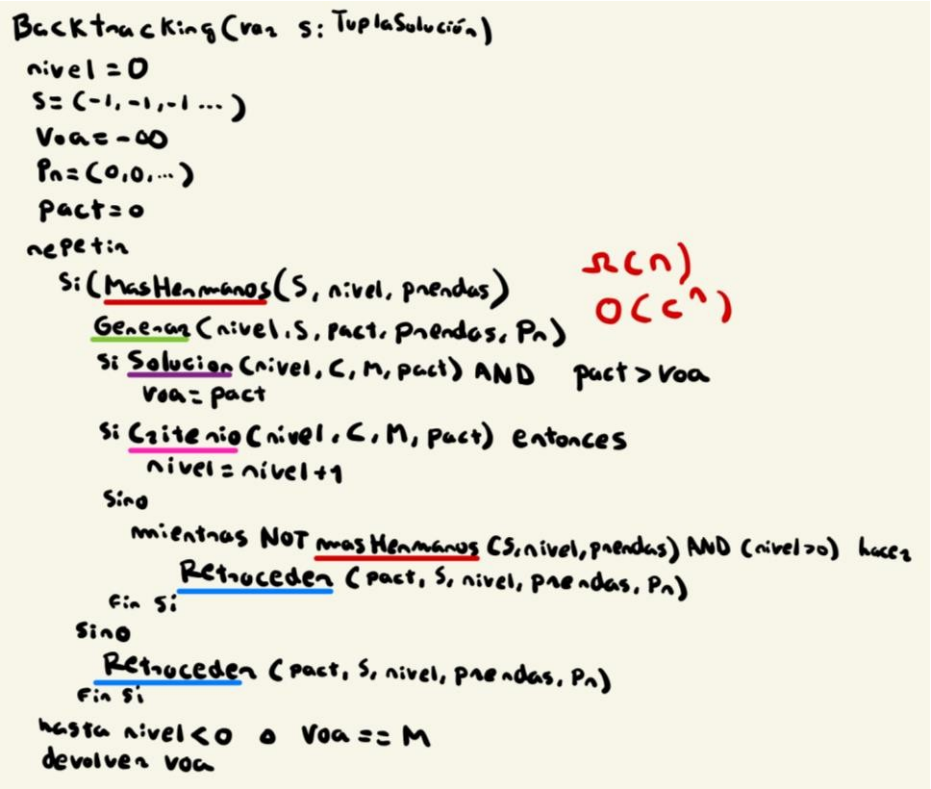
Implementación:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

void Generar(int nivel, vector<int> &S, int &pact, const vector<vector<int>> &prendas, vector<int> &Pn){    //Funcion que genera nodos

    S[nivel] = S[nivel] + 1;    //Generacion de un nuevo nodo
    if(S[nivel] - 1 > 0 and prendas[nivel][S[nivel]] <= prendas[nivel][S[nivel] - 1] and S[nivel] < int(prendas[nivel].size()) - 1){
        S[nivel] = S[nivel] + 1;    //Si no es rentable generar el nodo ya que va a ser menor al anterior pasamos al siguiente (poda)
    }
    pact = pact + prendas[nivel][S[nivel]]; //El precio acumulado actual sera el anterior mas lo que haya en el nodo actual
    Pn[nivel] = pact;   //Actualizacion de Pn
}

bool Solucion(int nivel, int C, int M, int pact){   //Funcion que comprueba si se tiene una solucion

    return (nivel == C - 1) and (pact <= M);    //Si el nivel es el mas alto y el precio acumulado actual es menor al presupuesto si es solucion
}

bool Criterio(int nivel, int C, int M, int pact){   //Funcion que comprueba si se puede subir de nivel

    return (pact <= M) and (nivel < C - 1);     //Si el precio acumulado actual es menor o igual al presupuesto y el nivel actual no es el mas alto podemos subir
}

bool MasHermanos(vector<int> S, int nivel, vector<vector<int>> prendas){    //Esta funcion comprueba si en el nivel actual hay mas hermanos

    return S[nivel] < (int)prendas[nivel].size() - 1;   // Si no estamos en el ultimo hermano es porque aun no lo hemos generado
}

void Retroceder(int &pact, vector<int> &S, int &nivel, vector<vector<int>> prendas, vector<int> Pn){    //Funcion que hace retroceder al nivel inferior

    S[nivel] = - 1;     //Anulamos los resultados actuales
    nivel = nivel - 1;  //Bajamos de nivel
    pact = Pn[nivel - 1];   //Configuramos el precio acumulado actual al que se tenia antes de subir
}
```

```cpp
int BackTracking(const vector<vector<int>> &prendas, int M, int C) {

    int nivel = 0;  //Variable que representa el nivel actual en el arbol
    int voa = INT_MIN;  //Valor optimo actual
    int pact = 0;   //Variable que contiene el precio acumulado en cierto momento
    vector<int> S(C, -1);   //Vector solucion S que contiene la solucion actual
    vector<int> Pn(C, 0);   //Vector el cual contendra el valor de niveles anteriores

    do{

        if(MasHermanos(S, nivel, prendas)){ //Antes de generar un nuevo nodo se comprueba que hayan mas hermanos
            Generar(nivel, S, pact, prendas, Pn);   //Se genera un nuevo nodo


            if(Solucion(nivel, C, M, pact) and (pact > voa)){   //Si tenemos una solucion y el precio actual es mayor que el mayor de los evaluados anteriormente se actualiza voa

                voa = pact;
            }

            if(Criterio(nivel, C, M, pact)){    //Si se cumple el criterio el cual comprueba que podemos subir de nivel subimos de nivel

                nivel = nivel + 1;
            }
            else{
                pact = Pn[nivel - 1];   //Si no se cumple el criterio guardamos en Pn el pact por si se usa mas tarde
                while(!MasHermanos(S, nivel, prendas) and (nivel > 0)){     //Comprobamos si no hay mas hermanos en el nivel actual y mas niveles abajo para poder bajar

                    Retroceder(pact, S, nivel, prendas, Pn);
                }
            }
        }
        else{   //Si no hay hermanos se retrocede
            Retroceder(pact, S, nivel, prendas, Pn);

        }
    } while(nivel >= 0 and voa != M);   //Se hace todo lo anterior hasta que nivel sea menos que 0 o voa valga M ya que no hay mejor solucion posible

    return voa; //Se devuelve el mejor valor obtenido
}
```

```cpp
int main(){
    int P;  //Numero de casos de prueba.
    cin >> P;

    for (int i = 0; i < P; i++){
        int M, C;
        cin >> M >> C;  //Se ingresa el presupuesto y el numero de tipos de prenda.

        vector<vector<int>> prendas(C); //Declaracion de la tabla C la cual contiene el precio de cada modelo de cada prenda
        for (int i = 0; i < C; i++){    //Una iteracion por cada tipo de prenda para rellenar la tabla
            int k;  //Numero de modelos para cada prenda
            cin >> k;
            prendas[i].resize(k);   //Dimensionado dinamico dependiendo del tamaño de cada fila
            for (int j = 0; j < k; j++){
                cin >> prendas[i][j];   //Precio de cada modelo de cada prenda
            }
        }

        for (auto& fila : prendas) {
            sort(fila.begin(), fila.end(), greater<int>()); //Ordenacion de mayor a menor de cada fila de manera que es mas probable encontrar
                                                            //la solucion antes suponiendo que los presupuestos suelen ser relativamente altos
        }



        int best = BackTracking(prendas, M, C); //Llamada al backtracking
        if(best < 0)    //Se comprueba si se ha obtenido o no la solucion
            cout << "no solution";
        else
            cout << best;
        cout << "\n";
    }

    return 0;
}
```

Análisis teórico:



En este algoritmo todas las funciones son de orden constante osea que no son relevantes. En el mejor de los casos, se inspeccionará solo una rama ya que se conseguirá que M sea igual a voa de manera que se termine el algoritmo. Consideramos que el número de niveles es n. En el peor de los casos, tendrá que visitar todos los niveles excepto los que sean podados. Siendo C el número máximo de opciones por nivel y n el número de niveles será un orden aproximado de C^n lo cual es exponencial pero esperado en este tipo de algoritmo.

Análisis práctico:

Para este análisis empírico hemos hecho igual que el anterior, pero para M y C con los valores establecidos en el enunciado, 1 <= *M* <= 200, y 1 <= *C* <= 20.

Resultados:

M = 50 y C = 1 t = 0.002s

M = 50 y C = 5 t = 0.002s

M = 50 y C = 10 t = 0.002s

M = 50 y C = 15 t = 0.002s

M = 50 y C = 20 t = 0.003s

M = 100 y C = 1 t = 0.002s

M = 100 y C = 5 t = 0.002s

M = 100 y C = 10 t = 0.005s

M = 100 y C = 15 t = 0.153s

M = 100 y C = 20 t = 0.521s

M = 200 y C = 1 t = 0.002s

M = 200 y C = 5 t = 0.002s

M = 200 y C = 10 t = 0.004s

M = 200 y C = 15 t = 0.177s

M = 200 y C = 20 t = 0.845s

M = 1 y C = 5 t = 0.002s

M = 10 y C = 5 t = 0.002s

M = 50 y C = 5 t = 0.002s

M = 100 y C = 5 t = 0.002s

M = 150 y C = 5 t = 0.002s

M = 200 y C = 5 t = 0.002s

M = 1 y C = 10 t = 0.002s

M = 10 y C = 10 t = 0.002s

M = 50 y C = 10 t = 0.004s

M = 100 y C = 10 t = 0.005s

M = 150 y C = 10 t = 0.005s

M = 200 y C = 10 t = 0.005s

M = 1 y C = 20 t = 0.002s

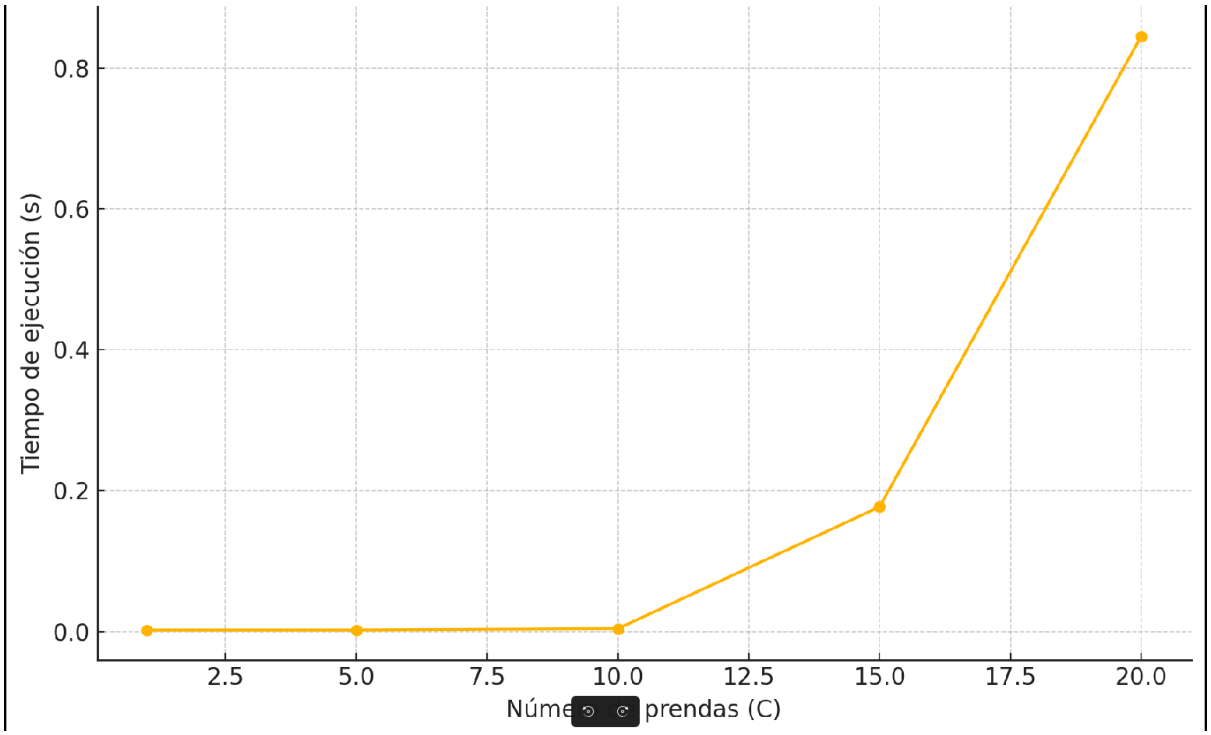
M = 10 y C = 20 t = 0.002s

M = 50 y C = 20 t = 0.003s

M = 100 y C = 20 t = 0.200s

M = 150 y C = 20 t = 3.7432s

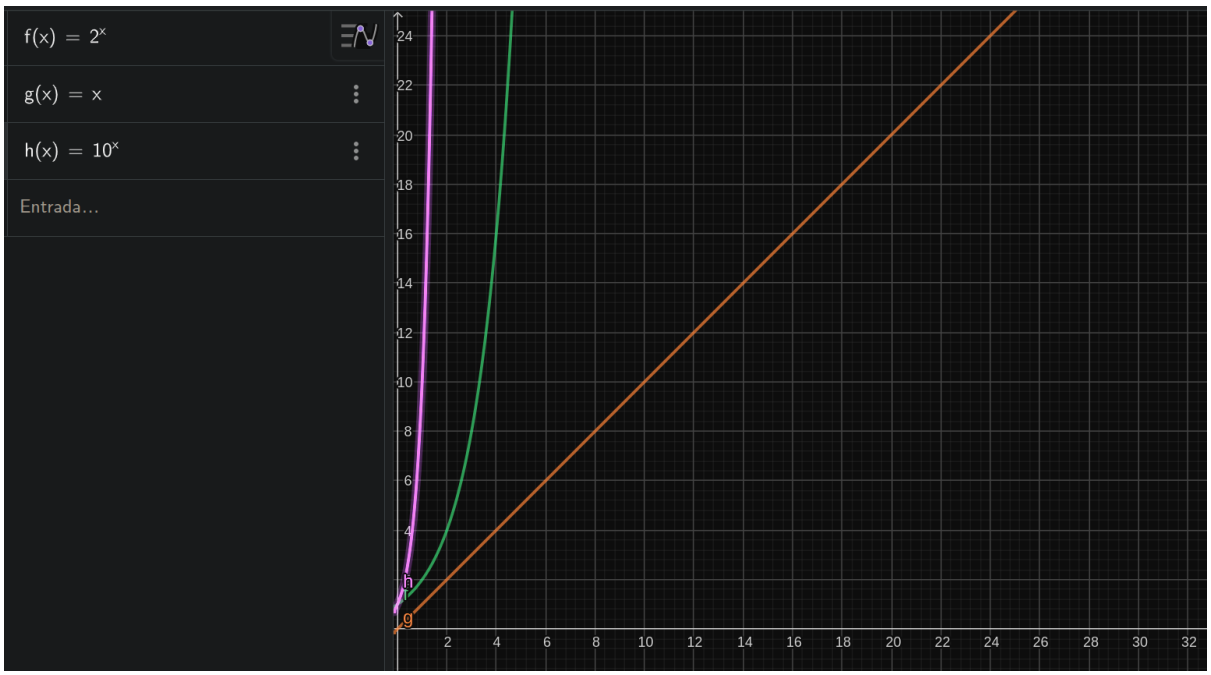
M = 200 y C = 20 t = 4.4324s



En este análisis también nos hemos percatado de que el tiempo de ejecución no depende de M. Esta función se puede aproximar a 2^n debido a su inicio lento y después exponencial.

Contraste:

Imagen de las gráficas:



Se ha utilizado x como aproximación del número de niveles (n) y 10^x como aproximación del número máximo de opciones por nivel elevado al número de niveles. Se puede ver como la función 2^x esta acotada por estas dos por lo que es un resultado y satisfactorio viniendo de este tipo de algoritmo.

Conclusión:

El algoritmo rápido ha sido sencillo de implementar y tarda lo que se esperaba. Lo único que queremos resaltar es el algoritmo posiblemente mejorado que no ha aceptado el juez (ARAnt.cpp).

El backtracking a pesar del orden exponencial nos ha sorprendido muchísimo como ha mejorado con la poda. Hemos creado un documento expresando la diferencia de tiempo con y sin poda. El fichero es “tiempoPoda.txt” y el contenido es el siguiente:

-El programa tarda 3 minutos y 57,820 segundos antes de la poda para la entrada extendida ofrecida por mooshak.

-El programa tras la poda tarda 6,379 segundos para la entrada extendida ofrecida por mooshak.

Es increible la diferencia y estamos convencidos de que esto puede suponer la diferencia entre ser un buen programador o no serlo.

Queremos aprovechar para comentar nuestra satisfacción con el juez online ya que es muy buena herramienta y hace que todo sea más cómodo.

Hemos tardado aproximadamente 25 horas cada uno para la práctica completa.