Algoritmos y Estructuras de Datos II

Práctica AR y BT

Curso 2023-2024

Grupo de prácticas: [2.*3*]
Profesor: Jesús Sánchez Cuadrado
Cuenta Mooshak: G2_53

Autores:
Ortiz García, Juan Jesús | 24420563Z
Vicente Pérez, Gonzalo | 24420789X

Índice

1. Lista de problemas resueltos

El ejercicio de Avance Rápido que nos ha tocado es el C, mientras que el de Backtracking es el F.

Avance rápido

Hicimos un total de dos envíos, prácticamente de forma seguida. Nuestro enfoque a la hora de hacer los ejercicios fue no subirlos hasta que obtuviéramos la misma salida que la indicada al pasarle la entrada que aparece en Mooshak. Por tanto, cuando lo subimos ya nos salía la misma salida, lo que implicaba que el algoritmo ya estaba bien hecho e implementado, pero vimos que se producía un error de compilación. Para solucionarlo, buscamos en Internet el aviso que salía en Mooshak y encontramos que seguramente era debido al uso del operador auto para iterar, por lo que alteramos el código para eliminarlos usando el contenedor size_t.

Número de envío: 126

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
126	532:59:36		G2 G2 53	C_AR	C++	2 Accepted	final
125	532:48:47		G2 G2 53	C_AR	C++	0 Compile Time Error	final

Backtracking

De nuevo hicimos dos envíos, asegurándonos otra vez que obteníamos la misma salida que la indicada en el archivo 901a.out para la entrada propuesta. En el primer de los envíos vimos que se excedía el tiempo límite, por lo que el cambio principal que decidimos introducir fue pasar de usar vectores a punteros y arrays dinámicos para ver si quizá aprovechando mejor la memoria podíamos solucionarlo. Esto nos llevó algo más de tiempo para solucionarlo, y hubo que adaptar otras partes del código en consecuencia, aunque el algoritmo en sí se mantuvo prácticamente igual.

Número de envío: 105

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
105	483:04:22		G2 G2 53	F_Ba	C++	3 Accepted	final
98	463:39:36		G2 G2 53	F_Ba	C++	0 Time Limit Exceeded	final

2. Avance Rápido

2.1 Pseudocódigo del algoritmo

Comenzamos definiendo una función de selección inicial, de modo que el primer candidato no se estudie a partir del primer candidato posible, sino del mejor candidato posible, es decir, aquel que maximiza las distancias a otros nodos. Esta función la introdujimos al darnos cuenta de que la función genérica de selección no siempre elegía al candidato con distancias maximizadas. Su código es el siguiente:

```
SeleccionarCandidatoInicial(distancias: matriz de enteros, n: entero)
: entero
  sumaMax := -1
  candidato := 0
  para i desde 0 hasta n hacer
    sumaActual := 0
    para j desde 0 hasta n hacer
      sumaActual := sumaActual + distancias[i][j]
    finpara
    si sumaActual > sumaMax entonces
      sumaMax := sumaActual
      candidato := i
    finsi
  finpara
  devolver candidato
```

A partir de este candidato inicial, ya podemos definir una función de selección genérica que, recibiendo una lista de seleccionados y una matriz de distancias, vaya comprobando para cada candidato no seleccionado el incremento en la diversidad que supondría seleccionarlo. En caso de que este incremento sea superior al máximo registrado hasta el momento, se actualiza la variable que guarda el incremento máximo y se actualiza el candidato. Una vez se llega al final del bucle, se devuelve el índice del candidato con el mayor incremento de diversidad:

```
Seleccionar(seleccionados: lista de enteros, distancias: matriz de
enteros, estaSeleccionado: lista de booleanos) : entero
  incrementoMax := -1
  siguienteCandidato := -1
  para i desde 0 hasta tamaño de distancias hacer
    si no estaSeleccionado[i] entonces
```

```
              incrementoActual   :=   calcularIncremento(seleccionados,
distancias, i)
      si incrementoActual > incrementoMax entonces
        incrementoMax := incrementoActual
        siguienteCandidato := i
      finsi
    finsi
  finpara
  devolver siguienteCandidato
```

Ahora definiremos el resto de funciones complementarias que nos van a completar el algoritmo voraz. Estas son Factible, Insertar y Solución, que siguen cada una la lógica clásica del avance rápido. Factible nos devuelve si es posible introducir un candidato, Insertar lo añade a la lista poniendo dicho candidato a "true" y Solución nos dice si el conjunto de candidatos es una solución válida.

```
Factible(seleccionados: lista de enteros, m: entero) : booleano
  devolver tamaño de seleccionados < m
```

```
Insertar(seleccionados: lista de enteros, estaSeleccionado: lista de
booleanos, candidato: entero)

  agregar candidato al final de seleccionados
  estaSeleccionado[candidato] := verdadero
```

```
Solucion(seleccionados: lista de enteros, m: entero) : booleano
  devolver tamaño de seleccionados == m
```

Definimos también una función que nos calcule el incremento de diversidad que supone la adición de un candidato:

```
CalcularIncremento(seleccionados: lista de enteros, distancias: matriz
de enteros, candidato: entero) : entero
  incremento := 0
  para elemento desde 0 hasta tamaño de seleccionados hacer
      incremento  :=  incremento  +  distancias[seleccionados[elemento]]
[candidato] + distancias[candidato][seleccionados[elemento]]
  finpara
  devolver incremento
```

Haciendo uso de las funciones que hemos definido, vamos a construir una función llamada Voraz que contenga la lógica del avance rápido con las particularidades del problema.

En primer lugar, inicializamos una lista de seleccionados, donde iremos guardando los candidatos que tomemos. Inicializaremos también a "false" un vector de candidatos que indique si cada uno de los candidatos ha sido seleccionado. A continuación, llamamos a la función de selección inicial y nos quedamos con el mejor primer candidato.

A partir de aquí, comenzamos a aplicar el algoritmo voraz, de modo que mientras no encontremos una solución, vayamos seleccionando candidatos e insertándolos, comprobando previamente si su inserción es factible.

Una vez acabado el bucle, calculamos la diversidad máxima y formamos el vector solución, devolviendo estos dos elementos.

El psuedocódigo queda de la siguiente manera:

```
diversidadVoraz(n: entero, m: entero, distancias: matriz de enteros) :
par de entero y lista de enteros
  seleccionados := lista vacía de enteros
  estaSeleccionado := lista de n elementos falsos
  inicial := seleccionarCandidatoInicial(distancias, n)
  Insertar(seleccionados, estaSeleccionado, inicial)
  mientras no Solucion(seleccionados, m) hacer
            candidato   :=   Seleccionar(seleccionados,   distancias,
estaSeleccionado)
    si Factible(seleccionados, m) entonces
      Insertar(seleccionados, estaSeleccionado, candidato)
    finsi
  finmientras
  diversidadMaxima := 0
  para i desde 0 hasta tamaño de seleccionados hacer
    para j desde i + 1 hasta tamaño de seleccionados hacer
                        diversidadMaxima    :=    diversidadMaxima    +
distancias[seleccionados[i]][seleccionados[j]]                          +
distancias[seleccionados[j]][seleccionados[i]]
    finpara
  finpara
  vectorSolucion := lista de n elementos 0
  para i en seleccionados hacer
    vectorSolucion[i] := 1
  finpara
  devolver (diversidadMaxima, vectorSolucion)
```

2.2 Programación del algoritmo

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

using namespace std;

// Calcular el incremento en diversidad si se agrega un nuevo candidato
int calcularIncremento(const vector<int>& seleccionados, const vector<vector<int>>& distancias, int candidato) {
    int incremento = 0;
      for (size_t elemento = 0; elemento < seleccionados.size(); ++elemento) {
        incremento += distancias[seleccionados[elemento]][candidato] + distancias[candidato][seleccionados[elemento]];
    }
    return incremento;
}

// Función para seleccionar el elemento inicial basado en la suma máxima de distancias a otros nodos
int seleccionarCandidatoInicial(const vector<vector<int>>& distancias, int n) {
    int sumaMax = -1, candidato = 0;
    for (int i = 0; i < n; ++i) {
        int sumaActual = 0;
        for (int j = 0; j < n; ++j) {
            sumaActual += distancias[i][j];
        }
        if (sumaActual > sumaMax) {
            sumaMax = sumaActual;
            candidato = i;
        }
    }
    return candidato;
}

// Seleccionar el mejor candidato posible a continuación del primero y siguientes
int Seleccionar(const vector<int>& seleccionados, const vector<vector<int>>& distancias, const vector<bool>& estaSeleccionado)
{
    int incrementoMax = -1;
    int siguienteCandidato = -1;
    for (size_t i = 0; i < distancias.size(); ++i) {
        if (!estaSeleccionado[i]) {
              int incrementoActual = calcularIncremento(seleccionados, distancias, i);
```

```
            if (incrementoActual > incrementoMax) {
                incrementoMax = incrementoActual;
                siguienteCandidato = i;
            }
        }
    }
    return siguienteCandidato;
}

// Verificar si es factible agregar un nuevo candidato
bool Factible(const vector<int>& seleccionados, int m) {
    return seleccionados.size() < static_cast<size_t>(m);
}

// Insertar un nuevo candidato en el conjunto solución
void      Insertar(vector<int>&      seleccionados,      vector<bool>&
estaSeleccionado, int candidato) {
    seleccionados.push_back(candidato);
    estaSeleccionado[candidato] = true;
}

// Función para verificar si se ha formado una solución completa.
bool Solucion(const vector<int>& seleccionados, int m) {
    return seleccionados.size() == static_cast<size_t>(m);
}

pair<int,   vector<int>>   diversidadVoraz(int   n,   int   m,   const
vector<vector<int>>& distancias) {
    vector<int> seleccionados;
    vector<bool> estaSeleccionado(n, false);

    // Seleccionar el candidato inicial
    int inicial = seleccionarCandidatoInicial(distancias, n);
    Insertar(seleccionados, estaSeleccionado, inicial);

    // Algoritmo voraz para seleccionar los siguientes candidatos
    while (!Solucion(seleccionados, m)) {
              int  candidato  =  Seleccionar(seleccionados,  distancias,
estaSeleccionado);
        if (Factible(seleccionados, m)) {
            Insertar(seleccionados, estaSeleccionado, candidato);
        }
    }

    // Calcular la diversidad final
    int diversidadMaxima = 0;
    for (size_t i = 0; i < seleccionados.size(); ++i) {
        for (size_t j = i + 1; j < seleccionados.size(); ++j) {
                    diversidadMaxima  +=  distancias[seleccionados[i]]
[seleccionados[j]] + distancias[seleccionados[j]][seleccionados[i]];
```

```
        }
    }

    vector<int> vectorSolucion(n, 0);
    for (int i : seleccionados) {
        vectorSolucion[i] = 1;
    }

    return {diversidadMaxima, vectorSolucion};
}

int main() {
    int T, n, m;
    cin >> T;
    while (T--) {
        cin >> n >> m;
        vector<vector<int>> distancias(n, vector<int>(n));
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                cin >> distancias[i][j];
            }
        }

            pair<int, vector<int>> resultado = diversidadVoraz(n, m,
distancias);
        cout << resultado.first << endl;
        for (int i : resultado.second) {
            cout << i << " ";
        }
        cout << endl;
    }

    return 0;
}
```

2.3 Estudio teórico del tiempo de ejecución

Función SeleccionarCandidatoInicial

En esta función recorremos todos los candidatos, seleccionando el mejor candidato inicial. Por tanto, recorreremos la matriz de distancias, que es de tamaño $n \, x \, n$, y además calculamos la suma a otros elementos para cada uno, obtenemos:

$$T(n) = 2 + 2n + 2n^2 \in O(n^2)$$

Función CalcularIncremento

Recorre todos los elementos ya seleccionados y calcula el incremento en la diversidad en caso de agregarse el candidato. Como cada para elemento se hace una operación constante (sumar distancias), el tiempo de ejecución vendrá directamente dado por el número de conjuntos seleccionados, m.

$$T(n) = 2m + 2 \in O(m)$$

Función Seleccionar

Iteramos sobre todos los posibles candidatos, comprobando que no hayan sido seleccionados y calculando su incremento en diversidad. Además, llamamos a calcularIncremento para cada candidato aún no seleccionado, por lo que el tiempo de ejecución total será n multiplicado por el tiempo de ejecución de calcularIncremento:

$$T(n) = 2 + n * O(m) = 2 + nm \in O(nm)$$

Funciones Insertar, Factible y Solución

Es trivial observar que son funciones de orden constante, puesto que simplemente se hace un añadido (en el caso de insertar) o se comprueba una condición (factible y solución). Por tanto, tendremos $O(1)$.

Función DiversidadVoraz

El bucle principal se ejecuta mientras se buscan candidatos hasta o bien alcanzar m elementos o bien hasta no encontrar más que incrementen la diversidad. En el peor caso, llamaremos a Seleccionar m veces, y cada llamada a Seleccionar es de $O(nm)$. Por tanto, el tiempo de ejecución será el siguiente:

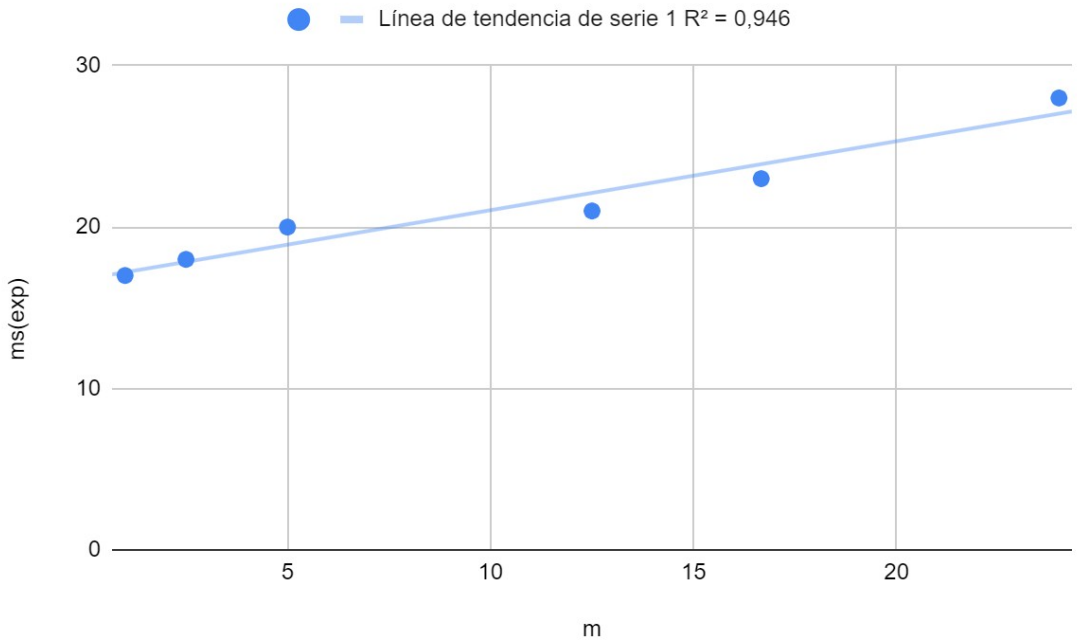
$$T(n,m)=O(n^2)+m * O(nm)=n^2+mnm=n^2+nm^2 \in O(nm^2)$$

Nos quedamos con el segundo término como el que más rápido crece porque los valores de n y m son bastante similares.

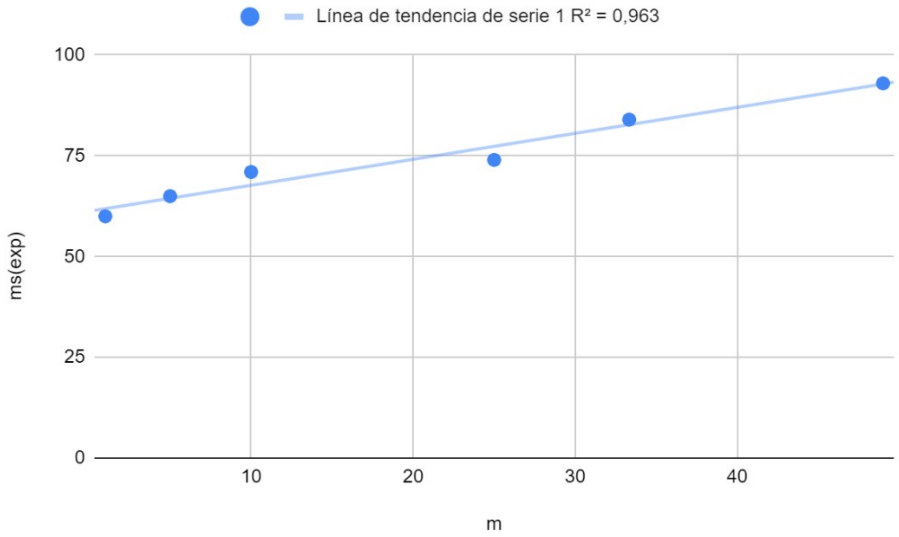
2.4 Estudio experimental del tiempo de ejecución

Para estudiar experimentalmente el tiempo de ejecución, generaremos matrices con valor n 25, 50, 100, 250 y 500. Para cada uno de estos casos, seleccionaremos los valores m de tal modo que tengamos m = n/10, n/5, n/2 y n/1,5. Además, generaremos 100 casos para cada valor n.

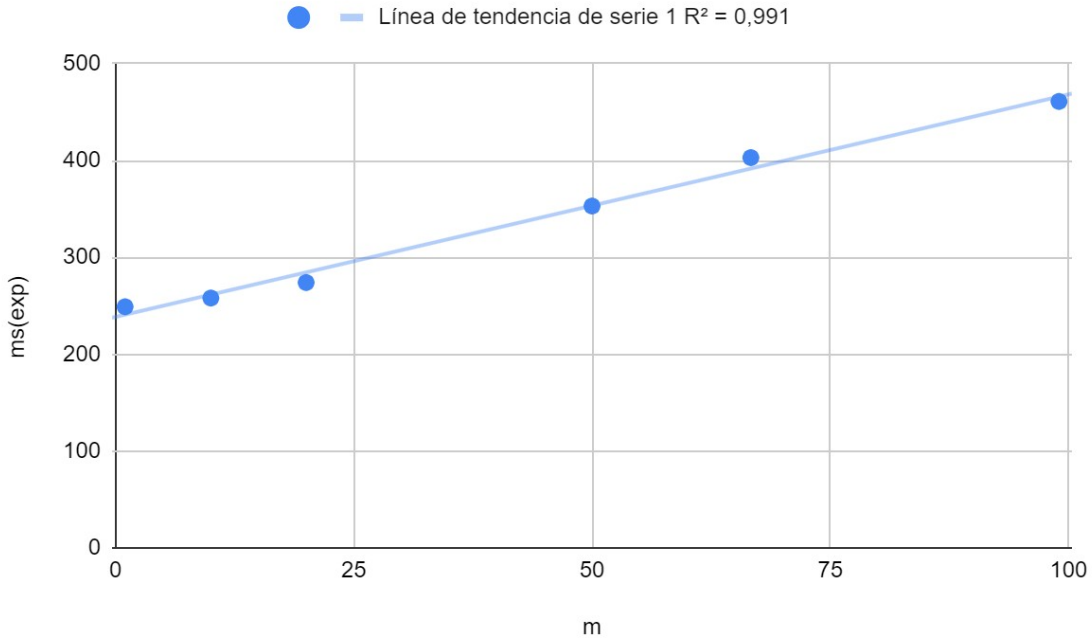
n = 25



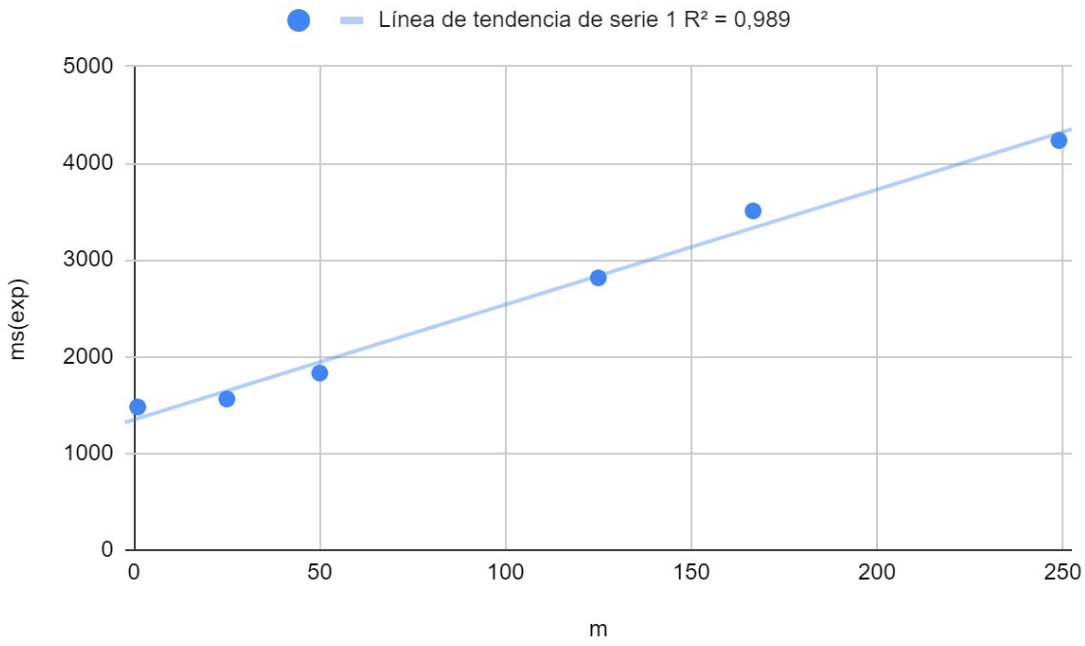
n = 50


Línea de tendencia de serie 1 R² = 0,963
100
75
50
25
0
ms(exp)
10
20
30
40
m

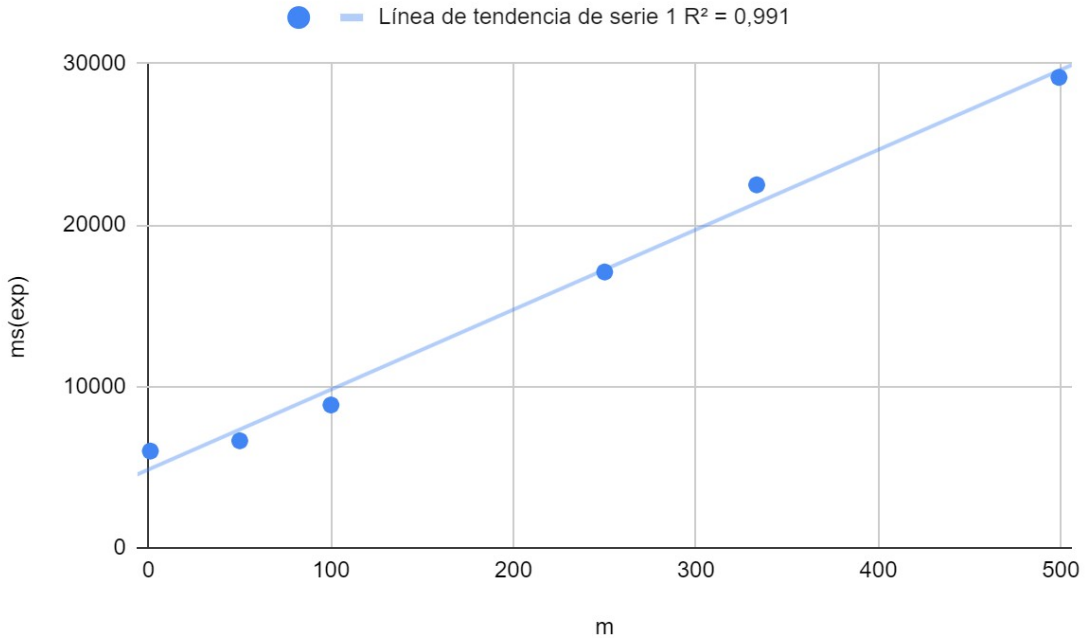
n = 100


Línea de tendencia de serie 1 R² = 0,991
500
400
300
200
100
0
ms(exp)
0
25
50
75
100
m

n = 250

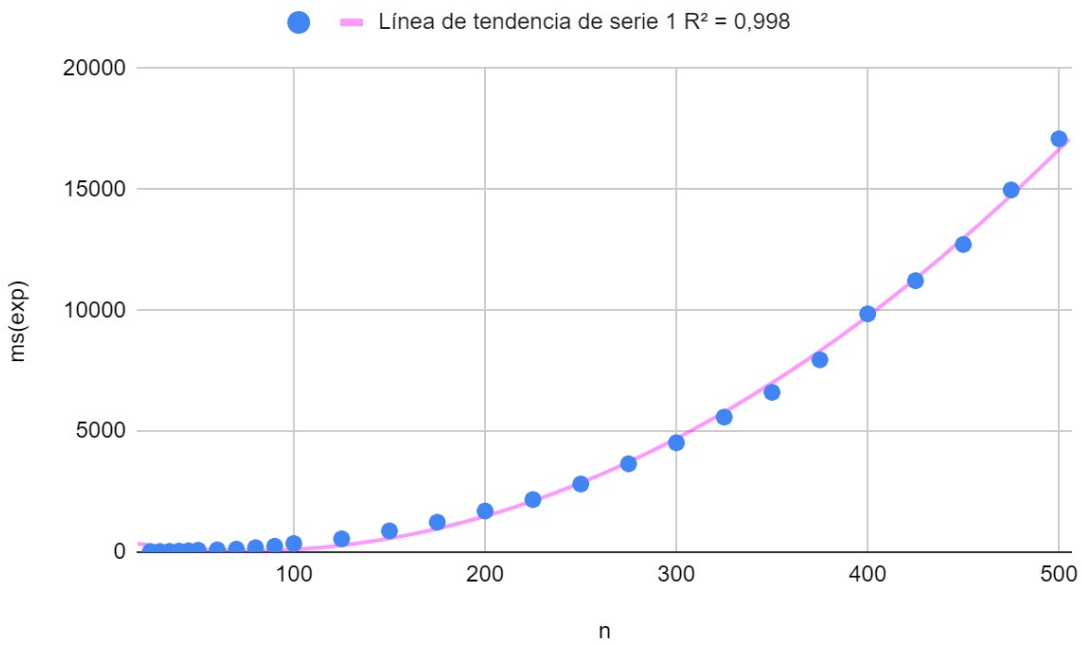


n = 500

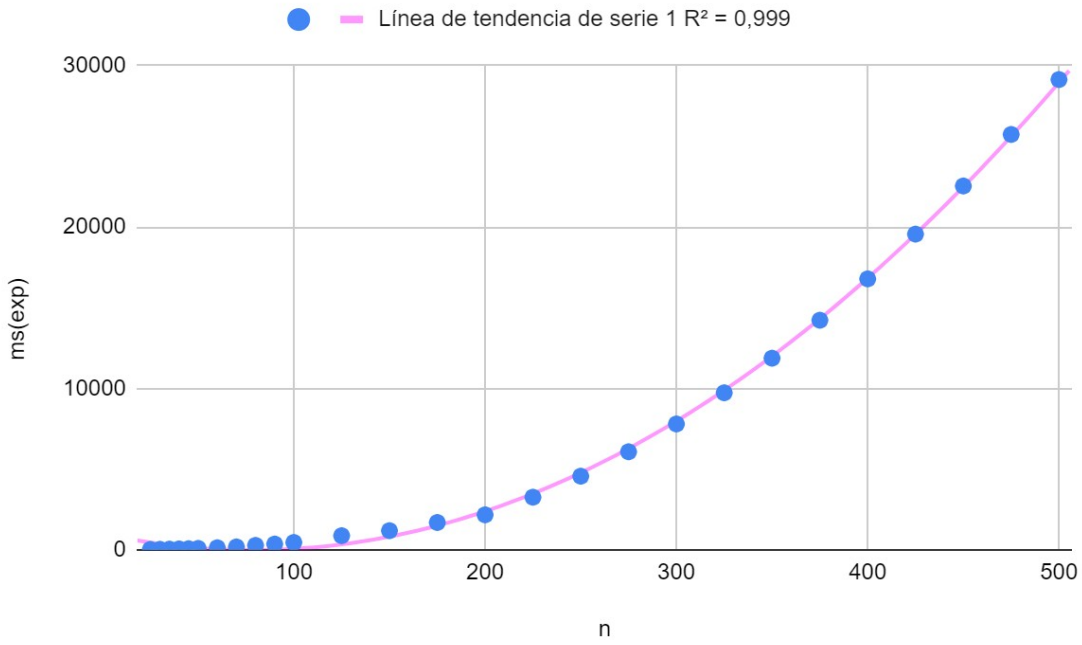


A continuación, mostraremos también los tiempos para n variable y m prefijada (m será n/2 y n-1). Los valores de n fluctúan entre 25 y 500. De nuevo generaremos 100 casos para cada caso particular (valor T=100):

m = n/2



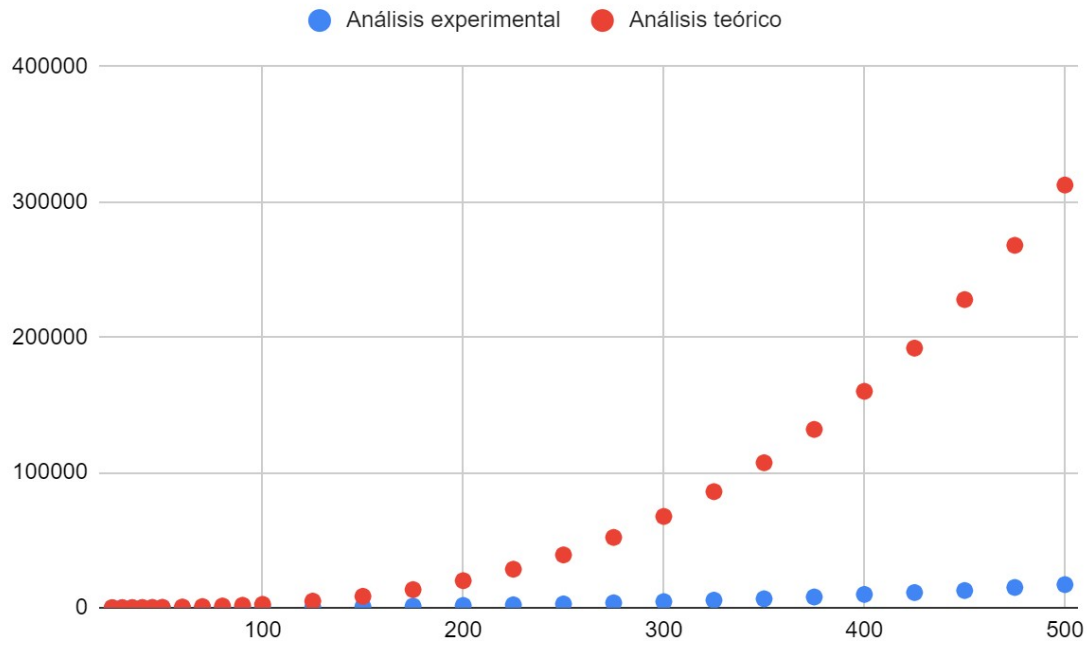
m = n-1



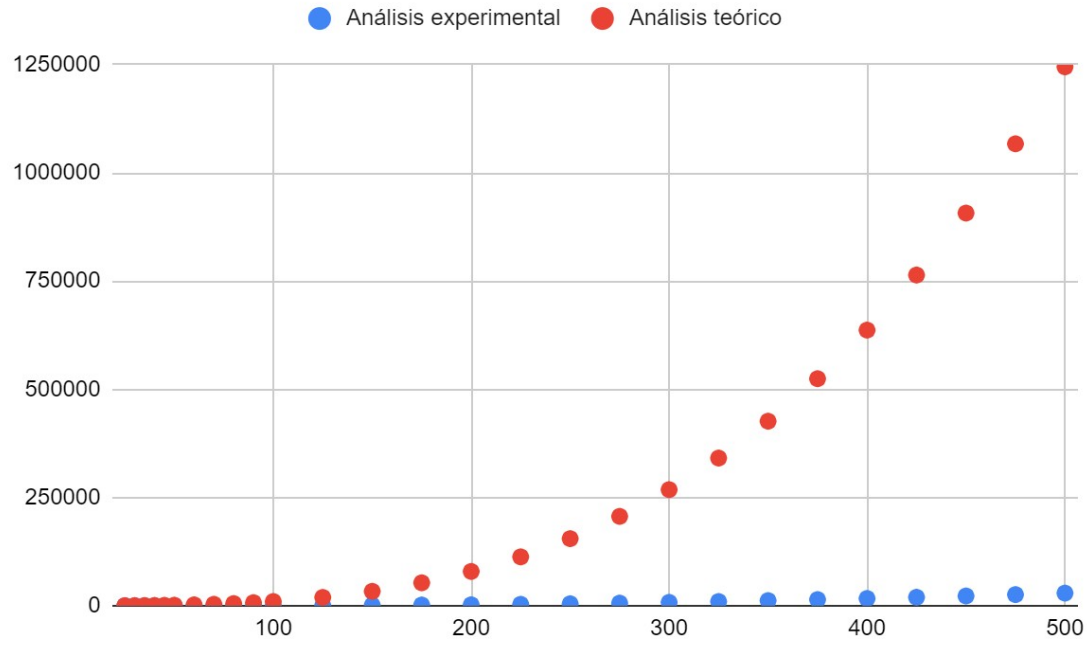
Realmente es interesante y relevante ver como el problema fluctúa con el cambio de las n, pues cambiar las m y dejar las n estáticas nos deja con una regresión polinómica poco importante y que fluctúa poco. Es por ello que en el contraste experimental-teórico nos centraremos en estas últimas dos gráficas.

2.5 Contraste teórico-experimental

m = n/2



m = n-1



Podemos observar en ambos gráficos como la cota teórica (que por defecto se pone en el peor caso) nunca se supera, y se registran valores bastante por debajo en el análisis experimental. El análisis teórico se queda más cerca del experimental cuando los valores de m son bajos respecto a n, es decir, cuando hay menos elementos a seleccionar. Esto se debe a que el análisis experimental se ha realizado en una máquina potente, por lo que los tiempos son relativamente bajos para casos grandes.

3. Backtracking

3.1 Pseudocódigo del algoritmo

En primer lugar, vamos a declarar una serie de variables que usaremos en la lógica del backtracking. En el comentario de al lado de cada variable se explica su significado:

```
N: entero  // Número total de casos a procesar
M: entero  // Presupuesto máximo disponible
C: entero  // Número de categorías de prendas disponibles
tablaPrecios: matriz de enteros  // Almacena los precios por modelo y
categoría
configuracionActual: array de enteros  // Índices de la configuración
actual de modelos
mejorConfiguracion:  array  de  enteros    //  Almacena  la  mejor
configuración encontrada
configuracionValida:  array  de  booleanos    //  Indica  si  una
       configuración es válida
tact: entero  // Costo total acumulado de la configuración actual
voa: entero  // Valor óptimo alcanzado hasta el momento
procesado: booleano  // Bandera para control de flujo de procesamiento
```

Cabe destacar que el modelo de **árbol escogido** ha sido un árbol n-ario, puesto que el binario quedaba descartado al no ser un problema de elección, y puesto que de cada tipo de prenda se pueden ofrecer hasta k modelos, consideramos que este tipo de árbol es el que mejor se adapta a nuestro problema.

Comenzaremos con la función Generar, que se encargará de avanzar la configuración actual de los modelos seleccionados en una categoría específica, incrementando el índice del modelo actual en la categoría dada (nivel) y actualizando el costo total acumulado (tact). Si ya había un modelo seleccionado antes, restamos del total su costo antes de añadir el nuevo. El código es el siguiente:

```
Generar(nivel)
    si configuracionActual[nivel-1] > 0 entonces
        tact -= tablaPrecios[nivel-1][configuracionActual[nivel-1]]
    incrementar configuracionActual[nivel-1]
    tact += tablaPrecios[nivel-1][configuracionActual[nivel-1]]
```

En la función Solución comprobaremos si el conjunto que hayamos obtenido hasta el momento forma una solución válida, es decir, si hemos llegado al último nivel de categorías y si el costo total no excede el presupuesto máximo:

```
Solucion(nivel) : booleano
    procesado := verdadero
    si nivel == C y tact <= M entonces
        configuracionValida[nivel-1] := verdadero
        devolver verdadero
    devolver falso
```

Incluimos también una función Criterio para comprobar si es posible seguir explorando desde el nivel actual, es decir, si aún no se han procesado todas los niveles y si no hemos superado el presupuesto inicial:

```
Criterio(nivel) : booleano
    devolver nivel < C y tact <= M
```

A continuación tenemos la función MasHermanos, la cual se encarga de comprobar si existen más configuraciones posibles al mismo nivel, es decir, si podemos probar otro modelo en la misma categoría sin retroceder a un nivel anterior, devolviendo “true” en caso afirmativo. Primeramente comprobamos si el nivel actual es 0, porque en ese caso no hay posibilidad de tener más hermanos. Después, comprobamos que la bandera “procesado” no esté activa, pues esto implicaría que la configuración actual no ha sido procesada completamente en llamadas anteriores a esta función. En este caso, comprobamos si el precio del modelo actual es menor que el almacenado en la mejor configuración, y en dicho caso actualizamos este valor de mejor configuración. Finalmente, reseteamos en todo caso la flag de procesado a false y devolvemos la verificación de si el índice del modelo actual es menor que el número total de modelos disponibles en esa categoría, en cuyo caso hay más modelos susceptibles de ser explorados y, por tanto, más hermanos disponibles. El pseudocódigo quedaría:

```
MasHermanos(nivel) : booleano
    si nivel == 0 entonces
        devolver falso
```

```
    si no procesado y tablaPrecios[nivel-1][configuracionActual[nivel-
1]] < mejorConfiguracion[nivel-1] entonces
                mejorConfiguracion[nivel-1]  :=  tablaPrecios[nivel-1]
[configuracionActual[nivel-1]]
    procesado := falso
    devolver configuracionActual[nivel-1] < tablaPrecios[nivel-1][0]
```

Ya finalizando, tendríamos la función Retroceder, que se encarga de volver a un nivel atrás en la búsqueda cuando no quedan modelos viables en la categoría actual o el criterio no se cumple. Se resta el costo del último modelo seleccionado del costo total y se reinicia el índice de modelo de esa categoría a cero, ajustando las configuraciones válidas y mejores de los niveles superiores según sea necesario:

```
Retroceder(nivel)
    tact -= tablaPrecios[nivel-1][configuracionActual[nivel-1]]
    configuracionActual[nivel-1] := 0
    si no configuracionValida[nivel-1] y nivel-1 > 0 entonces
        mejorConfiguracion[nivel-2] := configuracionActual[nivel-2]
    si nivel-1 > 0 entonces
        configuracionValida[nivel-2] := verdadero
    decrementar nivel
```

Por último, tenemos la función del método principal, Backtracking, la cual se nutre de las funciones anteriores para explorar todas las configuraciones posibles de modelos por categoría. Se inicia en el primer nivel y va profundizándose a medida que se cumplan las condiciones del criterio. Si se encuentra una solución mejor que la conocida hasta el momento, se actualiza el contenedor de la mejor solución, y se retrocede cuando no se alcanza una solución óptima o hay más configuraciones posibles. En caso de que el valor óptimo actual (voa) sea iguale al presupuesto, se ha encontrado la mejor solución y no es necesario comprobar ninguna más. El pseudocódigo queda:

```
Backtracking()
    nivel := 1
    tact := 0
    voa := 0
    fin := falso
    soa[C] : arreglo de enteros
    procesado := falso
    inicializar configuracionActual, soa, configuracionValida a cero o
falso según corresponda
    mientras no fin y nivel != 0 hacer
        Generar(nivel)
```

```
        si Solucion(nivel) y tact > voa entonces
            voa := tact
            copiar configuracionActual a soa
        fin si
        si Criterio(nivel) entonces
            incrementar nivel
        fin si
        si voa == M entonces
            fin := verdadero
        fin si
        mientras no MasHermanos(nivel) y nivel > 0 hacer
            Retroceder(nivel)
        fin mientras
    fin mientras
    copiar soa a configuracionActual
```

3.2 Programación del algoritmo

```
#include <iostream>
#include <cmath>

using namespace std;

// Variables globales para manejar el contexto del problema
int N; // Número total de casos a procesar
int M; // Presupuesto máximo disponible
int C; // Número de categorías de prendas disponibles
int** tablaPrecios; // Matriz que almacena los precios por modelo y
categoría
int* configuracionActual; // Array que almacena los índices de la
configuración actual de modelos

int* mejorConfiguracion; // Almacena la mejor configuración encontrada
bool* configuracionValida; // Indica si una configuración es válida

bool procesado; // Bandera para control de flujo de procesamiento

int tact; // Costo total acumulado de la configuración actual
int voa; // Valor óptimo alcanzado hasta el momento

// Función para actualizar la configuración actual de modelos
void Generar(int nivel, int configuracionActual[]){
    if (configuracionActual[nivel-1] > 0)
        tact -= tablaPrecios[nivel-1][configuracionActual[nivel-1]];
```

```
    configuracionActual[nivel-1]++;
    tact += tablaPrecios[nivel-1][configuracionActual[nivel-1]];
}

// Función para verificar si la configuración actual cumple con el presupuesto
bool Solucion(int nivel, int configuracionActual[]){
    procesado = true;
    if ((nivel == C) && (tact <= M)){
        configuracionValida[nivel-1] = true;
        return true;
    }
    return false;
}

// Función para determinar si se puede seguir explorando soluciones en el nivel actual
bool Criterio(int nivel, int configuracionActual[]){
    return (nivel < C) && (tact <= M);
}

// Función para verificar si existen más configuraciones posibles en el nivel actual
bool MasHermanos(int nivel, int configuracionActual[]){
    if (nivel == 0)
        return false;
     if (!procesado && tablaPrecios[nivel-1][configuracionActual[nivel-1]] < mejorConfiguracion[nivel-1])
                mejorConfiguracion[nivel-1]  =  tablaPrecios[nivel-1][configuracionActual[nivel-1]];
    procesado = false;
    return (configuracionActual[nivel-1] < tablaPrecios[nivel-1][0]);
}

// Función para retroceder a un nivel anterior en el proceso de búsqueda
void Retroceder(int& nivel, int configuracionActual[]){
    tact -= tablaPrecios[nivel-1][configuracionActual[nivel-1]];
    configuracionActual[nivel-1] = 0;
    if (!configuracionValida[nivel-1] && nivel-1 > 0){
        mejorConfiguracion[nivel-2] = configuracionActual[nivel-2];
    }
    else if (nivel-1 > 0){
        configuracionValida[nivel-2] = true;
    }
    nivel--;
}

// Implementa el algoritmo de backtracking para encontrar la mejor configuración posible
```

```
void Backtracking(int configuracionActual[]){
    int nivel = 1;
    tact = 0;
    voa = 0;
    bool fin = false;
    int soa[C];
    procesado = false;
    for (int i = 0; i < C; i++) {
        configuracionActual[i] = 0;
        soa[i] = 0;
        configuracionValida[i] = false;
    }
    while (!fin && nivel != 0) {
        Generar(nivel, configuracionActual);
        if (Solucion(nivel, configuracionActual) && tact > voa){
            voa = tact;
            for (int i = 0; i < C; i++)
                soa[i] = configuracionActual[i];
        }
        if (Criterio(nivel, configuracionActual))
            nivel++;
        else if (voa == M)
            fin = true;
        else
             while (!MasHermanos(nivel, configuracionActual) && nivel >
0)
                Retroceder(nivel, configuracionActual);
    }

    for (int i = 0; i < C; i++)
        configuracionActual[i] = soa[i];
}

int main(){
    cin >> N;
    for (int k = 0; k < N; k++) {
        cin >> M >> C;
        tablaPrecios = new int*[C];
        for (int i = 0; i < C; i++) {
             tablaPrecios[i] = new int[21]; // Suponemos un máximo de
20 modelos por categoría
        }

        for (int i = 0; i < C; i++) {
            cin >> tablaPrecios[i][0];
            for (int j = 1; j <= tablaPrecios[i][0]; j++)
                cin >> tablaPrecios[i][j];
        }

        configuracionActual = new int[C];
```

```
        mejorConfiguracion = new int[C];
        configuracionValida = new bool[C];
        Backtracking(configuracionActual);

        if (configuracionActual[0] == 0)
            cout << "no solution\n";
        else {
            int total = 0;
            for (int i = 0; i < C; i++)
                total += tablaPrecios[i][configuracionActual[i]];
            cout << total << '\n';
        }

        for (int i = 0; i < C; i++)
           delete[] tablaPrecios[i];
                  delete[] tablaPrecios;
           delete[] configuracionActual;
           delete[] mejorConfiguracion;
           delete[] configuracionValida;
        }
}
```

3.3 Estudio teórico del tiempo de ejecución

Cabe destacar que en este código, por su naturaleza arbórea, tendremos un caso mejor y peor claramente diferenciados, en función de cuan necesario sea recorrer el árbol para hallar la solución.

Funciones Generar, Solución, Criterio, MasHermanos y Retroceder

Todas tienen un orden constante sin mejor ni peor caso, pues se encargan de verificar condiciones o hacer comparaciones simples. Es decir, son de $O(1)$.

Función Backtracking

En el mejor caso, encontraremos en el primer nodo hoja que ya hemos gastado todo el dinero que había de presupuesto, y por tanto no será necesario seguir explorando ninguna rama más del árbol. Por tanto, podemos concluir que:

$$T_m(n) = 6 + 3n + 3n - 3 + 3 + n = 6 + 7n \in \Omega(N)$$

En el peor caso, tendremos que explorar el árbol entero, evaluando todas las posibles combinaciones de modelos para encontrar la solución óptima o para

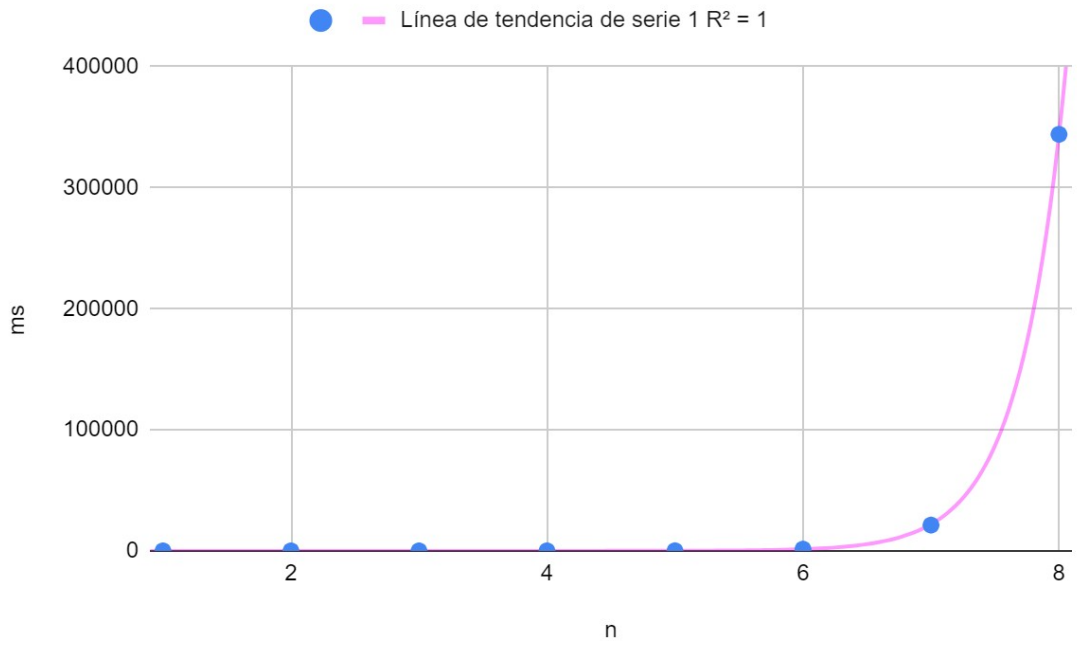
determinar que no existe solución. Siendo k el promedio de modelos disponibles de una categoría, el tiempo será:

$$T_M(n,k)=\frac{k^{(n+1)}-1}{k-1} \in O(k^n)$$

3.4 Estudio experimental del tiempo de ejecución

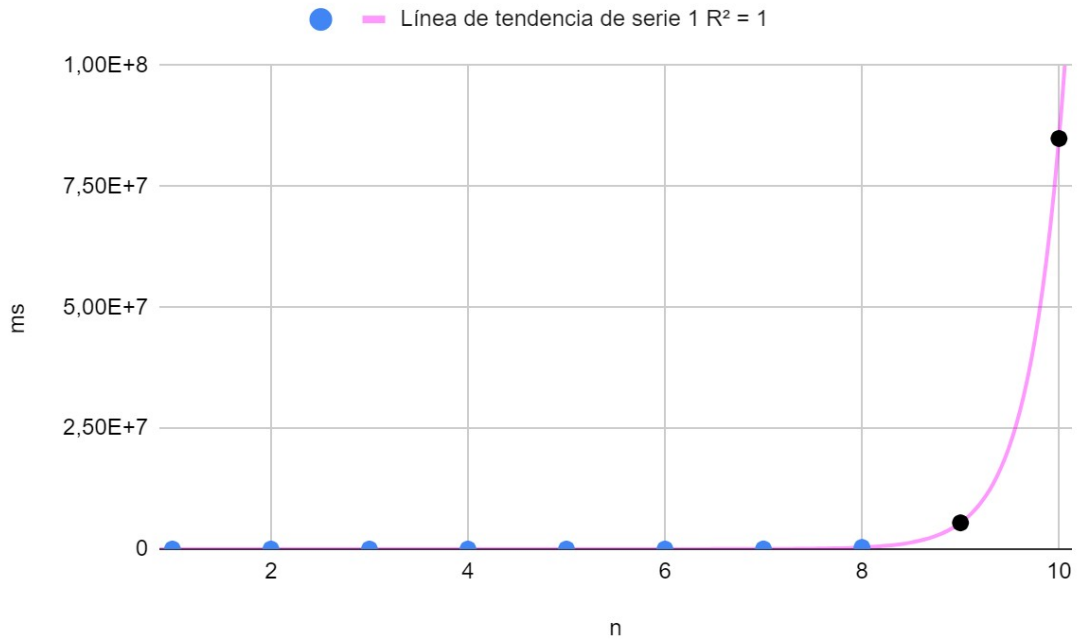
Para estudiar el tiempo de ejecución, fijaremos el número de modelos disponibles de una categoría, k, a 15. En esta ocasión, como el algoritmo puede llegar a tiempos exponenciales, únicamente trataremos con un caso de prueba por cada n que variemos. Iremos aumentando n de 1 en 1 hasta llegar a k.

Peor caso



Como podemos ver, a pesar de que se ha indicado que aumentaríamos n de 1 en 1, cortamos la gráfica en n=8. Esto lo hacemos porque en nuestra máquina tardaba demasiado, y ya con n=8 se fue a tiempos superiores a los 5 minutos. No obstante, abajo adjuntamos como quedaría el gráfico añadiendo n=9 y n=10 a partir de la ecuación de regresión que hemos obtenido del gráfico de arriba, la cual es la siguiente:

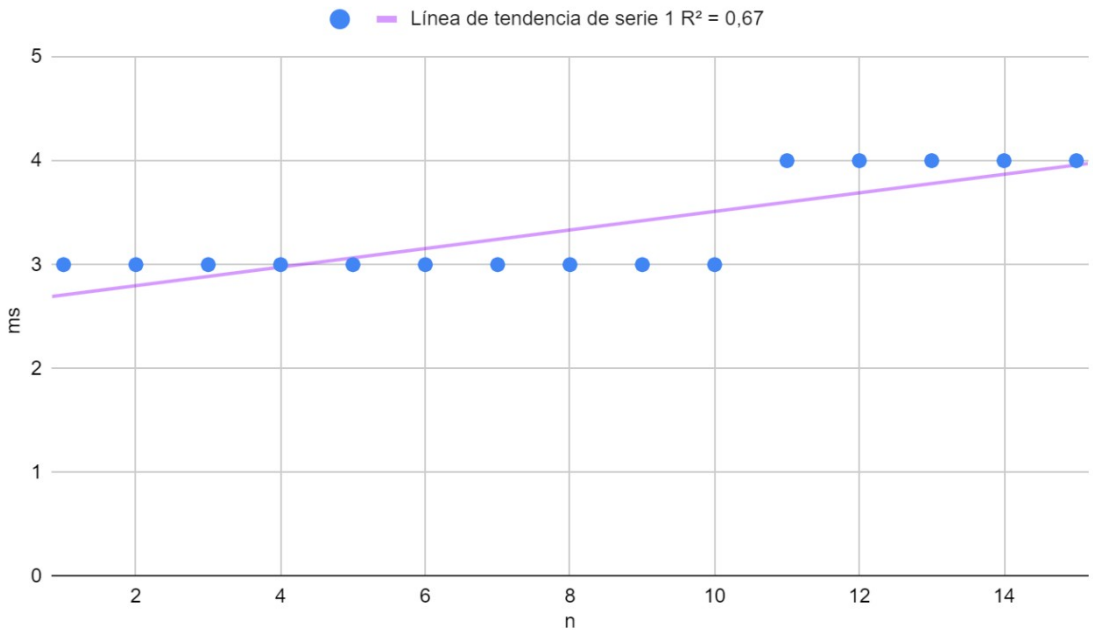
$$e(n)=9.68 * 10^{-5} * e^{(2,75\,n)}$$



Y aquí una tabla con los valores esperados hasta n=15:

n	tiempo (ms)
11	1328261042
12	20777498523
13	325014760874
14	5084086261278
15	79528489852623

Mejor caso

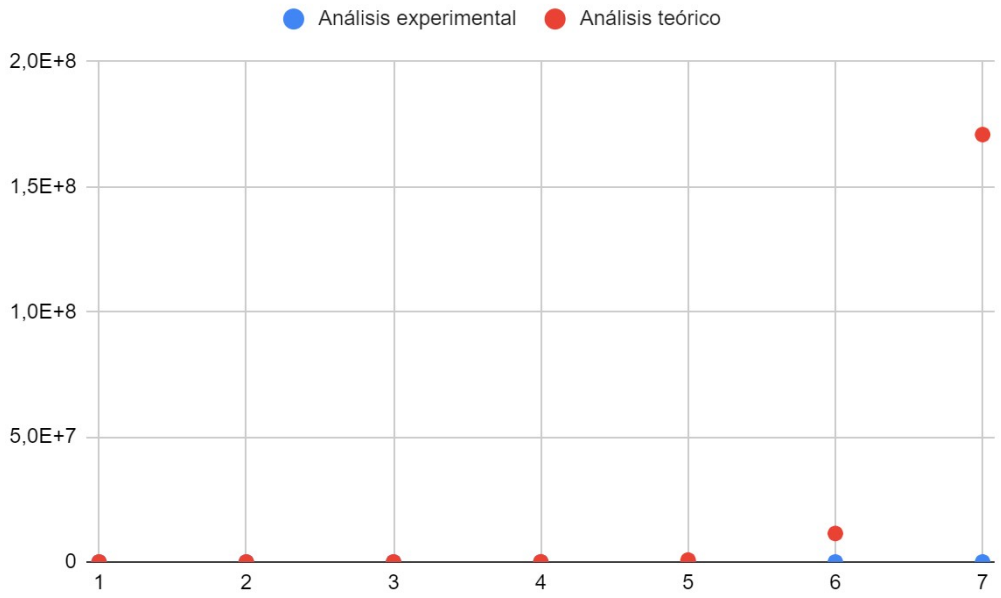


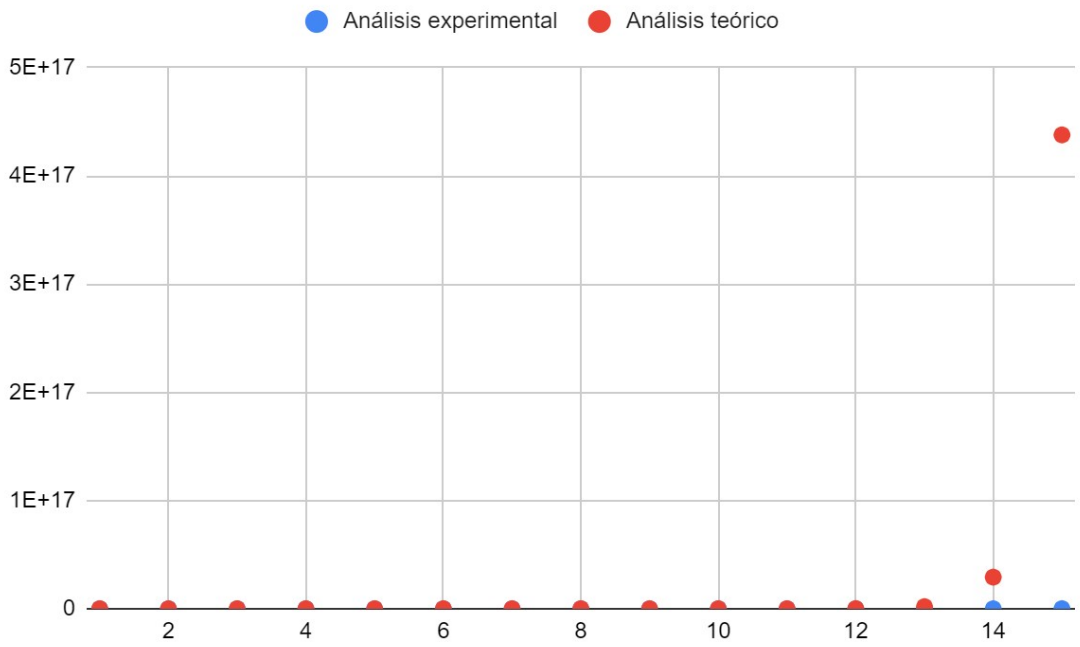
En el mejor caso, no es necesario explorar apenas el árbol, por lo que el tiempo de ejecución es prácticamente inmediato.

3.5 Contraste teórico-experimental

Peor caso

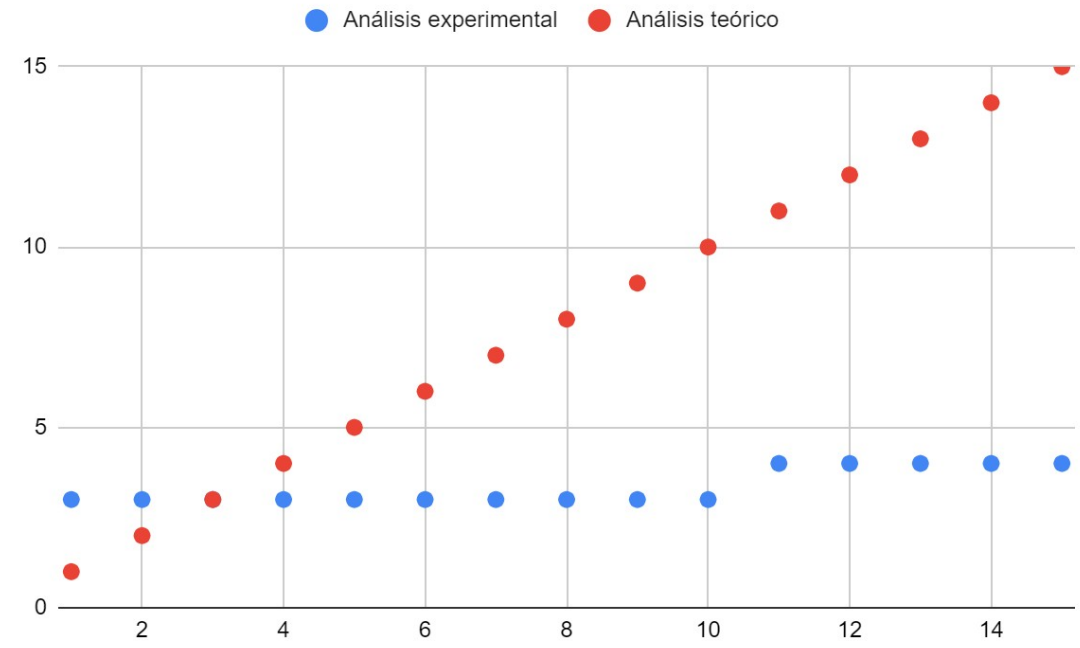
Debido a que las cotas son muy altas por el orden exponencial (k^n), mostraremos el gráfico primeramente hasta n=8, y después pondremos el gráfico entero:





Como podemos observar, la cota del estudio teórico se sitúa muy por encima de los datos recogidos con el análisis experimental, por lo que queda claro que es $O(k|n)$ es una cota correcta para el peor caso.

Mejor caso



En el mejor caso podemos ver que para n=1 y n=2 el análisis experimental supera al teórico. Esto se debe a que existe un tiempo prácticamente inherente a la ejecución de cualquier algoritmo en una máquina (que en caso de la nuestra era de 3ms). Este factor no es tenido en cuenta en el análisis teórico, por lo que para estos dos casos particulares vemos como no hace una cota superior. Sin embargo, para el resto, vemos como la cota de $\Omega(n)$ es una cota correcta para el resto de los casos.

4. Conclusiones

Este proyecto nos ha sido costoso de realizar, y hemos tenido que poner bastante empeño en realizarlo, especialmente en el ejercicio de Backtracking. Si bien el de Avance Rápido pudimos completarlo sin mucho problema, pues encontramos rápidamente la equivalencia entre los ejemplos de clase y nuestro problema, con el de Backtracking tardamos bastante en desarrollar correctamente todas las funciones auxiliares, y algunas como MasHermanos o Retroceder nos llevaron más de un día cada una. Seguramente también ha influido el hecho de que hemos empezado a realizar la práctica antes de terminar la teoría de Backtracking, por lo que hemos tardado más en entenderlo todo.

En cuanto a los análisis teóricos, fueron algo más sencillos que en el caso de la práctica de Divide y Vencerás, pero para los experimentales hemos tardado bastante en generar archivos para probar los diferentes tamaños, y ha supuesto un reto. No obstante, estamos bastante satisfechos con los resultados, porque además de que se cumplen las cotas teóricas también hemos conseguido que los gráficos se vean mejor usando otro programa distinto al de la práctica anterior, por lo que los datos han quedado dispuestos de forma muy visual y nos gusta bastante como ha quedado.

Estimamos alrededor de unas 35 horas en total para hacer el proyecto.