Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2017

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1

Integrante	LU	Correo electrónico
Nicolás Ippolito	724/14	ns_ippolito@hotmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2017

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1

Integrante	LU	Correo electrónico
Nicolás Ippolito	724/14	ns_ippolito@hotmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Ejercicio 1

1.1 Descripción del problema

Dada una tira de números, se debe pintar cada uno de ellos de color rojo o azul en busca de minimizar la cantidad de números sin pintar. Los criterios para el pintado son:

- Los números rojos deben formar una secuencia estrictamente creciente
- Los números azules deben formar una secuencia estrictamente decreciente

1.2 Solución propuesta

La solución propuesta resuelve el problema utilizando backtracking. La implementación se realizó en C++.

1.2.1 Resolución

La técnica de backtracking consiste en pensar conceptualmente en un árbol que contenga todas las posibles soluciones parciales y finales. En mi algoritmo, sería un árbol ternario donde el mismo sigue la siguiente idea:

- Partir de la raíz (ningún número pintado)
- Comenzar por el primer número -pintarlo de azul, rojo, o no pintarlo- y recorrer recursivamente cada una de esas 3 posibilidades para el elemento siguiente (3 ramas). Al recorrer la rama de 'no pintar', el paso recursivo debe considerar que ahora la cantidad de no pintados es 1 más.
- A partir del segundo elemento, recorrer recursivamente cada una de las 3 ramas sólo si ésta es válida, en otras palabras, si puede ser pintada dadas las correspondientes secuencias rojas y azules de esa rama. Notar que la rama correspondiente a 'no pintar' siempre se recorrerá (y al recorrerla, el paso recursivo debe considerar que ahora tu cantidad de no pintados es 1 más).
- Al llegar a una hoja que esté en el último nivel (cuando la altura del árbol es igual a la longitud de la secuencia de entrada), si la cantidad de no pintados en esa rama es menor a la que ya tenés, entonces ésta será la mejor solución. Si no, entonces me quedo con la solución que tenía antes (aquella con la que comparé la que me dio esta hoja).

De esta forma, voy a recorrer todas las combinaciones de secuencias rojas y azules válidas quedándome con la que minimiza la cantidad de elementos no pintados.

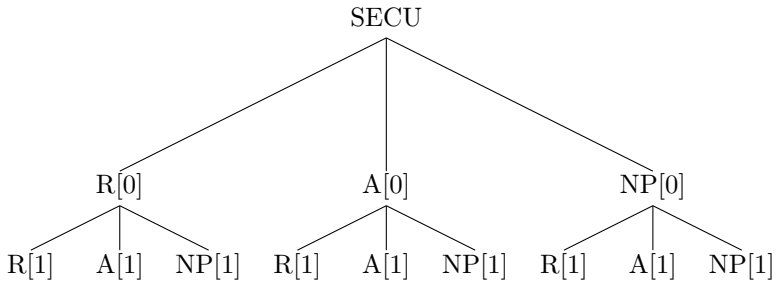
1.2.2 Pseudo Código

Algorithm 1 Ej1

```
procedure EJ1(vector tiraNum, int solucion)
    if Llego a una hoja del último nivel posible then
        if CantSinPintar(hoja) < solucion then
            solucion = CantSinPintar(hoja)
        return solucion
    if PuedoPintarRojo then
        Paso Recursivo rama Roja
    if PuedoPintarAzul then
        Paso Recursivo rama Azul
    Paso Recursivo rama no pintar al elemento
```

1.3 Complejidad teórica

Dado que en cada paso estoy recorriendo 3 alternativas (ramas distintas) -azul, rojo, o no pintar- y al ser un algoritmo recursivo, puedo representar los pasos de mi algoritmo con un árbol ternario:



Así, el arbol se seguirá abriendo hasta llegar a una altura que sea igual a la longitud de la secuencia de entrada. Si bien algunas ramas se cortarán mucho antes dado que mi algoritmo no recorre las ramas inválidas, la complejidad puede ser acotada superiormente por la cantidad de hojas de un árbol ternario completo ($O(3^n)$).

Formalmente, como mi algoritmo en cada paso recursivo realiza operaciones constantes, el mismo se representa con la recurrencia $T(N) = 3T(N-1) + 1$. Si la analizamos:

$T(N) = 3T(N-1)+1 = 3(3T(N-2)+1)+1 = 3^2T(N-2)+3+1 = 3^2(3T(N-3)+1)+3+1 = 3^3T(N-3)+3^2+3+1 = ... = 3^NT(N-N) + 3^{N-1} + 3^{N-2} + ... + 3^1 + 3^0 = \sum_{i=0}^{N-1} 3^i = \frac{1-3^N}{1-3} = \frac{3^N-1}{2} = O(3^n)$

Luego, al haber deducido mi cota temporal, quiero demostrarla por inducción:

Quiero ver que (T(N) $\leq c * 3^N - d$) para algún $c > 0$, donde $d \geq 0$ es una constante

Caso base: T(1) = 3 * T(0) + 1 = 1, y tomando c = 1 y d = 0, 1 $\leq$ 1 * 3^0 - 0 = 1

Paso Recursivo: Por **hipótesis inductiva**, T(N-1) $\leq$ c * 3^{N-1} − d

Luego, T(N) = 3T(N-1) + 1 $\leq$ 3(c * 3^{N-1} − d) + 1 = c * 3^N − 3d + 1 $\leq$ c * 3^N − d

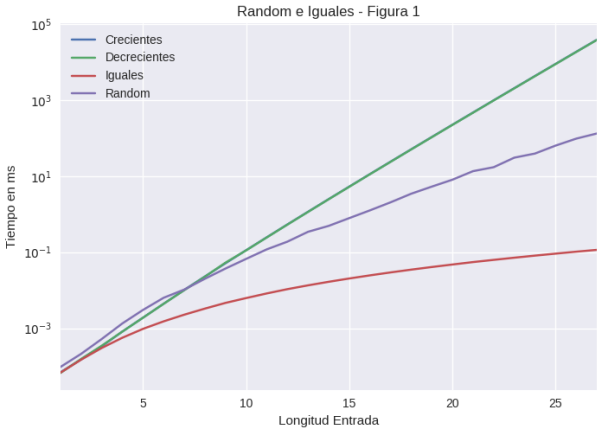
Siempre y cuando d $\geq \frac{2}{3}$

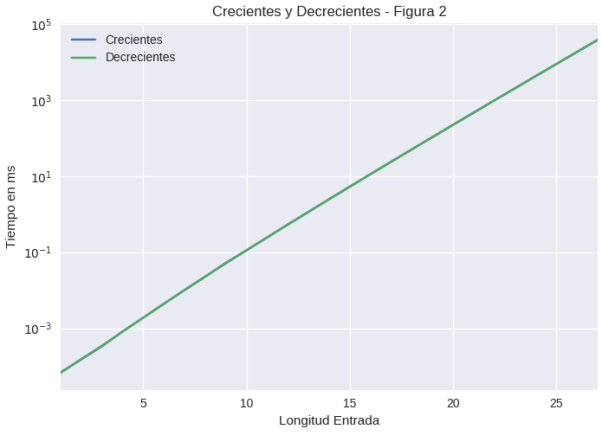
Por lo tanto, la cota temporal de mi algoritmo es $O(3^n)$.

1.4 Experimentación

Los experimentos fueron realizados en una computadora con un procesador Intel I5 4440 y 8GB de memoria RAM.

Para los mismos se utilizaron 40 entradas distintas generadas al azar (o sea, cada entero se generó con la librería random de C++, y está en el rango [0, 100000]) para cada longitud entre 1 y 27.
Por otro lado, para esas mismas longitudes se realizaron 10 entradas distintas crecientes, 10 decrecientes, y 10 cuyos elementos eran todos iguales.
Luego, para cada longitud correspondiente según el tipo de test (random, creciente o decreciente) se tomó la media para poder graficarlo.





Notar que las entradas crecientes y decrecientes (figura 2) tienen un tiempo de cómputo muchísimo más alto; esto tiene sentido ya que en estos casos en cada llamada recursiva se puede llegar al fondo de la rama correspondiente a pintarlo de rojo o de azul (según creciente o decreciente), lo que incrementa la cantidad de llamadas recursivas. Luego, se puede ver que el **peor caso** del algoritmo es con entradas crecientes y decrecientes.
Por otro lado, analizando la figura 1 queda claro que con entradas cuyos números son todos iguales se tiene el **mejor caso**, ya que se harían muchas menos llamadas recursivas porque en la mayoría de los casos no se va a poder pintar ni de azul ni de rojo.

2 Ejercicio 2

2.1 Descripción del problema

Se debe encontrar una forma de podar al árbol resultante de analizar el algoritmo de Backtracking del Ejercicio 1, para mejorar la eficiencia del mismo.

2.2 Solución Propuesta

2.2.1 Resolución

Para podar aun más mi árbol, lo que se me ocurrió es podar aquellas ramas que no van a llegar a una solución mejor que la que ya tengo (necesito primero que mi árbol llegue a una hoja del último nivel). Por lo tanto, si veo que la rama que quiero seguir recursivamente, aun pintando todos los que elementos restantes no mejoraría la solución que ya tengo, entonces mi algoritmo deja de avanzar por la misma.

De esta forma, tengo una poda que mejora la eficiencia de mi algoritmo (lo que quedará claro en la experimentación).

2.2.2 Pseudo Código

Algorithm 2 Ej2

```
procedure EJ2(vector tiraNum, int solucion)
    if Llego a una hoja del último nivel posible then
        if CantSinPintar(hoja) < solucion then
            solucion = CantSinPintar(hoja)
        return solucion
    if PuedoPintarRojo and PuedoLlegarASolMásÓptima then
        Paso Recursivo rama Roja
    if PuedoPintarAzul and PuedoLlegarASolMásÓptima then
        Paso Recursivo rama Azul
    if NoLleguéAHoja or (LleguéAHoja and ParaSolMásÓptima) then
        Paso Recursivo rama no pintar al elemento
```

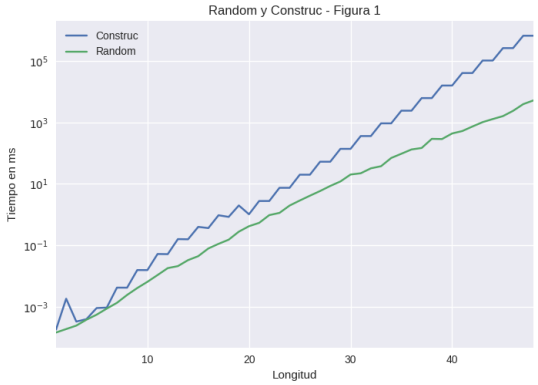
ParaSolMásÓptima es distinto que PuedoLlegarASolMásÓptima ya que el primero tiene en cuenta que si entra a ese IF entonces ahora va a haber un elemento más no pintado, mientras que en el segundo si entra al IF va a seguir teniendo la misma cantidad de no pintados que antes.

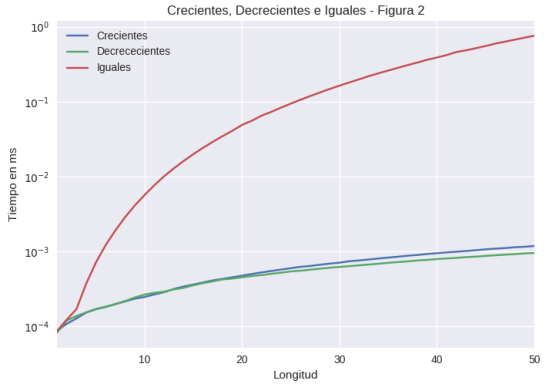
2.3 Complejidad teórica

Si bien la poda elegida aumenta bastante la eficiencia de mi algoritmo de Backtracking, en el peor caso el mismo no podaría nada o no tendría una cantidad de podas considerables como para que la cota de complejidad disminuya. Luego, se mantiene la complejidad teórica en el peor caso del Ejercicio 1, que es $O(3^n)$

2.4 Experimentación

Para este Ejercicio se realizaron experimentaciones iguales a las del Ejercicio 1, solo que en este caso las secuencias crecientes, decrecientes e iguales se hicieron para cada longitud en el rango [1, 50].
Mientras tanto, las entradas random se hicieron hasta longitud 48 para poder graficarse junto a otras instancias que son las generadas de la siguiente forma: secu[i] = secu[$i-1$] + 3 si i es par, y si es impar secu[i] = secu[$i-1$] - 1 (Ej: [1, 4, 3, 6, 5, 8, 7, 10]). A estas últimas secuencias las llamo "Construc". Fueron elegidas ante la posibilidad de que haya muchos llamados recursivos y pueda ser un caso malo del algoritmo.
Para ambos gráficos se utilizó escala logarítmica en el Eje Y (ya que había mucha diferencia entre construc y random, y entre iguales y crecientes/decrecientes).

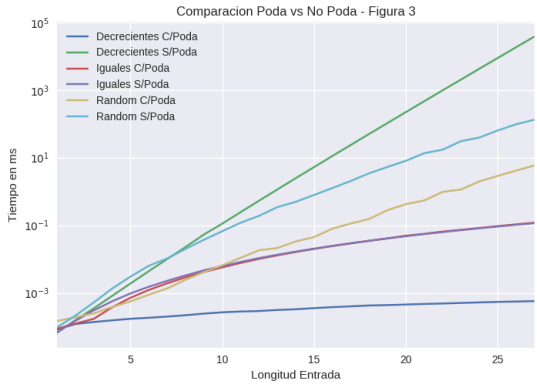




En la figura 2 se puede ver que los tiempos para instancias crecientes, decrecientes o iguales son mucho más pequeños y su crecimiento temporal a medida que va creciendo la longitud de la instancia es muy pequeño. Esto se debe a que, en el caso creciente la primer solución que va a encontrar va a ser 0 no pintados (llega al último nivel en la parte izquierda del árbol) y luego no va a entrar a ningún otro paso recursivo (ya que nunca se va a cumplir que $sinpintar < res$), y en el caso decreciente la primera solución óptima se encuentra pintando uno de rojo y luego todos de azul, por lo que una vez que llega tampoco entra a ningún otro paso recursivo. Luego, el **mejor caso** se obtiene con entradas crecientes y decrecientes.

Por otro lado, en la figura 1 vemos que para instancias Random y Construc el tiempo a medida que crece en n aumenta muchísimo. En particular, se ve que de las distintas instancias que generé para esta experimentación, el **peor caso** de mi algoritmo es con entradas Construc.

2.4.1 Comparación con Ejercicio 1



Analizando la figura 3 se puede ver que, exceptuando el caso en el que todos los números son iguales, todas las instancias que generé son más veloces cuando implementás una poda en el algoritmo, sobretodo cuando las entradas se encuentran ordenadas de forma creciente o decreciente.
Para el gráfico decidí omitir las entradas crecientes ya que su tiempo de cómputo es muy cercano al de las entradas decrecientes (y la herramienta que utilicé para graficar volvía a repetir colores cuando tenía más de 6 curvas diferentes)

3 Ejercicio 3

3.1 Descripción del problema

Se debe resolver el problema del Ejercicio 1 utilizando la técnica de programación dinámica. La complejidad temporal del algoritmo utilizado debe ser a lo sumo $O(n^3)$.

3.2 Solución Propuesta

3.2.1 Resolución

Para poder encontrar una resolución con programación dinámica, se debe encontrar una forma de relacionar a una instancia del problema con subinstancias del mismo. Luego, la fórmula siguiente hace lo descripto:

$$f(i,r,a) = \begin{cases} 0 & i = 0 \\ min(f(i-1,i,a), f(i-1,r,i), f(i-1,r,a)+1) & i \neq 0 \land \text{PuedoPintarRojo} \land \text{PuedoPintarAzul} \\ min(f(i-1,i,a), f(i-1,r,a)+1) & i \neq 0 \land \text{PuedoPintarRojo} \land \neg\text{PuedoPintarAzul} \\ min(f(i-1,r,i), f(i-1,r,a)+1) & i \neq 0 \land \neg\text{PuedoPintarRojo} \land \text{PuedoPintarAzul} \\ f(i-1,r,a)+1 & i \neq 0 \land \neg\text{PuedoPintarRojo} \land \neg\text{PuedoPintarAzul} \end{cases}$$

f(i, r, a) = "cantidad mínima de no pintados utilizando los primeros i elementos, suponiendo que r es el último rojo pintado (r > i) y a es el último azul pintado (a > i)"

Donde mi problema original es: f(n, r, a) = "cantidad mínima de no pintados en los primeros n elementos suponiendo que ninguno se pintó antes (ya que r, a > n)"

Aclaración: Cuando digo "r es el último rojo pintado (r > i)" por ejemplo, esto significa que es el último rojo pintado más cerca de i yendo para la derecha, y no el último rojo pintado de toda la secuencia (el que estaría más a la derecha de todo)

3.2.2 Pseudo Código

Algorithm 3 Ej3

```
procedure EJ3(vector tiraNum, int n)
    vector[n][n+1][n+1] cube ← matriz de 3 dimensiones inicializada con -1 en todos sus elementos
    int r ← n
    int a ← n
    int res ← Ej3Aux(tiraNum, cube, n − 1, n, r, a)
    return res
```

$PuedoRojoYAzul = (ultRo > n-1 \lor tiraNum[i] < tiraNum[ultRo] \land ultAz > n-1 \lor tiraNum[i] > tiraNum[ultAz])$
$PuedoSoloRojo = ((ultRo > n-1 \lor tiraNum[i] < tiraNum[ultRo]) \land \neg(ultAz > n-1 \lor tiraNum[i] > tiraNum[ultAz]))$
$PuedoSoloAzul = (\neg(ultRo > n-1 \lor tiraNum[i] < tiraNum[ultRo]) \land (ultAz > n-1 \lor tiraNum[i] > tiraNum[ultAz]))$

Algorithm 4 Ej3Aux

```
procedure EJ3AUX(vector tiraNum, vector(vector(vector))) cubo, int i, int n, int ultRo, int ultAz)
    int c ← 0
    int v ← 0
    int b ← 0
    if i = −1 then                                          ▷ Caso base
        return 0
    if cubo[i][ultRo][ultAz] ≠ −1 then                      ▷ Si ya calculé el subproblema, lo devuelvo
        return cubo[i][ultRo][ultAz]
    if ultRo > n − 1 ∨ tiraNum[i] < tiraNum[ultRo] then     ▷ Si se puede pintar de rojo, calculo el resultado
                                                              pintándolo de rojo
        c ← Ej3Aux(tiraNum, cubo, i − 1, n, i, ultAz)
    if ultAz > n − 1 ∨ tiraNum[i] > tiraNum[ultAz] then     ▷ Si se puede pintar de azul, calculo el resultado
                                                              pintándolo de azul
        v ← Ej3Aux(tiraNum, cubo, i − 1, n, ultRo, i)
    b ← Ej3Aux(tiraNum, cubo, i − 1, n, ultRo, ultAz) + 1   ▷ Encuentro el resultado sin pintarlo
    if PuedoRojoYAzul then  ▷ Si lo podía pintar de rojo y de azul, tomo el mínimo entre las 3 posibilidades
        cubo[i][ultRo][ultAz] ← min(c, b, v)
    else if PuedoSoloRojo then                              ▷ Si sólo lo podía pintar de rojo, tomo el mínimo entre pintarlo de rojo
                                                              y no pintarlo
        cubo[i][ultRo][ultAz] ← min(c, b)
    else if PuedoSoloAzul then                              ▷ Si sólo lo podía pintar de azul, tomo el mínimo entre pintarlo de azul
                                                              y no pintarlo
        cubo[i][ultRo][ultAz] ← min(v, b)
    else                                                    ▷ Si no se podía pintar, devuelvo el mínimo de no pintarlo
        cubo[i][ultRo][ultAz] ← b
    return cubo[i][ultRo][ultAz]
```

3.3 Complejidad teórica

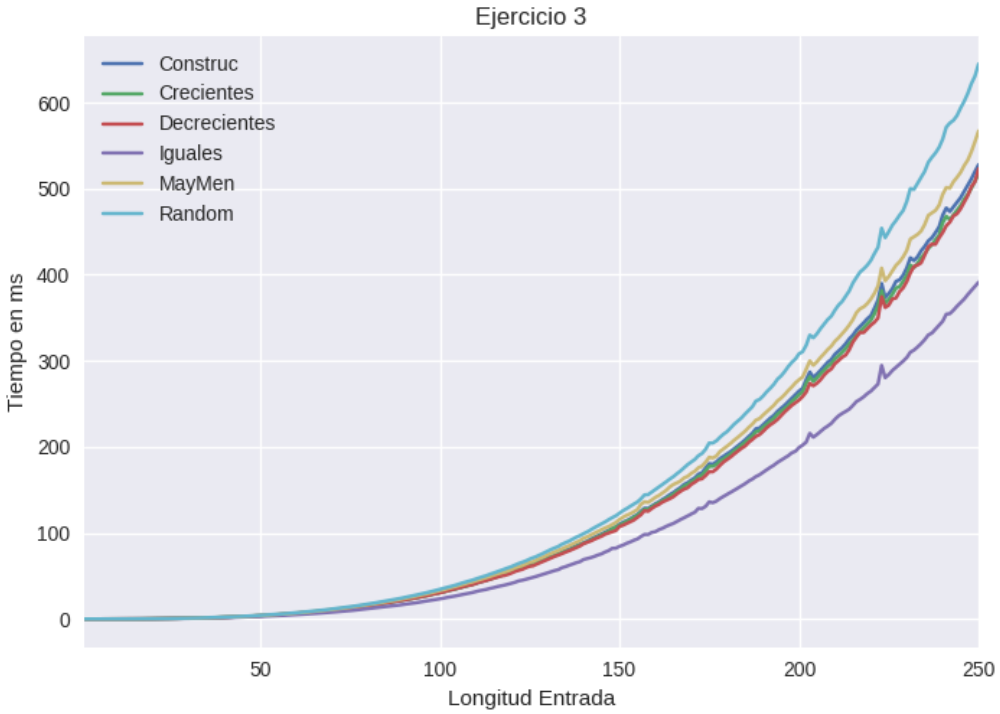
Mi algoritmo hace a lo sumo tantas llamadas recursivas como posiciones tiene mi cubo (matriz de tres dimensiones). Luego, en el peor caso se calculan todas las posiciones de mi matriz (que es de $n * n + 1 * n + 1$, lo que da un total de $n^3 + 2n^2 + n$) y esa cantidad pertenece a $O(n^3)$. Además, cada llamada recursiva cuesta $O(1)$.

Ahora, por más que la cantidad de llamadas recursivas totales sea $O(n^3)$, la complejidad podría ser mucho peor ya que algunos subproblemas que ya calculé podrían volverse a calcular, lo que haría que la complejidad se torne exponencial.
Pero por como está implementado mi algoritmo (y esa es la idea de programación dinámica), cada vez que calculo un subproblema lo guardo y cuando en otra rama del árbol (el generado al hacer el seguimiento de mi función recursiva propuesta) este es solicitado, lo devuelvo en vez de hacer las respectivas llamadas recursivas.

Finalmente, la cantidad de llamadas totales que hago es $O(n^3)$ y al simplemente devolver los subproblemas ya calculados en lugar de seguir aplicando recursión en cada subproblema (sumado a que en cada paso recursivo sólo hago cosas constantes i.e $O(1)$), mi complejidad teórica es esa ($O(n^3)$).

3.4 Experimentación

Para este ejercicio se experimentó con 40 entradas distintas generadas al azar (con la librería random de C++, cada número en el rango [0. 100000]) para cada longitud entre 1 y 250. Se corrió el algoritmo 5 veces en cada una de las entradas y se tomó el promedio de las mismas.
Luego, para esas longitudes también se realizaron tests con 10 entradas distintas crecientes, 10 decrecientes, 10 formadas con arreglos donde para las posiciones impares i, el elemento es mayor que todos los anteriores, y para las posiciones pares j, el elemento es menor que todos los anteriores (Ej: [50, 49, 51, 48, 52, 47]), 10 en las cuales todos los elementos son iguales y por último 10 en los cuales secu[i] = secu[$i-1$] + 3 si i es par, y si es impar secu[i] = secu[$i-1$] - 1 (Ej: [1, 4, 3, 6, 5, 8, 7, 10]). A estas últimas secuencias las llamo "Construc" (como en el Ejercicio 2). Así mismo, cada una de las entradas también se corrió 5 veces y se promedió.



Puede apreciarse observando el gráfico que, de todos los distintos tipos de instancias que observé, el **mejor caso** de mi algoritmo está cuando la entrada es una secuencia en la que todos los números son iguales.
Por otro lado, si bien en el resto de las instancias los tiempos son similares, cuando la entrada es random el algoritmo tiene un tiempo de cómputo más alto, lo que lo haría el **peor caso** de entre mis instancias generadas.

4 Ejercicio 4

4.1 Solución Propuesta

4.1.1 Resolución

La función propuesta para resolver el problema con programación dinámica es distinta a la del Ejercicio 3:

$$f(r,a) = \begin{cases} 0 & r = a \\ \max([f(r,i) | i < a \wedge PuedoPintarAzul]) + 1 & a > r \\ \max([f(i,a) | i < r \wedge PuedoPintarRojo]) + 1 & a < r \end{cases}$$

f(r, a) = "Cantidad máxima de elementos pintados suponiendo que el índice r es el último rojo pintado y el índice a es el último azul pintado"

Mi problema original es g(n) = n - max($[f(r,a) \mid$ a, r $\in [0, n-1]]$)

La idea es crear una matriz de (n+1 * n+1) ya que la columna 0 significa que no hay azules pintados y la fila 0 que no hay rojos pintados (o sea, veo los índices de la secuencia de entrada de 1 a n) que vaya guardando los subproblemas y luego elegir el máximo de ellos (que sería la máxima cantidad de pintados). Por ejemplo, en el caso $a > r$, debo mirar el máximo para la izquierda de esa misma fila que se pueda pintar, y sumarle 1 (sería: dado que pintaste el a de azul, mirar cuál es tu valor máximo de pintados dado que el último azul pintado fue el $a-1$, $a-2$,..., y sabiendo que estás en la fila correspondiente a pintar r rojos)
Ejemplo de cómo quedaría la matriz con la entrada [100, 101, 102, 99]:

$$M = \begin{bmatrix} \mathbf{0} & 1 & 1 & 1 & 2 \\ 1 & \mathbf{0} & 2 & 2 & 3 \\ 2 & 2 & \mathbf{0} & 3 & 4 \\ 3 & 3 & 3 & \mathbf{0} & 4 \\ 1 & 2 & 2 & 2 & \mathbf{0} \end{bmatrix}$$

4.1.2 Pseudo Código

Algorithm 5 Ej4

```
procedure EJ4(vector tiraNum, int n)
    vector[n+1][n+1] matriz ← Matriz con 0 en todos sus elementos
    Ej4Aux(tiraNum, n, matriz)                    ▷ Lleno la matriz
    int max ← BuscarMax(matriz, n + 1)            ▷ Tomo el máximo de la matriz
    int res ← n − max                             ▷ El resultado es números - máximo
    return res
```

Algorithm 6 Ej4Aux

```
procedure EJ4AUX(vector tiraNum, int n, matrix matriz)
    for fila in [0, 1, .., n + 1) do
        for column in [0, 1, .., n + 1) do
            if fila = column then
                matriz[fila][column] = 0          ▷ Si el último rojo y el último azul están en la misma posición
            if fila > column then                 ▷ Si tengo que recorrer la columna
                matriz[fila][column] = MaxAntQuePuedeRojo(tiraNum, matriz, 0, fila, column) + 1
            if fila < column then                 ▷ Si tengo que recorrer la fila
                matriz[fila][column] = MaxAntQuePuedeAzul(tiraNum, matriz, 0, column, fila) + 1
```

Algorithm 7 MaxAntQuePuedeAzul

```
procedure MaxAntQuePuedeAzul(vector tiraNum, matrix matriz, int i, int j, int fila)
    int res ← matriz[fila][i]                ▷ Fija el de la columna 0 como resultado
    for k in [i + 1, i + 2, .., j) do        ▷ Recorre la columna para buscar el máximo (también se
                                               debe poder pintar)
        if tiraNum[j − 1] < tiraNum[k − 1] ∧ matriz[fila][k] > res then
            res ← matriz[fila][k]
    return res
```

Donde $MaxAntQuePuedeRojo$ es un algoritmo muy similar a $MaxAntQuePuedeAzul$ solo que se recorre la fila para buscar el máximo, y además en este caso tengo que considerar que la secuencia roja es creciente y no decreciente como la azul; y $BuscarMax$ devuelve el máximo de la matriz iterandola por filas y columnas normalmente.

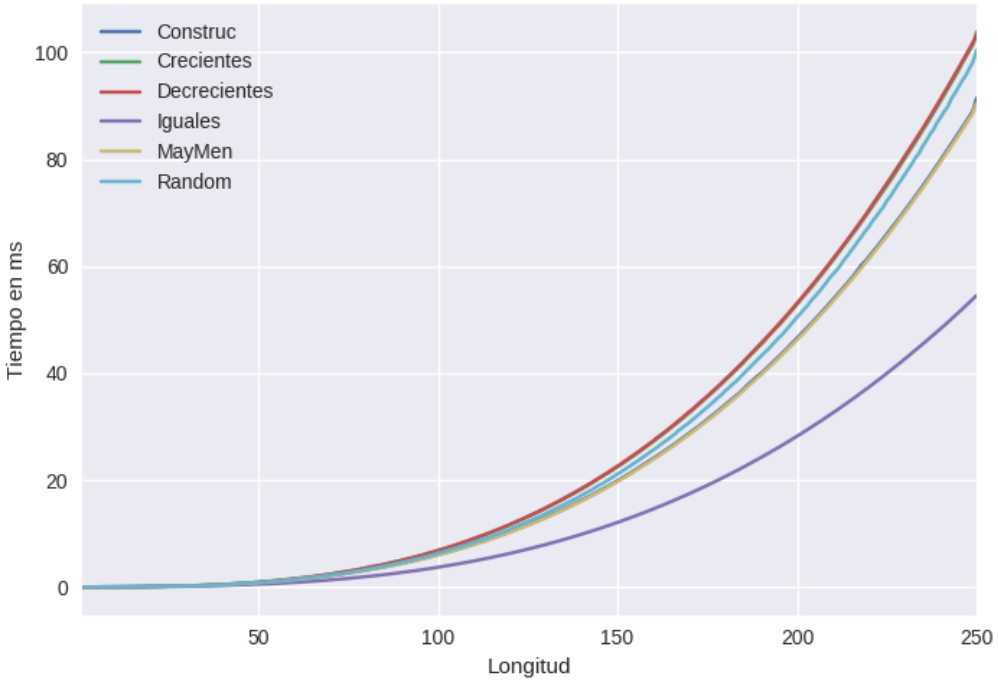
4.2 Complejidad teórica

El algoritmo realiza a lo sumo tantas llamadas recursivas como posiciones tiene mi matriz ($(n+1)*(n+1) = n^2+2n+1 = O(n^2)$) y en cada iteración recorre todos los anteriores de la fila/columna correspondiente, lo que tiene complejidad lineal.
Luego, la complejidad de llenar la matriz quedaría ($O(n^2) * O(n) = O(n^3)$).

Teniendo la matriz llena, encontrar el máximo me toma $O(n^2)$, por lo que esta operación no afecta la complejidad teórica final que termina siendo $O(n^3)$.

4.3 Experimentación

Se realizaron exáctamente los mismos experimentos que para el Ejercicio 3, incluso con las mismas entradas:



Analizando el gráfico queda claro que las entradas crecientes y decrecientes son las que tardan más tiempo, por lo que llego a la conclusión que estas entradas son las que generan el **peor caso** de mi algoritmo dadas las instancias con las que decidí realizar los experimentos.
Por otro lado, también se ve que de mis instancias generadas, el **mejor caso** de la función está cuando todos los elementos de la secuencia de entrada son iguales.

Finalmente, comparando este gráfico con el del Ejercicio 3 veo que los tiempos de cómputo con este algoritmo son bastante menores y por ende el mismo es más eficiente (sumado a que también utiliza menos memoria adicional).