



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico 1

16 de abril de 2017

Algoritmos y Estructuras de Datos III  
Primer Cuatrimestre de 2017

Autor	LU	Correo electrónico
Tarrío, Ignacio	363/15	itarrio@dc.uba.ar



**Facultad de Ciencias Exactas y Naturales**  
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

<http://www.exactas.uba.ar>

## Índice

1. Ejercicio 1 (BackTracking):	2
2. Ejercicio 2 (BackTracking con Poda):	4
3. Ejercicio 3 (Programacion dinamica):	7
4. Ejercicio 4 (otro de Programacion dinamica):	10

## 1. Ejercicio 1 (BackTracking):

En este ejercicio se nos pide resolver el problema utilizando un algoritmo de backtracking.

Backtracking es una técnica para buscar exhaustivamente todas las configuraciones del espacio de soluciones de un problema, para eso va generando las posibles soluciones constructivamente y va dejando atrás los candidatos cuando sabemos que no son válidos (que no pertenecen al espacio de soluciones válidas del problema).

Para este problema queremos buscar el mínimo de números sin pintar para una secuencia de números de largo  $n$ . Ya que un número puede estar pintado de rojo, azul o sin pintar, y como tenemos  $n$  números, podemos tener como máximo  $3^n$  posibles combinaciones, aunque podrían ser menos ya que de esas combinaciones hay posibilidades que no son válidas como soluciones ya que puede ser que la subsecuencia de rojos no sea creciente o decreciente para la de azul. Una vez generadas todas estas soluciones, deberíamos encontrar la que menos números tiene sin pintar y esa va a ser la solución a nuestro problema (se nos pide la cantidad no qué combinación de rojos y azules).

Ya que estamos construyendo las distintas soluciones, empezamos decidiendo de qué color pintar el número primer número, rojo azul o sin pintar, para cada una de estas 3 decisiones vamos a pasar al siguiente número y repetir el proceso, así hasta llegar al final en el que vamos a tener como pintamos los números y nos fijamos cuantos dejamos sin pintar. Una vez que tenemos todas las soluciones buscamos la óptima.

Se puede implementar este algoritmo de manera recursiva, dado el arreglo de números, el índice, y cuál fue el último rojo y último azul pintado (si hay), devuelve la mínima cantidad de números sin pintar dentro de las posibilidades. Pedimos el último rojo y el último azul porque no me importa saber cómo los pinte si no los últimos ya que eso me va a dar la limitante de si el próximo número lo puedo pintar de rojo o azul (y descartamos los que no pude pintar). Entonces esta función recursiva guarda los resultados del índice siguiente para las 3 posibilidades (rojo, azul y sin pintar), se fija cuál es la menor y devuelve esa (si es la que no se pinta tiene que sumarle uno), excepto que sea el caso basó en el que no hay índice siguiente y devolvemos 0 si podemos pintarlo (0 elementos sin pintar en un arreglo que contiene sólo al último elemento pintado de cualquier color) o 1 si no lo pudimos pintar, y después se van a ir llenando las anteriores llamadas con estos casos bases.

### Pseudocódigo

ej (int arreglo[], n) → res: int
<b>Data:</b> arreglo = el arreglo de números entero n = el tamaño del arreglo <b>Result:</b> La cantidad mínima de números sin pintar para una secuencia de números de largo n
<pre>/* Empezamos el backtracking desde el primer índice(0) y no inicializamos todavía el último rojo y el último azul */ res ← BT(0, arreglo, n, NULL, NULL)</pre>

BT (int i, arreglo[], n, ultimoRojo, ultimoAzul) → res: int

```

Data: i = índice actual a pintar
arreglo = el arreglo de numeros entero
n = el tamaño del arreglo
ultimoRojo = el ultimo numero que se pinto de rojo(NULL si no se pinto ninguno)
Result: La cantidad minima de numeros sin pintar a partir de i hasta n
numeroActual ← arreglo[i]
if i = n - 1 then
    /* Si el indice es el ultimo numero entonces estamos en el caso base */
    /* Tenemos que fijarnos si podemos pintarlo */
    if ultimoRojo = NULL or ultimoAzul = NULL or ultimoRojo < numeroActual or ultimoAzul >
        numeroActual then
            res ← 0
        else
            /* No lo podemos pintar porque no seria una solucion valida entonces devolvemos 1 */
            res ← 1
        end if
else
    int minSiRojo, minSiAzul, minSinPintar
    if ultimoRojo = NULL or ultimoRojo < numeroActual then
        /* Si lo podemos pintar el numero actual de rojo osea si es mas grande que el anterior
           rojo pintado, o si es el primero rojo en pintarse, y cambiamos el ultimoRojo por
           este numero */
        minSiRojo ← BT(i+1, arreglo, n, numeroActual, ultimoAzul)
    end if
    if ultimoAzul = NULL or numeroActual < ultimoAzul then
        /* Lo mismo con el azul */
        minSiAzul ← BT(i+1, arreglo, n, ultimoRojo, numeroActual)
    end if
    /* Calculamos tambien si no lo pintamos y al resultado le sumamos uno porque este no lo
       pintamos */
    minSinPintar ← BT(i+1, arreglo, n, ultimoRojo, ultimoAzul) + 1
    /* retornamos el minimo de ambos(considerando que la func min es O(1) y si es nulo la
       variable no lo considera) */
    res ← min(minSiRojo, minSiAzul, minSinPintar)
end if

```

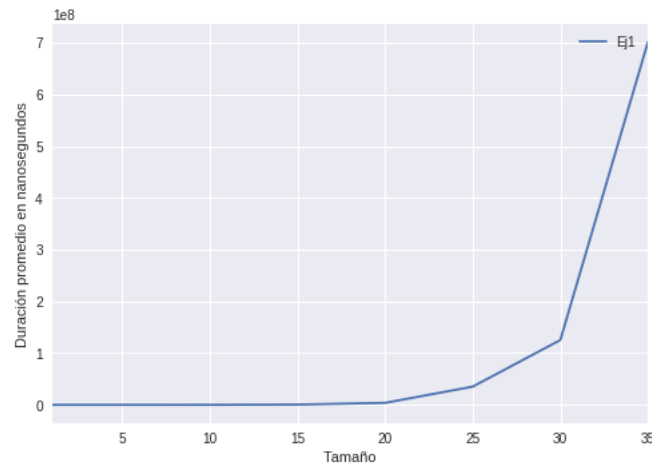
## Análisis de Complejidad

Ya dijimos que la cantidad de combinaciones posibles son  $3^n$ , y podemos decir que este algoritmo recorre como máximo todas ellas porque empieza desde el índice 0 y por cada número del arreglo prueba las 3 combinaciones (nada más las válidas), entrando recursivamente al índice siguiente (i+1) y esos lo van a repetir hasta llegar al caso base que sería llegar al último índice. Ahora, cada llamada recursiva hace una cantidad de operaciones  $O(1)$  constantes (sumar, comparar, mínimo) y si no es el caso base además hace 3 llamadas recursivas en peor caso, pero reduciendo el n. Esto nos hace quedar que la complejidad de la función recursiva es  $T(n) = 3T(n-1) + O(1)$  con  $T(0) = 1$  como el caso base, que se puede demostrar fácilmente (por inducción) que es  $O(3^n)$  para el peor caso. Otra forma de llegar es ver el árbol de ejecución de la recursión, cada nodo desprende tres nodos hijos, vamos a tener un árbol de altura n, y como es un árbol ternario tenemos  $3^n$  hojas, que son nuestras soluciones. Esta complejidad se considera exponencial y es muy mala para instancias grandes. Obviamente esto no pasa siempre, porque no siempre es posible entrar calcular las 3 posibilidades, de hecho es muy probable que no se pueda, supongamos que ya pintamos un numero de rojo y un número azul, para que entre a las tres bifurcaciones el número este tiene que ser mayor al rojo y menor al azul, osea que el azul tiene que ser mayor que el rojo y además el numero actual tiene que estar en el medio, ahora cuando sigamos con el próximo número esta brecha se va a haber achicado entonces va a ser menos probable que el número caiga en el medio de ambos.

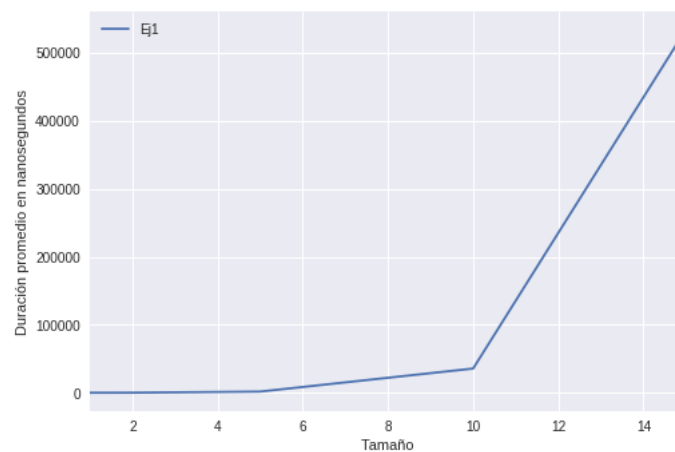
## Experimentación Computacional

### Instancias Aleatorias

Se corrió una experimentación, sobre el código realizado en c++, generando arreglos aleatorios de tamaño  $n$ , y los números tienen una distribución uniforme donde el mínimo es 0 y el máximo es 2 veces  $n$ , esto se hizo así, para que exista una buena probabilidad de que haya números repetidos, ya que si por ejemplo hay un número repetido 3 veces no existe posibilidad de que el óptimo sea 0, así agregando casos malos a la experimentación. Se probaron arreglos de tamaño: 1, 2, 5, 10, 15, 20, 25, 30, 35. De cada uno de estos tamaños se computaron 30 arreglos diferentes, de estos 30 se tomo el promedio.



Como podemos ver en el gráfico lo que tarda en resolver el problema aumenta exponencialmente a medida que crece el tamaño del arreglo, de hecho en promedio para calcular el arreglo de tamaño 35 es 0,701732075 segundos que es mucho, y para el de 30 tardó 0,125287312 segundos casi 5 veces más. Para seguir haciendo énfasis en que cada vez tarda más hagamos zoom para los primeros datos.



En este gráfico también podemos apreciar como va creciendo exponencialmente (en el otro gráfico parecía que al principio era constante la duración). En este caso tenemos que en promedio para los arreglos de tamaño 5 tardan 0,001741 milisegundos y para los de tamaño 10 tarda 0,035622 milisegundos casi 20 veces más.

Conclusión si bien es viable para instancias chicas, es muy ineficiente y tarda mucho cuando las instancias empiezan a ser más grande.

## 2. Ejercicio 2 (BackTracking con Poda):

En este ejercicio nos piden hacer una mejora en el anterior algoritmo de backtracking y hacerle una poda. Las podas consisten en que hay instancias de soluciones parciales que ya no nos interesan y ni siquiera las necesitamos calcular,

porque por ejemplo esta solución va a hacer peor que una que ya calculamos anteriormente. Esto lo que provoca es una optimización porque dejamos de calcular cosas, aunque sigue siendo una búsqueda exhaustiva, si vemos las el árbol de recursión lo que pasa es que cortamos muchas ramas.

Si recordamos el anterior algoritmo, para un índice calculaba el menor si lo pintamos de rojo, despues de azul y después sin pintar, pero que pasa si uno nos devuelve que se pueden pintar todos los proximos numeros, osea que los los otros no hace falta computarlos ya que solo nos importa la cantidad de números sin pintar, y no vamos a encontrar una mejor porque esta es la mínima. Esta es nuestra primer poda, una vez calculado una solución parcial para un color, nos fijamos si es óptima y si lo es nos salteamos(no computamos) las otras alternativas de colores.

Tambien podriamos llevar una cuenta de cuál es la solución(entera) óptima y no calculamos una que pueda ser peor que esa, por ejemplo si voy por el 3 índice y encontramos una solución que solo no puede pintar un numero, solo tenemos que buscar una solución que pinta todos los numeros, asi que no vamos a entrar a las recursiones de no pintar.

Para la implementación de estas podas modificamos un poco la base del código del anterior ejercicio así se nos es más fácil calcular los datos. Ahora la función recursiva tiene 2 parámetros de entrada nuevos, el mínimo encontrado en las soluciones ya calculadas(lo pasamos por referencia y editamos ese) y la cantidad de números que ya no pintamos y ahora en vez de devolver un resultado parcial, devuelve la cantidad de números sin pintar para todo el arreglo(ya que tenemos la cantidad que no pintamos antes).

Podemos afirmar que el algoritmo sigue siendo correcto porque las únicas subsoluciones que descartamos son las que ya sabemos que no van a ser las óptimas, esta es la gracia de la poda en el backtracking.

## Pseudocodigo

ej2 (int arreglo[], n) → res: int

**Data:** arreglo = el arreglo de numeros entero

n = el tamaño del arreglo **Result:** La cantidad minima de numeros sin pintar para una secuencia de numeros de largo n

/\* Empezamos el backtracking desde el primer indice(0) y no inicializamos todavia el ultimo rojo y el ultimo azul, y ademas por defecto el optimo es n(no existe peor solucion que esta) y no pintamos ninguno todavia \*/  
res ← BT(0, arreglo, n, NULL, NULL, n, 0)

BTI (int i, arreglo[], n, ultimoRojo, ultimoAzul, optimoActual, sinPintar) → res: int

**Data:** i = índice actual a pintar

arreglo = el arreglo de numeros entero

n = el tamaño del arreglo

ultimoRojo = el ultimo numero que se pinto de rojo(NULL si no se pinto ninguno)

ultimoAzul = el ultimo azul pintado

optimoActual = la mejor solucion encontrada hasta el momento sinPintar = cantidad de numeros sin pintar antes de i

**Result:** La cantidad minima de numeros sin pintar a partir de i hasta n

numeroActual ← arreglo[i]

**if**  $i = n - 1$  **then**

    /\* Si el indice es el ultimo numero entonces estamos en el caso base \*/

    /\* Tenemos que fijarnos si podemos pintarlo \*/

**if**  $ultimoRojo = NULL$  **or**  $ultimoAzul = NULL$  **or**  $ultimoRojo < numeroActual$  **or**  $ultimoAzul > numeroActual$  **then**

        res ← sinPintar

**else**

        /\* No lo podemos pintar porque no seria una solucion valida entonces devolvemos la cantidad que pintamos antes mas 1 porque este no lo pintamos \*/

        res ← sinPintar + 1

**end if**

**else**

    int minSiRojo, minSiAzul, minSinPintar

**if**  $ultimoRojo = NULL$  **or**  $ultimoRojo < numeroActual$  **then**

        minSiRojo ← BTI(i+1, arreglo, n, numeroActual, ultimoAzul, optimoActual, sinPintar)

        /\* PODA 1: si es el optimo no calculamos los demas y devolvemos este \*/

**if**  $minSiRojo = sinPintar$  **then**

            res ← minSiRojo

**end if**

**if**  $minSiRojo < sinPintar$  **then**

            optimoActual ← minSiRojo

**end if**

**end if**

**if**  $ultimoAzul = NULL$  **or**  $numeroActual < ultimoAzul$  **then**

        /\* Lo mismo con el azul \*/

        minSiAzul ← BTI(i+1, arreglo, n, ultimoRojo, numeroActual, optimoActual, sinPintar)

        /\* PODA 1 \*/

**if**  $minSiAzul = sinPintar$  **then**

            res ← minSiAzul

**end if**

**if**  $minSiAzul < sinPintar$  **then**

            optimoActual ← minSiAzul

**end if**

**end if**

    /\* PODA 2: solo calculamos sin pintar este numero, si todavia podemos mejorar el optimo \*/

**if**  $sinPintar < optimoActual - 1$  **then**

        minSinPintar ← BTI(i+1, arreglo, n, ultimoRojo, ultimoAzul, optimoActual, sinPintar + 1)

**end if**

    res ← min(minSiRojo, minSiAzul, minSinPintar)

**end if**

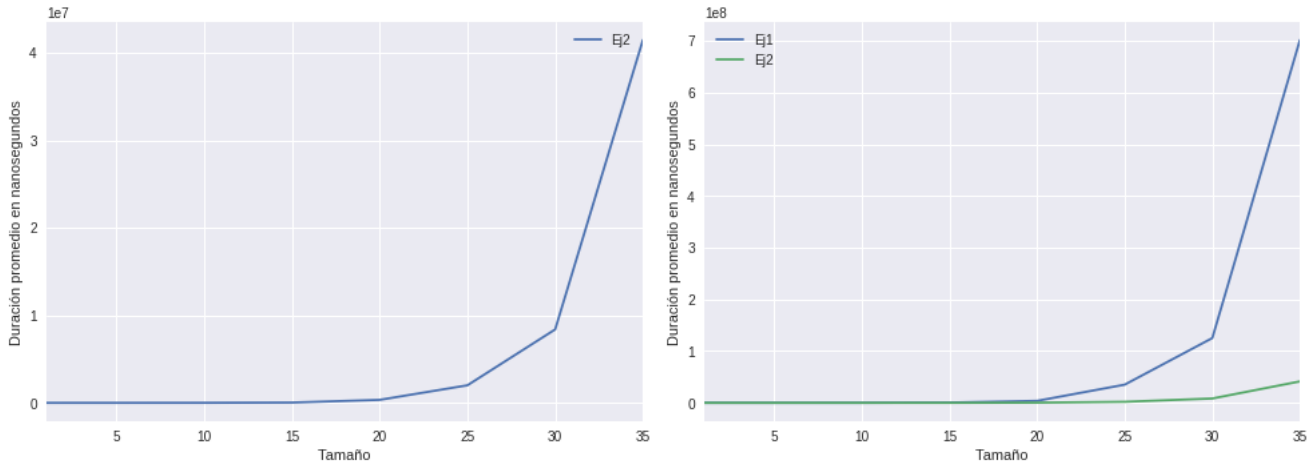
## Análisis de Complejidad

La cantidad de recursiones a la que se sigue pudiendo entrar sigue siendo de 3, osea que sigue siendo  $O(3^n)$ , si bien va a disminuir demasiado las recursiones que hacemos(ya que con estas podas cortamos muchos caminos), esto va a mejorar muchos el promedio de los casos, en los que casi siempre vamos a cortar muchas ramas. El mejor caso ahora es lineal, si nos dan una secuencia creciente, la primer solución que se encuentra es la que pintamos todos de rojo y como es 0 ya devolvemos eso, esto es  $O(n)$

## Experimentación Computacional

### Instancias Aleatorias

Se corrieron la experimentacion sobre la misma computadora y para los mismos arreglos que para el anterior ejercicio



A simple vista podemos ver que sigue creciendo exponencialmente, pero al compararlo con el algoritmo de sin poda notamos una gran mejora. Para instancias de 35 el original tarda 0,701732075 segundos pero el nuevo tarda 0,041461628 segundos 16 veces menos. Estos tiempos ya son mas viables que el anterior ejercicio.

### 3. Ejercicio 3 (Programacion dinamica):

La programación dinámica es otra técnica algorítmica, consiste en resolver los subproblemas de tamaños menores y dados estos resultados, generar la solución al problema original, pero la técnica aprovecha estructuras de datos(generalmente matrices) para guardar los subproblemas que ya calculamos, y si en algún momento lo tenemos que calcular de nuevo(por ejemplo si dos subproblemas del mismo tamaño necesitan del mismo subproblema de tamaño más chico) nos ahorramos el cálculo porque tenemos en la estructura la respuesta.

Principio de optimalidad de Bellman: Para que el problema satisfaga el principio una subsolución óptima debe ser solución del subproblema asociado a esa subsolucion. En este problema podríamos decir que para una secuencia de largo en un subproblema sea la subsecuencia de 0 a  $n - 1$ , y si tenemos esta su subsolución, entonces, la solución al problema original va a ser menor o igual a esta subsolución, de hecho va a ser igual o la misma menos uno(siempre y cuando no sea 0). Probemos esto que acabamos de decir, supongamos que tenemos una subsolucion que vale  $x \geq 2$  para el subproblema de tamaño igual menos uno a la del problema que tenemos que resolver, y también supongamos que la solución al problema es un  $x'$  tal que  $x' = x - 2$ , si pasa esto podríamos quedarnos con esa combinación de colores(con la que nos da  $x'$ ) sacarle el último número y tendríamos una nueva subsolucion  $x''$  tal que  $x'' = x' - 1$  ya que solamente sacamos el último número, pero entonces  $x'' < x$  osea que  $x$  no era óptima, por ende entramos en un absurdo. Lo absurdo sale de suponer que había una solución mucho mejor que la de la solución. Osea que de una sub solucion no podemos empeorar la solucion o solo la empeoramos en una unidad, la dejamos igual cuando el nuevo número lo podemos pintar desde alguna de la combinaciones óptimas anteriores, o empeora en uno si no existe combinación óptima anterior desde la cual no podamos pintar este numero de rojo o de azul(notar que ahora hay combinaciones que por ahí no son óptimas para el subproblema anterior pero se diferencian en uno). Gracias a esto cumple el principio de optimalidad de Bellman, pero notemos que si el problema fuese dar las subsecuencias rojas y azules, no se cumpliría el principio ya que, por lo último que dijimos, cuando no hay una combinación óptima anterior puede ser que las nuevas soluciones no están asociadas a las del subproblema, entonces no cumpliría el principio.

Como vimos en backtracking al pintar un numero, solo nos importa cuales fueron el último rojo y el último azul que se pintaron, entonces al agregar un nuevo número a la secuencia, tenemos que buscar la combinación del último rojo y último azul óptima tal que nos deje pintar este último número o buscar una combinación de rojo y azul la cual no pintemos este último pero sea mejor que si lo pintamos. Entonces por problema tenemos que buscar cual es la combinación de último rojo y último azul que nos haga óptima la solución.

El algoritmo dado se va a basar en eso, va a terminar calculando todas las combinaciones de último rojo y último azul para ver cual es la óptima. Como calculamos cada una de estas combinaciones? recientemente dijimos que para pintar un número de forma óptima teníamos que buscar en la combinación de ultimo rojo ultimo azul valida que sea óptima. Entonces si queremos pintar el último índice de rojo, sea  $r$ , y como último azul un índice  $a$  con  $0 \leq a < r$ , deberíamos buscar el  $r'$  tal que sea óptimo y qué  $secuencia_{r'} < secuencia_r$ , para esto vamos a tener que recorrer de 0 a  $r - 1$  buscando el que cumpla estas condiciones. Otro tema importante de la programación dinámica es como



guardamos los datos, una manera es tener una matriz de 2 dimensiones, en el que el índice de la fila representa el índice del último azul y el índice de la columna el del rojo, osea que sea  $M$  la matriz,  $M_{r,b}$  nos da cual la sub solución para esa combinación de rojo y azul. Un caso que deje afuera pero hay que tener en cuenta es que hay una combinación que es en la que no se pinta ninguno de rojo o de azul, o no se pinta ninguno, este índice se va a representar en la última fila y en la última columna, resumiendo para un problema de  $n$  números vamos a tener una matriz  $M$  de  $(n+1) \times (n+1)$  donde la fila  $n+1$  y la columna  $n+1$  representan cuando el último azul o ultimo rojo respectivamente son nulos.

Este es nuestro algoritmo, calculamos la matriz y buscamos en ella cuál es el óptimo, ahora tenemos que decidir en qué orden calculamos los datos, ya que como para calcular  $M_{r,b}$  tenemos que tener calculado  $M_{i,b}$  con  $i < r$  y  $M_{r+1,b}$  va a necesitar de estos también, podemos ir de una manera constructiva, empezando en los casos bases, que son los cuando pintamos nada más el primero de un color (que sabemos que va a dejar a todos los otros números siguientes sin pintar entonces la subsolución es  $n-1$ ) y si no pintamos ninguno (osea quedan  $n$  sin pintar), esto era el índice 0, después pasamos para el índice 1 y calculamos todas las combinaciones nuevas, osea ya teníamos calculado  $M_{0,n+1}$  y  $M_{n+1,0}$  y  $M_{n+1,n+1}$ , y ahora calculamos  $M_{1,0}$ ,  $M_{1,n+1}$ ,  $M_{0,1}$ ,  $M_{n+1,1}$  (notar que  $M_{r,b}$  con  $r=b$  es inválido porque no podemos pintar un mismo número de dos colores). Una vez calculada toda la matriz buscamos el óptimo dentro de esta y esa es nuestra solución.

ej3 (int arreglo[], n) → res: int

**Data:** arreglo = el arreglo de numeros entero

n = el tamaño del arreglo

**Result:** La cantidad minima de numeros sin pintar para una secuencia de numeros de largo n

/\* Creamos nuestra matriz \*/

int m[n+1][n+1]

/\* Le guardamos los triviales, los del primer indice(0) \*/

int m[0][n] = n - 1

int m[n][0] = n - 1

int m[n][n] = n

/\* Ahora recorremos todos los indices \*/

for i ← 1 to n do

/\* Y para cada indice, calculamos el optimo si pintaramos este de azul o de rojo y las combinaciones del otro \*/

for j ← 0 to i - 1 do

m[i][j] ← buscarOptimoPintandoDeRojo(i, j, arreglo, m, n)

m[j][i] ← buscarOptimoPintandoDeAzul(i, j, arreglo, m, n)

end for

/\* En esa iteracion dejamos de lado la combinacion si pintamos este ultimo de azul y no pintamos ninguno de rojo, y viceversa \*/

m[i][n] ← buscarOptimoPintandoDeRojo(i, n, arreglo, m, n)

m[n][i] ← buscarOptimoPintandoDeAzul(i, n, arreglo, m, n)

end for

/\* Una vez ya calculada la matriz buscamos el menor, min es una funcion que busca el menor de una matriz en  $O(m)$  donde m es la cantidad de elementos que hay en la matriz \*/

res ← min(m)

```
buscarOptimoPintandoDeRojo (int r, a, arreglo[], m, n) → res: int
```

```
Data: r = El indice que estamos pintando de rojo
a = El indice que fijamos de azul
arreglo = La secuencia de numeros
m = La matriz(pasa por referencia)
n = el tamaño del arreglo
Result: La cantidad minima de numeros sin pintar si elegimos como ultimo pintado de rojo al indice r y de azul
a a
/* Ponemos como optimo momentaneo al  $M_{n,a}$  osea si el primer rojo que pintamos sea este
numero, total siempre es una solucion valida */
int optimoActual ← m[n][a]
/* Recorremos con el azul fijado, que pasaria si partieramos desde los anteriores rojos */
for ri ← 0 to r - 1 do
    /* Nos fijamos si el numero de la secuencia de ri es menor al de r, (osea si podemos
    pintarlo a partir de ri), y si es mas chico que el optimoActual */
    if ri != a and arreglo[ri] < arreglo[r] and m[ri][a] < optimoActual then
        /* Si es menor, marcamos a este como el optimoActual */
        optimoActual ← m[ri][a]
    end if
end for
/* Retornamos el optimo del cual podemos pintar y como pintamos este ultimo se reduce en 1
*/
res ← optimoActual - 1
```

```
buscarOptimoPintandoDeAzul (int a, j, arreglo[], m, n) → res: int
```

```
Data: a = El indice que estamos pintando de azul
r = El indice que fijamos de rojo
/* El codigo de esta funcion es totalmente analogo al de rojo */
int optimoActual ← m[r][n]
for ai ← 0 to a - 1 do
    if ai != r and arreglo[ai] > arreglo[a] and m[r][ai] < optimoActual then
        | optimoActual ← m[r][ai]
    end if
end for
res ← optimoActual - 1
```

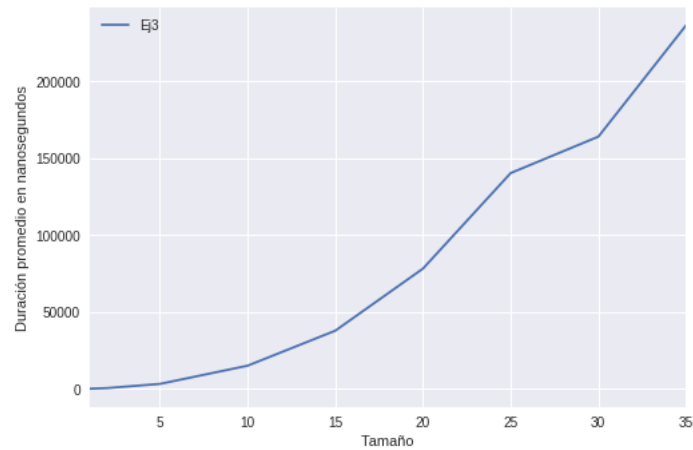
## Analisis de Complejidad

Vamos a calcular en todos los casos, toda la matriz que ambas dimensiones son de tamaños  $n + 1$ , osea que ya recorrer todos los elementos nos cuesta  $O(n^2)$  y para calcular cada una de estos elementos de la matriz recorremos  $O(n)$  elementos(en la función auxiliar para buscar el óptimo), eso ya nos da una complejidad de  $O(n^3)$  y después buscar el óptimo dentro de la matriz también es  $O(n^2)$  y después son una cantidad constante de operaciones elementales, en definitiva nos queda una complejidad para el peor de los casos de  $O(n^3)$ , de hecho en el código dado no cortamos nunca y siempre completamos la matriz entonces para cualquier caso el código va a hacer  $O(n^3)$ . Una alternativa de análisis es considerar los casos que tenemos, podríamos decir que tenemos 3 variables en cada caso, el último azul, el ultimo rojo, y el índice, entonces como cada una de estas variables está acotada por  $n$  tenemos  $n^3$  casos y son los que terminamos calculando.

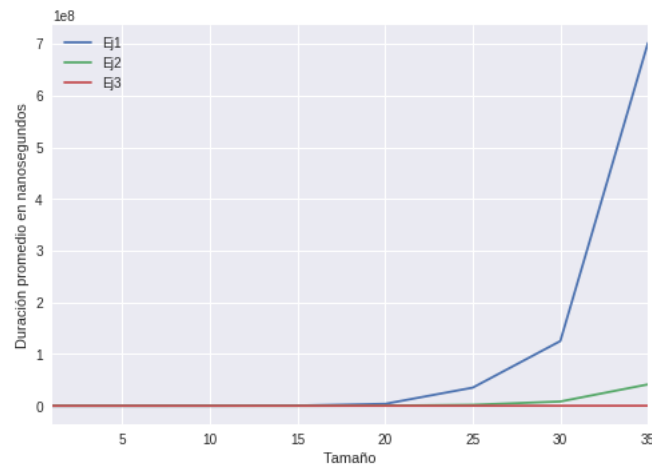
## Experimentación Computacional

### Instancias Aleatorias

De vuelta usamos las mismas instancias que antes, generadas para el ejercicio 1.



Ya a simple vista podemos ver que no crece exponencialmente si no mas lineal, compramoslo con los otros ejercicios



Obviamente no es que se mantiene constante, es que es mucho menor y a esa escala no se puede apreciar que va aumentando. El segundo ejercicio tarda 0,041461628 segundos y este tarda 0,236551 milisegundos la diferencia es abismal. Aquí podemos notar la diferencia entre un algoritmo exponencial vs uno polinomial donde para instancias grande el exponencial ya es muy grande.

#### 4. Ejercicio 4 (otro de Programacion dinamica):

En este ejercicio nos piden resolver el problema usando otro algoritmo de programacion dinamica. En nuestro caso lo vamos a realizar en base al anterior ejercicio pero vamos a hacerlo recursivo.

Habiamos visto que teniamos que buscar cual era la mejor combinacion de ultimo rojo y ultimo azul, para el anterior ejercicio estabamos calculando la matriz constructivamente, lo haciamos de una forma que cuando necesitabamos un subproblema ya sabiamos que lo teniamos solucionado. Ahora lo que vamos a hacer es, no nos importa el orden en el que lo calculamos, pero cuando lo pido y no lo tengo, recursivamente me pongo a calcularlo, obviamente la matriz la mantenemos, asi no calculamos algo 2 veces. Entonces lo unico que tenemos que hacer es preguntar cuando vale cada el

emento de la matriz en cualquier orden y si los subproblemas necesarios no estan calculados se calculan recursivamente.

ej3 (int arreglo[], n) → res: int	
<b>Data:</b> arreglo = el arreglo de numeros entero n = el tamaño del arreglo <b>Result:</b> La cantidad minima de numeros sin pintar para una secuencia de numeros de largo n /* Creamos nuestra matriz y esta toda inicializada en -1 int m[n+1][n+1] /* Le guardamos los triviales, los del primer indice(0) int m[0][n] ← n - 1 int m[n][0] ← n - 1 int m[n][n] ← n int optimo ← n /* Recorremos todos los indices buscando el optimo for i ← 0 to n do for j ← 0 to n do if (i != j — i == n) and optimo(i,j) > optimo then   optimo ← optimo(i,j) end if end for end for res ← optimo	*/ */ */ */

optimo (int r, a) → res: int	
<b>Data:</b> Suponemos que tiene acceso al arreglo a la matriz y al largo r = El indice que estamos pintando de rojo a = El indice que fijamos de azul <b>Result:</b> El optimo para m[r][a] /* Nos fijamos si esta calculado if m[r][a] = -1 then /* Si no esta calculado hay que calcularlo /* Ahora nos tenemos revisamos cual fijar if (ultimoAzul < ultimoRojo and ultimoRojo != largo) or ultimoAzul = largo then   m[ultimoRojo][ultimoAzul] ← buscarOptimoRojo(ultimoRojo, ultimoAzul); else   m[ultimoRojo][ultimoAzul] ← buscarOptimoAzul(ultimoAzul, ultimoRojo); end if end if /* Devolvemos el numero calculado res ← m[r][a]	*/ */ */ */ */

buscarOptimoPintandoDeRojo (int r, a) → res: int	
/* Como defecto ponemos el largo del arreglo int optimoActual ← largo /* Recorremos con el azul fijado, que pasaria si partieramos desde los anteriores rojos for ri ← 0 to r - 1 do /* Nos fijamos si el numero de la secuencia de ri es menor al de r, (osea si podemos pintarlo a partir de ri), y si es mas chico que el optimoActual if ri != a and arreglo[ri] < arreglo[r] and m[ri][a] < optimoActual then /* Si es menor, marcamos a este como el optimoActual optimoActual ← m[ri][a] end if end for /* Retornamos el optimo del cual podemos pintar y como pintamos este ultimo se reduce en 1 res ← optimoActual - 1	*/ */ */ */ */

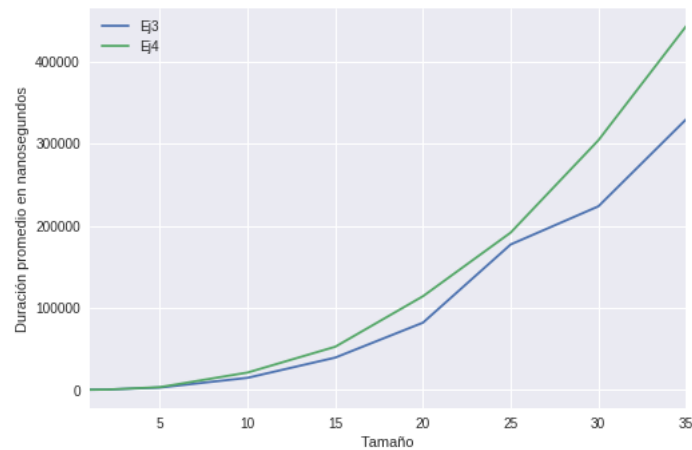
## Analisis de Complejidad

Basicamente es la misma que en la de del ejercicio 3. Recorremos todos los numeros de la matriz y solos lo vamos a calcular una vez, y en el peor de los casos calcular este numero es  $O(n)$ . Podemos notar que dependiendo de que se pregunta primero, vamos a tener algunos datos primeros que otro. Por ejemplo si empezamos por los ultimos indices se va a calcular toda la matriz. Y despues cuando recorramos toda vamos a tener todos los datos en  $O(1)$

## Experimentación Computacional

### Instancias Aleatorias

De vuelta usamos las mismas instancias que antes, generadas para el ejercicio 1.



Podemos notar que son muy similares(hacen casi los mismos calculos), pero podemos notar que este nueva implementacion, esto se puede deber a que ya que es recursiva gastamos memoria teniendo las llamadas, o a algun tema con la implementacion que lo empeoro.