DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Oller, Luca	667/13	ollerr67@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este informe se explicará el desarrollo del código realizado para resolver los problemas pertenecientes al Trabajo Práctico 1 y la justificación de la complejidad de cada ejercicio.

Los problemas serán resueltos utilizando las técnicas algorítmicas conocidas como Backtrackin y programación dinámica

El código está desarrollado en C++ y los gráficos fueron generados a partir de un archivo de texto y Matplotlib.

Es importante destacar además que a medida que se resolvieron los problemas, las soluciones fueron probadas para verificar el funcionamiento del código.

Para justificar la complejidad de los algoritmos que resuelven los problemas se utiliza pseudocódigo con un comentario en cada linea con la complejidad y luego una justificación de la complejidad del algoritmo. Por último, se realizará una experimentación para comparar el funcionamiento del algoritmo con la complejidad estimada.

2. Problema I: Backtracking

El problema consiste en una entrada, un vector de números y se quiere pintar cada uno de ellos de color rojo o azul.

Los rojos deben ser una secuencia estrictamente creciente, mientras que los azules estrictamente decreciente.

Se busca minimizar la cantidad de elementos sin pintar.

En este ejercicio se pide el backtracking sin podas, su pseudocódigo y además se realizará una experimentación analizando, mejor caso, peor caso y caso promedio.

2.1. Idea general de la solución

La idea para resolver este problema es la de aplicar backtracking para analizar todas las soluciones posibles y al no utilizar podas la complejidad es la cantidad de nodos del árbol, es decir $O(3^n)$ porque es un árbol ternario balanceado y n es la longitud de la entrada.

Esto es así porque en cada nodo que no es hoja, hay tres opciones posibles pintar de rojo, de azul o no pintar de ningún color, y en el peor caso el algoritmo debe recorrer todo el árbol, es decir donde para cada elemento sea posible agregar el mismo al vector azul, vector rojo o no agregarlo.

El backtracking se realiza de la siguiente manera:

- Comenzamos desde la raíz llamando a una función que comience coloreando el primer elemento de rojo, luego otra función que coloree el primero de azul y luego otra que intente no colorearlo.
- Luego para cada nodo intermedio se verifica que el elemento siguiente cumpla con las condiciones para pintarlo de rojo, es decir que sea más grande que el último elemento pintado de rojo, se calcula al máximo entre los elementos pintados del estado actual y el resultado de la llamada recursiva pintándolo de rojo. Para el azul se verifica que cumpla la condición de ser menor al elemento anterior pintado de azul y se calcula al máximo entre los elementos pintados del estado actual y el resultado de la llamada recursiva pintándolo de azul. Y finalmente se llama al máximo entre los elementos pintados del estado y el resultado de la función recursiva con el siguiente elemento sin pintar.
- Cuando se llega a una hoja, se verifica si el elemento cumple con las condiciones mencionadas en el item anterior y se retorna la longitud del vector menos la cantidad de elementos pintados, dando como resultado la cantidad de elementos sin pintar.

Este algoritmo obtiene la mejor solución debido a que analiza todas las posibles soluciones que podrían haber. De entre las soluciones analizadas, devuelve la mejor que se encontró, siendo la mejor (o mejor dicho una de las mejores, ya que puede haber varias soluciones óptimas) solución posible.

2.2. Pseudocódigo

El pseudocódigo consiste de dos algoritmos, uno para inicializar el backtracking, llamado primer caso y otro resolver que es el backtracking que resuelve las soluciones de los nodos intermedios y las hojas.

Hay que destacar que el resolver calcula la cantidad de elementos pintados y luego se retorna la longitud de entrada restándole la cantidad de elementos pintados, dándonos los elementos sin pintar.

Algorithm 1 primer caso

```
1: procedure PRIMER CASO(vector<int>entrada,vector<int>azul,int ultimo_azul, vector<int>rojo,int ultimo_rojo,int actual,int pintados) → res : int
2:     pinto de rojo el primer elemento                                                    ▷ O(1)
3:     pintados=resolver(con el primer elemento pintado de rojo);                          ▷ O(resolver)
4:     despinto de rojo el primer elemento                                                 ▷ O(1)
5:     Pinto de azul el primer elemento                                                    ▷ O(1)
6:     pintados = max(pintados,resolver con el primer elemento pintado de azul);           ▷ O(resolver)
7:     Despinto de azul el primer elemento                                                 ▷ O(1)
8:     pintados=max(pintados,resolver(con el primer elemento sin pintar));                 ▷ O(resolver)
9:     return entrada.size()-pintados;
```

Complejidad: $\mathcal{O}(resolver)$+ $\mathcal{O}(resolver)$ + $\mathcal{O}(resolver)$ = $\mathcal{O}(resolver)$
Justificación: Pintar y despintar un elemento es $\mathcal{O}(1)$ y luego se realizan 3 llamadas a resolver que su complejidad será analizada en el siguiente pseudocódigo.

Algorithm 2 Resolver

```
1: procedure RESOLVER(vector<int>entrada,vector<int>azul,int ultimo_azul, vector<int>rojo,int ultimo_rojo,int actual,int pintados) → res : vector<vector<int>>
2:     pintadosAux=0;
3:     if estoy en el último elemento(caso base) then
4:         if el elemento actual es mayor al ultimo rojo pintado then
5:             return pintados+1;
6:         if el elemento actual es menor al ultimo azul pintado then
7:             return pintados+1;
8:         return pintados;
9:     if el elemento actual es mayor al ultimo rojo pintado then
10:        pintadosAux=max(pintadosAux,resolver(con el elemento actual pintado de rojo));
11:    if el elemento actual es menor al ultimo azul pintado then
12:        pintadosAux=max(pintadosAux,resolver(con el elemento actual pintado de azul));
13:    pintadosAux=max(pintadosAux,resolver(con el elemento actual sinpintar));
14:    return entrada.size();
```

Complejidad: $\mathcal{O}(3^n)$
Justificación: El algoritmo se llama a sí mismo como máximo 3 veces para cada elemento del arreglo, esto nos da 3 opciones posibles para cada elemento formando el árbol de soluciones ternario completo y como su cantidad de nodos es de 3^n, recorrerlo nos da una complejidad de $\mathcal{O}(3^n)$

2.3. Experimentación

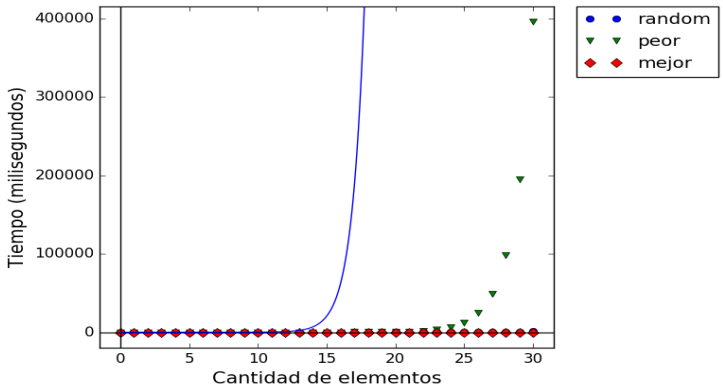
Como el único parámetro de entrada del algoritmo es la cantidad de elementos del vector entrada, como experimentación se correrá el programa variando este número entre 1 y 50 con valores aleatorios.

Luego, se identificó un mejor caso y peor caso.

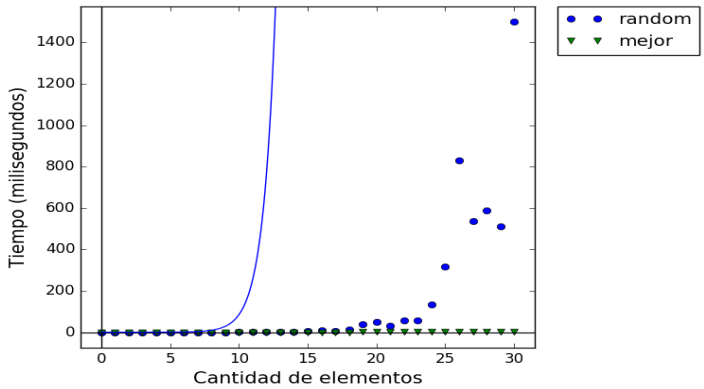
El mejor caso del algoritmo se presenta cuando el vector tiene todos los elementos repetidos. En este caso lo que ocurre es que el algoritmo va a pintar el primer elemento de rojo, el segundo de azul y viceversa en otra rama y luego no va a hacer nada. Y en la llamada de no hacer nada del final va a volver a suceder que pinte uno de azul y otro de rojo y luego nada. Esto sucede porque las secuencias son estrictamente crecientes y estrictamente decrecientes, entonces si los elementos son todos iguales, se puede pintar solo dos elementos.

Mientras que el peor caso fue tomar una sucesión estrictamente creciente.

Y finalmente se realizó un gráfico comparando estas tres instancias y la curva de la complejidad calculada multiplicado por una constante para mostrar que el algoritmo cumple con esta complejidad.



Podemos ver que en el grafico, no se presenta ninguna diferencia muy notable entre las ejecuciones del algoritmo entre el mejor caso y el caso random, se puede concluir que esto se debe a que es poco probable que aleatoriamente ocurra que un caso de los casos random sea similar a un peor caso. Además se puede ver la curva 3^n.



Por último se decidió colocar un gráfico que compare el mejor caso con el caso random para poder verificar que el mejor caso sea efectivamente el mejor y además para ver que sus tiempos se corresponden con la función 3^n

3. Problema II: Backtracking con podas

En este ejercicio se pide reutilizar el backtracking agregándole una o más podas.

En este caso se optó por agregar una poda que consiste en verificar si las soluciones parciales son peores que una solución óptima.

Esto se logra a partir de agregar un if al comienzo de las llamadas recursivas, como se verá en el pseudocódigo, que verifique que la variable que almacena la mejor solución es mayor o igual a la solución de esa rama sumado la cantidad total de elementos para llegar a la hoja, es decir asumiendo que vamos a pintar todos los elementos restantes de la entrada recibida.

Si no se cumple esta condición, el código interrumpe su ejecución retornando la solución actual y se dice que esa rama es podada.

Esto impacta en el performance del código como veremos a continuación.

3.1. Pseudocódigo

El pseudocódigo es similar el ejercicio 1, la función primer caso es exactamente igual. Mientras que a resolver se agregan algunos ifs para realizar la poda.

Algorithm 3 Resolver

```
1: procedure RESOLVER(vector<int>entrada,vector<int>azul,int ultimo_azul, vector<int>rojo,int ultimo_rojo,int actual,int pintados) → res : vector<vector<int>>
2:     pintadosAux=0;
3:     if pintados+entrada.size()-actual<=pintados_aux ∧ pintados_aux!=-1 ∧ actual>1 then
4:         return pintados_aux;
5:     else
6:         if estoy en el último elemento(caso base) then
7:             if el elemento actual es mayor al ultimo rojo pintado then
8:                 return pintados+1;
9:             if el elemento actual es menor al ultimo azul pintado then
10:                return pintados+1;
11:            return pintados;
12:        if el elemento actual es mayor al ultimo rojo pintado then
13:            pintadosAux=max(pintadosAux,resolver(con el elemento actual pintado de rojo));
14:        if el elemento actual es menor al ultimo azul pintado then
15:            pintadosAux=max(pintadosAux,resolver(con el elemento actual pintado de azul));
16:        pintadosAux=max(pintadosAux,resolver(con el elemento actual sinpintar));
17:    return pintados_aux;
```

Complejidad: $\mathcal{O}(3^n)$

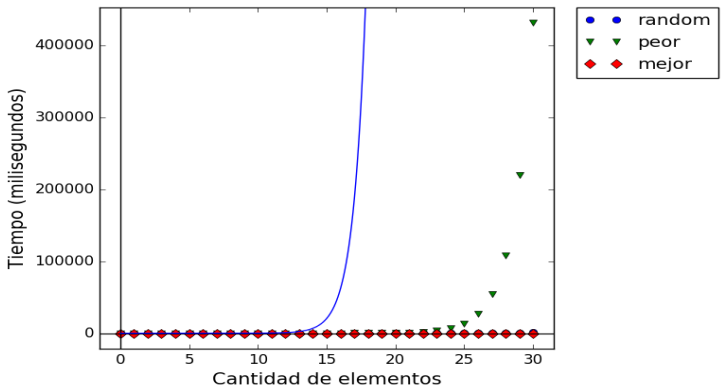
Justificación: A pesar de la poda que mejora la mayoría de casos, hay instancias para la cual la poda no mejora la complejidad, por lo tanto el algoritmo sigue teniendo la misma complejidad que en el ejercicio 1.

3.2. Experimentación

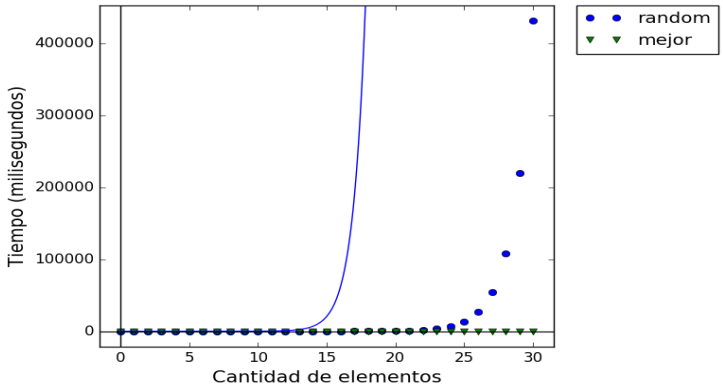
Para este problema se hicieron 3 generadores de casos: caso random, peor caso y mejor caso.

Los casos son los mismos que en el Ejercicio 1.

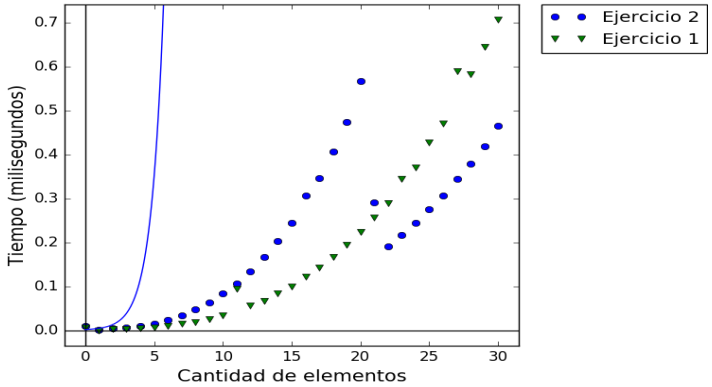
Para levantar los datos, se ejecutaron los tres casos con n variando entre 1 y 30 saltando de a 1. Los datos obtenidos son los siguientes:



Y de la misma manera que en el ejercicio anterior, se optó por graficar el mejor caso y el random juntos para ver que cumpla con la complejidad.



Y para verificar el impacto que tiene la poda en los tiempos de ejecución se decidió comparar el ejercicio1 con el ejercicio2.



4. Problema III: Programación dinámica

En este punto se pide resolver el mismo problema de pintar la secuencia, pero utilizando programación dinámica y con una complejidad de a lo sumo $O(n^3)$.

4.1. Idea general de la solución

La solución que ideé es a partir de calcular dinámicamente por un lado las subsecuencias crecientes máximas, mientras que por otro lado se calcularon las subsecuencias decrecientes de manera dinámica.

La complejidad de calcular las subsecuencias crecientes máximas es de $O(n^2)$ que se justificará en el pseudocódigo.

Mientras que calcular las subsecuencias decrecientes máximas es de igual complejidad a la de calcular las crecientes máximas.

El algoritmo consiste en primero obtener la longitud máxima de las soluciones crecientes máximas, a partir de ese valor obtener los índices de las subsecuencias crecientes de dicha longitud, y guardarlas en la matriz como filas.

Una vez que se tienen estas soluciones en la matriz, se procede a crear otra matriz en la que se guardan los índices de las soluciones en la posición de esta nueva matriz como 1 si existe y como 0 si no, es decir si el elemento fue pintado para la solución se coloca un 1 en su posición y un 0 si no, esto nos va a servir para luego verificar eficientemente con el índice para saber si fue pintado en el momento de calcular la subsecuencia decreciente.

Luego se calcula las subsecuencias decrecientes máximas pero solo de los elementos que no estén pintados y para verificar que dichos elementos no estén pintados, para esto se realiza un for que itere sobre las filas de la matriz de índices, pasándole la fila de dicha matriz y el vector de entrada a la función que calcula la subsecuencia decreciente máxima.

Esta función verifica antes de intentar pintar el elemento si en el vector de índices el elemento figura como pintado y de ser así, ignora el elemento y va al siguiente, de esta forma uno se puede asegurar que solo se pinten los elementos que están sin pintar y los que fueron pintados de rojo previamente son ignorados.

4.2. Funciones recursivas

Se utilizan dos funciones recursivas, una para calcular la máxima subsecuencia creciente y otra para la mínima subsecuencia creciente, al llamar una vez a la subsecuencia creciente máxima, una vez que se tiene la longitud y las soluciones que cumplen, se debe llamar al arreglo de entrada sin los elementos que pertenecen a dicha subsecuencia máxima.

Entonces defino F, G y H y sea v el vector de entrada, sea w el vector de entrada quitándole los elementos que pintó cada subsecuencia creciente y sea M la matriz que contiene las subsecuencias crecientes máximas.

$F(i, v) =$ Tamaño de la subsecuencia creciente máxima que incluye a v[i] como su último elemento

$$F(i,v) = \begin{cases} 1 + Max_{j=1..i-1}\{F(j)\} & \text{si } (\forall\ 1 < j < i)\ v_j < v_i \\ 1 & \text{sino} \end{cases}$$

$G(i, w) =$ Tamaño de la subsecuencia decreciente máxima que incluye a w[i] como su último elemento

$$G(i,w) = \begin{cases} 1 + Max_{j=1..i-1}\{G(j)\} & \text{si } (\forall\ 1 < j < i)\ w_j > w_i \\ 1 & \text{sino} \end{cases}$$

$H(i, v, m) =$ Longitud de v - Máxima subsecuencia creciente + máxima subsecuencia decreciente hasta i

$$H(i, v, m) = \begin{cases} v.size() - Max_{j=0..i-1}\{G(v - M(i))\} - M(i).size() & \text{si } i > 0 \\ 1 & \text{sino} \end{cases}$$

Donde G toma como parámetro el vector que se obtiene de restar como conjuntos el vector v que es la entrada y una subsecuencia posible que es una fila de la matriz.

4.3. Pseudocódigo

Algorithm 4 subsec_decrec_aux

```
1: procedure SUBSEC_DECREC_AUX(vector<int>v, vector<int>indices) → res : int
2:     max=0;
3:     vector <int>vec(v.size(),-1);
4:     vec[0]=1;
5:     for i=1.. v.size() do                                              ▷ O(n)
6:         if el elemento no esta en una solucion creciente then          ▷ O(1)
7:             for j=0.. i do                                             ▷ O(n)
8:                 if v[j] >v[i] ∧ la longitud de la subsecuencia hasta i-1 es menor o igual a la longitud
   hasta j  then
9:                     incremento el contador de elementos de la subsecuencia de v[i]
10:    for k=0.. v.size() do                                              ▷ O(n)
11:        if max<vec[k] then
12:            max = vec[k];
```

Complejidad: $\mathcal{O}(n^2)$
Justificación: Hay dos for anidados de $\mathcal{O}(n)$ y luego un for separado de $\mathcal{O}(n)$. Esto nos da $\mathcal{O}(n)$ * $\mathcal{O}(n)$+ $\mathcal{O}(n)$ = $\mathcal{O}(n^2)$

Algorithm 5 subsec_crec_aux

```
1: procedure SUBSEC_CREC_AUX(vector<int>v, vector <pair <int,vector <int> > > vec) → res : int
2:     max=0;
3:     vec[0].first=1
4:     vec[0].second.push_back(-1);
5:     for i=1.. v.size() do                                              ▷ O(n)
6:         for j=0.. i do                                                 ▷ O(n)
7:             if v[i] >v[j] ∧ la longitud de la subsecuencia hasta i-1 es menor o igual a la longitud hasta
   j  then
8:                 incremento el contador de elementos de la subsecuencia de v[i]
9:                 me guardo el indice del elemento anterior para luego rearmar la secuencia en el vector
   de vec[i]
10:    for k=0.. v.size() do                                              ▷ O(n)
11:        if max<vec[k].first then
12:            max = vec[k].first;
```

Complejidad: $\mathcal{O}(n^2)$
Justificación: Hay dos for anidados de $\mathcal{O}(n)$ y luego un for separado de $\mathcal{O}(n)$. Esto nos da $\mathcal{O}(n)$ * $\mathcal{O}(n)$+ $\mathcal{O}(n)$ = $\mathcal{O}(n^2)$

Algorithm 6 f_rec

```
1: procedure F_REC (vector<int>v, vector <pair <int,vector <int> > > vec, matriz mat, int max, int
   m, int indice)
2:     if max==0 then
3:         return;
4:     else
5:         for j=0...vec[i].second().size do                                   ▷ O(n)
6:             if el elemento es el penúltimo de una subsecuencia máxima then
7:                 guardo el elemento en la máx-1 posición de mat
8:                 f_rec(v,vec,mat,max-1,j);
```

Complejidad: $\mathcal{O}(n)$
Justificación: La justificación de porque la complejidad de rearmar todas las soluciones en estas llamadas recursivas es $\mathcal{O}(n)$ es porque a lo sumo hay v.size() cantidad de soluciones posibles de longitud v.size(), es decir n^2 elemenos que llenarían toda la matriz y hay muchos elementos del vector que se descartan por la guarda del if porque la estructura de vec en sus vectores guarda todas las subsecuencias crecientes, aún las de menor longitud, entonces la cantidad de llamadas recursivas es $\mathcal{O}(1)$ respecto de la cantidad de elementos, dándonos en total complejidad $\mathcal{O}(n)$.

Algorithm 7 buscar_crecientes

```
1: procedure BUSCAR_CRECIENTES (vector<int>v, vector <pair <int,vector <int> > > vec) → res : int
2:     int max = subsec_crec_aux(v,vec);                                   ▷ O(n^2)
3:     matriz mat;                                                         ▷ O(n^2)
4:     int indice_matriz=0;
5:     for i=0...v.size() do                          ▷ O(n) son 3 for anidados y da O(n^3)
6:         if el elemento es el último de una subsecuencia máxima then
7:             guardo el elemento en la máx posición de mat
8:             for j=0...vec[i].second().size do                           ▷ O(n)
9:                 if el elemento es el penúltimo de una subsecuencia máxima then
10:                    guardo el elemento en la máx-1 posición de mat
11:                f_rec(v,vec,mat,max-1,j);                               ▷ O(n)
12:    for i=0...v.size() do                                               ▷ O(n)
13:        for j=0...v.size() do                                           ▷ O(n)
14:            if mat[i][j]==0 then
15:                mat[f][c]=mat[f-1][c]; ▷ acomodo las soluciones haciendo estas operaciones para que sean más facil de ver O(1)
16:    matriz indices;                                                     ▷ O(n^2)
17:    for i=0...v.size() do                                               ▷ O(n)
18:        for j=0...v.size() do                                           ▷ O(n)
19:            if si el elemento mat(i,j) está pintado then
20:                indices[i][mat[i][j]]=pintado
21:            else
22:                indices[i][mat[i][j]]=no pintado
23:    int solucion_decr =0;
24:    for i=0...v.size() do                                               ▷ O(n)
25:        int auxdec= subsec_decrec_aux(v,indices[i]);                    ▷ O(n*n)
26:        if auxdec>solucion_decr then
27:            solucion_decr=auxdec;
28:    return v.size()-solucion_decr-max;
```

Complejidad: $\mathcal{O}(n^2)$
Justificación: Hay tres for anidados de $\mathcal{O}(n)$ y luego dos for anidados de $\mathcal{O}(n)$. Esto nos da $\mathcal{O}(n)$ * $\mathcal{O}(n)$ *$\mathcal{O}(n)$+ $\mathcal{O}(n)$*$\mathcal{O}(n)$ = $\mathcal{O}(n^3)$ para la parte de instanciar las matrices. La función f_rec realiza el procediimiento de la linea 5 hasta la 10 recursivamente llamándose a si misma hasta que su cuarta variable sea 0 y escribe una posición de la matriz en $\mathcal{O}(1)$. Para la parte de calcular las subsecuencias decrecientes se realiza un for $\mathcal{O}(n)$ mientras que la operación interna de calcular la subsecuencia decreciente es $\mathcal{O}(n^2)$ lo que nos queda como $\mathcal{O}(n^3)$. Finalmente se tiene que la complejidad es $\mathcal{O}(n^3)$+ $\mathcal{O}(n^3)$ = $\mathcal{O}(n^3)$.

4.4. Experimentación

Para la experimentación se instanció variando la cantidad de elementos desde 1 hasta 50, diferenciando, como en el backtracking en el mejor , peor caso y caso aleatorio.

Los casos random y mejor son iguales al punto 1 y 2, mientras que el peor caso es cuando para obtener la solución se deba considerar la mayor cantidad de subsecuencias posibles. Y la forma de garantizar esto es hacer que la secuencia sea de la pinta (1,3,2,5,4,7,..) esto nos da como resultado que el algoritmo intenta pintar cada elemento porque puede pertenecer a una subsecuencia creciente o decreciente, dándonos la mayor cantidad posible de subsecuencias crecientes y decrecientes a analizar.

Se pudieron evaluar más instancias porque al tener una mejor complejidad el tiempo de ejecución era notablemente más bajo. Se obtuvieron los siguientes resultados. Como en el 1 y 2, el peor caso "plancha.ª las curvas mejor y random y por eso hay otro gráfico aparte para verificar que cumplan con la complejidad $\mathcal{O}(n^3)$.