DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Recuperatorio Trabajo Práctico I

Pintando Secuencias

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Arroyo, Luis Alberto	913/13	luis.arroyo.90@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) 4576-3300
http://www.exactas.uba.ar

Contents

1 Introducción

En este informe, vamos a analizar resoluciones distintas para el problema presentado a continuación.

Problema: Dada una secuencia de numeros naturales, se pide pintar de color rojo a una secuencia estrictamente creciente y de color azul a una secuencia estrictamente decreciente de manera tal que se minimice la cantidad de elementos sin pintar.

Por ejemplo dada la secuencia:

1	2	2	1	1	2	3	3	2	1

Rojo:

1	2	3

Azul:

3	2	1

Resultado = 4

7	2	4	5	5	3	9

Rojo:

2	4	5	9

Azul:

7	5	3

Resultado = 0

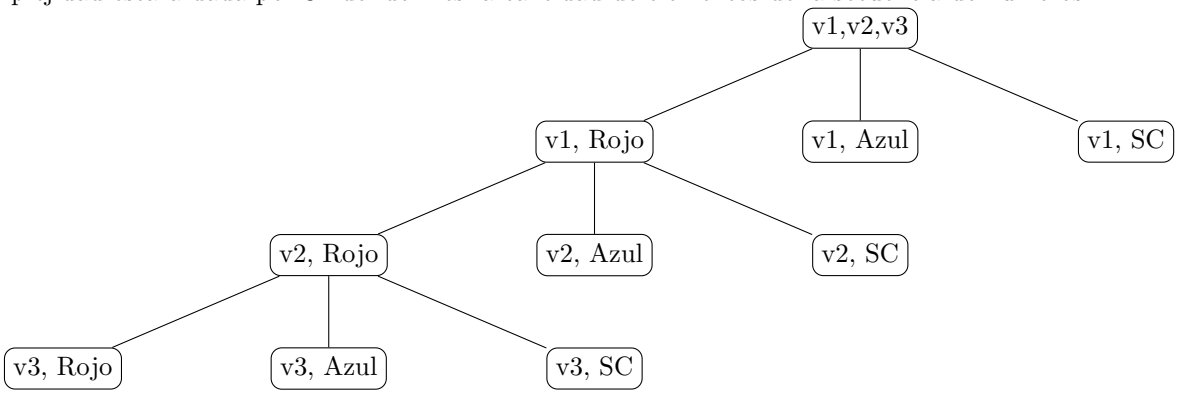
Para ello resolveremos este problema de tres formas distintas:

- Backtracking.
- Backtracking con poda.
- Dinámica.

2 Problema I: Bactracking

2.1 Introducción

Vamos a utilizar un algoritmo de backtraking que,es lo mas parecido a hacer fuerza bruta y recorrer todas las posibilidades de pintar cada elemento. Es decir, cada elemento tiene tres formas diferentes de colorearse: Rojo, Azul o Sin Color. de esta forma las combinaciones para un elemento esta determinado por 3 valores, luego para dos elemento las combinaciones son 3^2 para k elementos 3^k por lo tanto la complejidad estará dada por 3^n donde n es la cantidad de elementos de la secuencia de números



La solución está en las hojas del árbol, por lo tanto habrá que recorrer todo el árbol en peor caso para encontrar la mejor solución.

2.2 Resolución del problema y representación

Para resolver este problema con backtracing, se iterará en toda la lista subsecuencias de tamaño 1 hasta el tamaño total de la secuencia buscando la combinación máxima de elementos rojos y azules pintados. Para realizarlo presentaré el siguiente algoritmo.

Dicho algoritmo tiene como representación dos vectores uno de enteros no negativos llamado elementos, otro de booleanos llamado validar y dos pilas una roja y otra azul con las siguientes características:

- Los elementos de las pilas son las posiciones de los elementos de las secuencias.
- Una pila es estrictamente creciente y la otra decreciente.
- Las pilas son disyuntas entre si.
- Todos los elementos de las pilas están en elementos.
- Cuando un elemento ingresa a una pila, se asigna false a la posición de validar del elemento que ingreso.
- el vector validar me permite saber que elemento puedo agregar a la pila.

2.3 Algoritmo

El algoritmo consta de las siguientes funciones:

2.3.1 Triviales

- Establecer Validos:
  toma una pila y el vector validar, y establece el invariante para que cuando la otra función sea llamada no pueda elegir los elementos que están en la pila máxima.
- Maximizar:
  toma el tamaño de las dos pilas y devuelve el tamaño máximo, por ultimo bastaría restar este valor al tamaño de entrada y se obtiene la cantidad de libres.

  Observación: Cada función tiene otra idéntica modificando las guardas para encontrar el caso inverso por ejemplo la secuencia creciente tiene una decreciente y mientras que rojos maximiza la secuencia creciente azules maximiza la decreciente ambas teniendo en cuenta de no pisar la otra.

  secuCreciente procesa todos los elementos de la lista desde dos posiciones cualesquiera del vector y agrega los que son validos o mayores.

Algorithm 1 Backtracking

```
1: procedure secuCreciente(secu(int): elementos,secu(bool): validar,pila(int): Pila,int inicio,int fin )
2:     iterador ← inicio
3:
4:     while (iterador <=fin) do
5:         if tamaño(pila) == 0 then
6:             pila.push(iterador)
7:             validar[iterador]← false
8:
9:         else
10:
11:            if elementos[iterador] > elementos[pila.top()] && validar[iterador] then
12:                pila.push(iterador)
13:                validar[iterador] ← false
14:            end if
15:            iterador ++
16:        end if
17:    end while
```

Complejidad: $\mathcal{O}(fin - inicio)$
Justificación: debido a que como máximo hace un recorrido lineal.

Algorithm 2 Backtracking

```
1: procedure subSecuRojaMaxima(secu(int) elementos,secu(bool): validar,pila (int): PilaMaxima,pila (int): Pila,int fin,int it,int n,bool notermine )
2:
3:
4:     while (¬pila.vacia∧ notermine ) do
5:
6:         while (it < fin ∧ notermine ) do
7:             SecuCreciente(elementos,validar,pila,it,fin)
8:
9:             if pilamaxima.size()> fin then
10:                pilaMaxima ← pila
11:            end if
12:
13:            if pila.size())== fin then
14:                notermine ← false
15:            else
16:                it ← pila.top()+1
17:                validar[pila.top()]= true
18:                pila.pop()
19:            end if
20:
21:            if pila.size() ! =0 then
22:                it ← pila.top()+1
23:                validar[pila.top()] ← True
24:                pila.pop()
25:            end if
26:        end while
27:    end while
```

Complejidad: $\mathcal{O}(c^c)$

Justificacion: Siendo p = pila.size(), sea c = p + inicio - fin. Supongamos que de entrada hay una tira de elementos crecientes menos el ultimo, la pila pasa a tener tamaño c Luego, cada vez que termina el ciclo quito el primer elemento y llamo a la función SecuCreciente que recorre los elementos hasta el final De manera tal que, en cada iteración resto un elemento a la pila pero recorro uno mas en secuCreciente haciendo que en todas las iteraciones recorra todos los elementos por cada iteración.

Algorithm 3 Backtracking

```
1: procedure ROJOS(secu(int): elementos,secu(bool): validar,pila(int): PilaMaxima,int inicio,int fin )
2:
3:     notermine ← true;
4:
5:     while (validar[j] == false ∧ j <= fin ) do
6:         j++
7:     end while
8:
9:     if j > fin then
10:        notermine ← false;
11:    end if
12:
13:    while j <= fin ) do
14:        pila.push(j)
15:        validar[j] ← false
16:        subSecuRojaMaxima(elementos,validar,pilaMaxima,pila,fin,it,notermine)
17:        j++
18:    end while
```

Complejidad: $\mathcal{O}(k * c^c)$ Justificacion: Es el costo de la operacion subSecuRojaMaxima llamada k veces, desde inicio a fin.

Algorithm 4 Backtracking

```
1: procedure CANTLIBRES(secu(int): elementos,secu(bool): validar) → int
2:     fin ← elementos.size();
3:     for i = 0; i < n; i++ do
4:         j ← i;
5:
6:         while (j < fin) do
7:             Rojos(elementos,validar,pilaRoja,i,j)
8:             EstablecerValidos(validar,pilaroja)
9:             Azul(elementos,validar,pilaAzul,0,j)
10:            EstablecerValidos(validar,pilaroja)
11:            res ← minimo(pilaAzul,PilaRoja)
12:        end while
13:    end for
14:    return res
```

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

Justificacion: Rojos y azules tienen la misma complejidad, Establecer validos es lineal por lo que queda acotado, nada mas que azul recorre el total de los elementos n veces teniendo una complejidad de n^{n+1}

2.4 Experimentación

Queremos demostrar que la complejidad de este algoritmo es exponencial, para ello vamos a tomar una secuencia de 40 números y mediremos el tiempo de ejecución del programa partiendo de 2 elementos y

sumando de a 2 sucesivamente hasta llegar a los 40.

2.4.1 experimento 1

Vamos a crear 3 secuencias de números y luego las vamos a comparar para ver cual de las tres tiene peor complejidad.

- Secuencia de todos los elementos iguales
- Secuencia de numeros random
- Secuencia de elementos decreciente (supuesto peor caso)

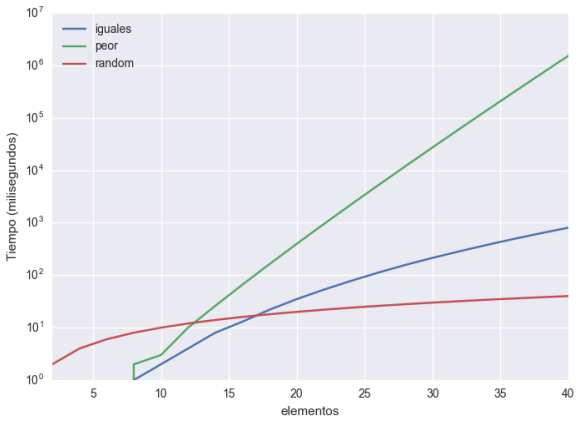


Figure 1: Complejidad temporal

Como podemos ver, en la figura 1 la que suponíamos peor caso,la de elementos decrecientes, es efectivamente así y se ve claramente como crece mas aceleradamente que el resto.

2.4.2 experimento 2

Una vez encontrada la secuencia de peor resultado, vamos a compararla contra una función exponencial y determinar el coeficiente de pearson entre dichas funciones para determinar la correlación entre las curvas y luego comparar contra una función polinomial.

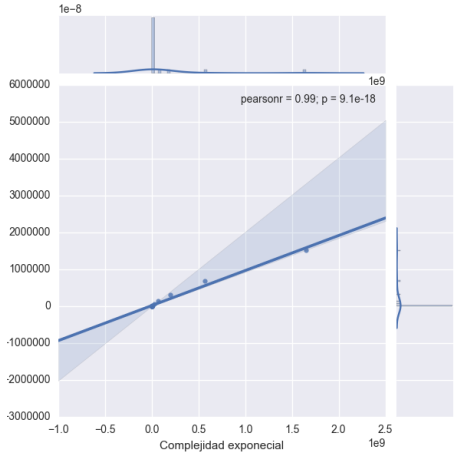


Figure 2: Correlación

Para determinar que la complejidad era exponencial, lo que hice fue comparar esta secuencia de tiempos contra una funcion exponencial 2^n y luego contra una función polinómica n^3.Como resultado

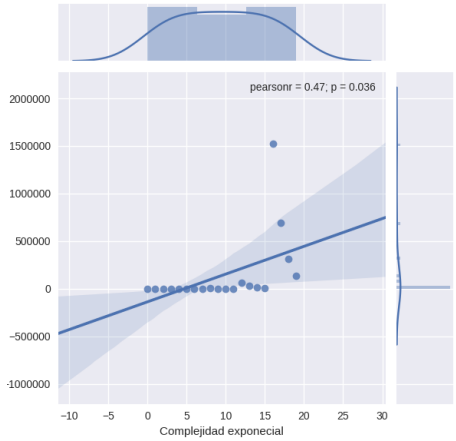


Figure 3: Correlación

nos da un coeficiente de pearson de 0,99 para la función exponencial y un coeficiente de para la función polinómica lo que nos dice que nuestro programa se corresponde con la de una función exponencialy esto nos permite concluir que nuestro programa es exponencial, como se aprecia en la teoría.

3 Problema II: Backtracking con poda

3.1 Introducción

Este tipo de solución es utilizando el mismo algoritmo en el paso anterior con alguna que otra modificación a la hora de tomar decisiones

3.2 Resolución del problema

La poda que vamos a aplicar es la siguiente, ya que el algoritmo recorre exahustivamente todos los elementos, pero. Resulta que si miramos e mas detalle, resulta que comparar o buscar una secuencia máxima en una tira de números que nunca puede llegar a superar la máxima encontrada por mas que se agreguen a todos, no tiene sentido ya que nunca podría superar al máximo por mas que pintemos todos los elementos faltantes por lo tanto ahí podemos parar el algoritmo.

Algorithm 5 Backtracking

```
1:  procedure subSecuRojaMaxima(secu(int): elementos,secu(bool): validar,pila(int):
    PilaMaxima,pila(int): Pila,int fin,int it,int n,bool notermine )
2:
3:
4:      while (pila ! = vacio() ∧ notermine ) do
5:          losquefaltan ← fin - it
6:          libres ← it - pila.size() - 1
7:
8:          if libres > losquefaltan then
9:              notermine ← false
10:
11:         else
12:
13:             while (it < fin ∧ notermine ) do
14:                 SecuCreciente(elementos,validar,pila,it,fin)
15:                 ..
16:                 ..
17:
18:             end while
19:         end if
20:     end while
```

Respecto a la complejidad no se debe hacer un análisis exhaustivo debido a que, esta poda en el peor de los casos no resulta efectiva solamente es útil para los casos promedios, ya que tarde o temprano el algoritmo puede caer en un caso en el que deba recorrer todo el árbol de soluciones

Solamente hace falta modificar la función ya descripta con anterioridad y su hermana azul.

3.3 Experimentación

Vamos a utilizar los mismos casos para el ejercicio uno y comparar contra las complejidades del ejercicio uno. Como se ve en la figura 3 nuevamente el peor caso sigue siendo el peor del ejercicio uno , pero ahora

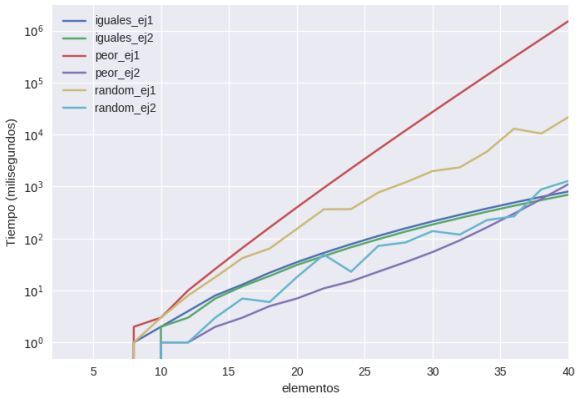


Figure 4: Complejidad temporal

se puede apreciar que hay una reducción significativa del tiempo de cómputo(10^3), sin embargo el resto de los experimentos se mantuvieron bastante igual, esto sucede porque la poda todo sirve para reducir tiempo de computo en ciertos casos, pero en el peor de los casos necesariamente tiene que visitar todos los elementos hasta el ultimo. Debido a esto la función continua siendo exponencial.

4 Problema III: Dinámica

4.1 Introducción

Este tipo de solución es la mas eficiente, ya que nos va a permitir resolver este problema en tiempo polinómico no como bactracking que es exponencial, de todas formas no es gratis, ya que vamos a requerir mas memoria.

4.2 Resolución del problema y representación

Para esta resolución vamos a usar una matriz de n+1 x n+1 elementos donde cada posición de la matriz indica la solución de elementos pintados si la posición [i,j] es pintada de rojo / azul. Las filas representan el ultimo rojo pintado y las columnas el ultimo azul.

Por ejemplo para la secuencia | 9 | 7 | 8 | 5 | 6 | 6 | la matriz es

$$\begin{bmatrix} 0 & 1 & 1 & 2 & 1 & 2 & 2 \\ 1 & 0 & 2 & 3 & 2 & 3 & 3 \\ 2 & 2 & 0 & 3 & 3 & 4 & 4 \\ 2 & 2 & 3 & 0 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 & 0 & 4 & 4 \\ 3 & 3 & 4 & 4 & 4 & 0 & 5 \\ 3 & 3 & 4 & 4 & 4 & 5 & 0 \end{bmatrix}$$

Donde en este caso, la mejor solución está al final de la matriz. Y es posible que la mejor solución no se encuentre al final por lo cual haya que revisar toda la matriz una vez terminada, pagando el costo de leer toda la matriz nuevamente, aunque también podriamos guardarnos el máximo histórico y nos ahorramos el leer todo nuevamente.

La función que usaremos para completar la matriz es la siguiente

G(k) representa " Cantidad maxima de elementos pintados hasta la posición k"
donde G(k) = $max(g(i,j))\forall i,j \leq k \in N$

$$\text{g(i,j)}= \begin{cases} 0 & \text{if } i{=}j, \\ 1 & \text{if } i{=}0 \wedge j = 1 \vee i{=}1 \wedge j = 0 \\ \text{upper(i,j)} & \text{if } i{>}j \\ \text{lower(i,j)} & \text{if } i{<}j \end{cases}$$

$$\text{upper(i,j)}= \begin{cases} 1 & \text{if } i{=}0 \\ \text{upper(i-1,j)} & \text{if } \neg \text{ PuedoPintarDeRojo(i,j)} \\ \max(1+ \text{ g(i,j)}, \text{ upper(i-1,j)} & \text{if PuedoPintarDeRojo(i,j)} \end{cases}$$

$$\text{lower(i,j)}= \begin{cases} 1 & \text{if } i{=}0 \\ \text{lower(i,j-1)} & \text{if } \neg \text{ PuedoPIntarDeAzul(i,j)} \\ \max(1+ \text{ g(i,j)}, \text{ lower(i,j-1)} & \text{if PuedoPintarDeAzul(i,j)} \end{cases}$$

Donde PuedoPIntarDeRojo y Azul son verdaderos si el elemento en la posición j es menor o mayor que el elemento en la posicion i.

4.3 Algoritmo-Pseudocódigo

Plantearemos una solución de la forma botton-up, vamos a declarar una matriz de $(n+1)^2$ elementos siendo n el tamaño de entrada y luego la llenaremos con la función g(i,j)

Algorithm 6 DINAMICA

```
1: procedure DINAMICA(secu: elementos, int: n)
2:     M ← CrearMatriz(n+1,n+1)
3:
4:     for i = 0; i < n; i++ do
5:         for j = 0; j < n; j++  do
6:
7:             if i = j ∨ i = 1 ∧ j = 0 ∨ i = 0 ∧ j = 1 then
8:                 // si i = j M[i,j] ← 0
9:                 // sino M[i,j] ← 1
10:            else
11:                if i > j then
12:                    M[i,j] ← upper(i,j)
13:                else
14:                    M[i,j] ← lower(i,j)
15:                end if
16:            end if
17:        end for
18:    end for
```

4.4 Complejidad

La complejidad de este algoritmo está determinado por la cantidad de operaciones basicas necesarias para completar toda la matriz, en cada posición de la matriz, se invoca a la función lower o upper que a su vez esta misma llama una cantidad i o j de posiciones de manera tal que.

por lo tanto la complejidad esta definida por: $\mathcal{O}(n^3)$
pues: $j = n, y \forall i \leq n. \sum_{i=1}^{n} i = n(n-1)n/2$

Respecto a la complejidad espacial, este algoritmo consume $\mathcal{O}(n^2)$ ya que hay que inicializár una matriz pero se puede optimizar consumiendo solamente $\mathcal{O}(2n)$ si se guarda el máximo histórico y la fila de soluciones correspondientes al indice i-1 y j-1, ya que son estos los últimos consultados para calcular la posición i,j.

4.5 Experimentación

Queremos demostrar que la complejidad de este algoritmo es polinomial, para ello vamos a tomar una secuencia de 730 números y mediremos el tiempo de ejecucion del programa partiendo de 2 elementos y sumando de a 2 sucesivamente hasta llegar a los 730.

4.5.1 experimento 1

Vamos a crear 3 secuencias de números construidas con los mismos criterios de los ejercicios anteriores y luego las vamos a comparar para ver cual de las tres tiene peor complejidad.

La figura `5` no muestra demasiada diferencia entre estas curvas, esto sucede porque el tiempo de cómputo requerido para este algoritmo dinámico se mantiene relativamente constante a la complejidad que es polinomial cúbica

4.5.2 experimento 2

Para demostrar que tiene como complejidad polinomial, lo que vamos a hacer es comparar los resultados de alguna de las secuencias contra una secuencia que representa una función exponencial y otra polinomial y veremos cual es la mejor de estas.

La correlación de pearson con una función cúbica es de exactamente 1 contra 0,42 de una función exponencial `ver figura 6 y 7` . Lo cual implica que la complejidad es polinomial cúbica.

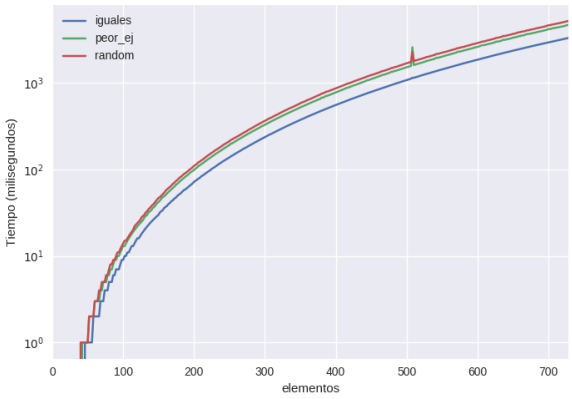


Figure 5: Complejidad temporal

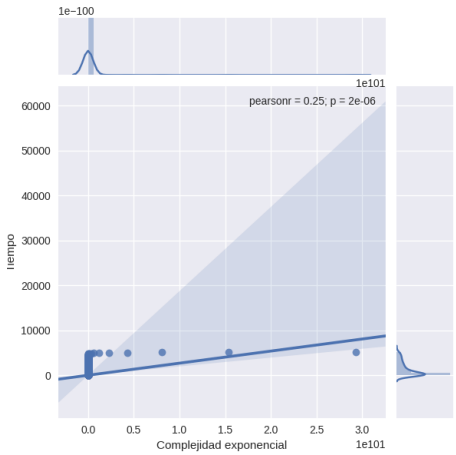


Figure 6: Correlación

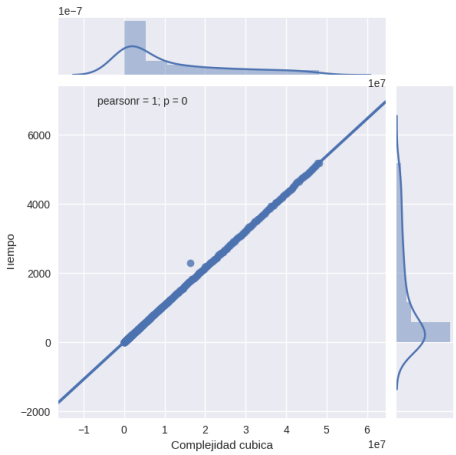


Figure 7: Correlación