



**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):	6

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 y 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 que 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 cuántos dejamos sin pintar. Una vez que tenemos todas las soluciones buscamos la óptima.

Se implementó 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 base en el que no hay índice siguiente y devolvemos 0 si podemos pintarlo (0 elementos sin pintar en un arreglo que contiene solo 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.

Un poco de codigo:

<pre> ej (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 */ res ← BT(0, arreglo, n, NULL, NULL) </pre>
<pre> BT (int i, arreglo[], n, ultimoRojo, ultimoAzul) → res: int Data: i = indice 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 </pre>

Análisis de complejidad: Ya dijimos que la cantidad de combinaciones posibles eran 3^n , y podemos decir que este algoritmo recorre como maximo todas estas instancias porque empieza desde el indice 0 y por cada numero del arreglo prueba las 3 combinaciones(o al menos las validas), entrando recursivamente al indice siguiente(i+1) y esos lo van a repetir hasta llegar al caso base que seria llegar al ultimo indice. Ahora, cada llamada recursiva hace una cantidad de operaciones $O(1)$ constantes (sumar, comparar, minimo) y si no es el caso base ademas hace 3 llamadas recursivas, pero reduciendo el n. Esto nos hace quedar que la complejidad de la funcion recursiva es $T(n) = 3T(n-1) + O(1)$ con $T(0) = 1$ como el caso base, que se puede demostrar facilmente (por induccion) que es $O(3^n)$. Otra forma de llegar es ver el arbol de ejecucion de la recursion, cada nodo desprende tres nodos hijos, vamos a tener un arbol de altura n, y como es un arbol ternario tenemos 3^n hojas, que son nuestras soluciones. Esta complejidad se considera exponencial y es muy mala para instancias grandes.

TODO Y PONER CASOS PEORES Y MEJORES GRAFICOS y DEMOSTRAR CORRECTITUD

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 calculamos, porque por ejemplo esta solucion va a hacer peor que una que ya calculamos anteriormente. Esto lo que provoca es una optimizacion porque dejamos de calcular cosas, aunque sigue siendo una busqueda exhaustiva.

Si recordamos el anterior algoritmo, para un indice calculaba el menor si lo pintabamos de rojo, despues de azul y despues 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 numeros sin pintar, y no vamos a encontrar una mejor porque esta es la minima. Esta es nuestra primer poda, una vez calculado una solucion parcial para un color, nos fijamos si es optima y si lo es nos salteamos(no computamos) las otras alternativas de colores. Tambien podriamos llevar una cuenta de cual es la solucion(entera) optima y no calculamos una que pueda ser peor que esa, por ejemplo si voy por el 3 indice y encontramos una solucion que solo no puede pintar un numero, solo tenemos que buscar una solucion que pinta todos los numeros, asi que no vamos a entrar a las recursiones de no pintar.

Para la implementacion de estas podas modificamos un poco la base del codigo del anterior ejercicio asi se nos es mas facil calcular los datos. Ahora la funcion recursiva tiene 2 parametros de entrada nuevos, el minimo encontrado en las soluciones ya calculadas y la cantidad de numeros que ya no pintamos y ahora en vez de devolver un resultado parcial, devuelve la cantidad de sin pintar para todo el arreglo(ya que tenemos la cantidad que no pintamos antes).

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 = indice 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

Podemos apreciar que en el peor de los casos tenemos que seguir calculando los 3 caminos así que no podemos mejorar la complejidad. Pero sin embargo vamos a recortar muchas ramas, que no necesitamos calcular, ya que por ejemplo si encontramos el óptimo pintando de rojo nos ahorramos 2/3 de las bifurcaciones al no calcular si pintamos de azul y si no lo pintamos y además si estamos al límite de números sin pintar comparados con el óptimo no avanzamos en las ramas de no pintar.

El algoritmo es correcto ya que las soluciones que descartamos sabemos que son peores a las que ya calculamos.

3. Ejercicio 3 (Programación dinámica):

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 con la respuesta.

Principio de optimalidad de Bellman: Un problema satisface el principio si una subsolución óptima debe ser solución del subproblema asociado a esa subsolución. En este problema podríamos decir que para una secuencia de largo n un subproblema sea la subsecuencia de 0 a $n - 1$, y si tenemos esta 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 subsolución 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 respuesta x'' tal que $x'' = x' - 1$ ya que solamente sacamos el último número, pero entonces $x'' < x$ o sea que x no era óptima, por ende entramos en un absurdo. Que esto lo supusimos al decir que había una solución mucho mejor que la de la solución. O sea que de una subsolución no podemos empeorar la solución o solo la empeoramos en una unidad, la dejamos igual cuando el nuevo número lo podemos pintar desde alguna de las combinaciones óptimas anteriores, o empeora en uno si no existe combinación óptima anterior desde la cual no podamos pintar este número 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 estén asociadas a las del subproblema, entonces no cumpliría el principio.

Como vimos en backtracking al pintar un número, solo nos importa cuáles fueron el último rojo y el último azul que se pintaron, entonces al agregar un nuevo a la secuencia, tenemos que buscar la combinación de 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 pintáramos. Entonces por problema tenemos que buscar cuál 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 cuál 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 último rojo último azul válida 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 que $\text{secuencia}_{r'} < \text{secuencia}_r$, 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 la que el índice de la fila representa el índice del último azul y el índice de la columna el del rojo, o sea que sea M la matriz, $M_{r,b}$ nos da cuál la subsolución 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 último 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 (o sea quedan n sin pintar), esto era el índice 0 , después pasamos para el índice 1 y calculamos todas las combinaciones nuevas, o sea 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 solucion.

<p>ej3 (int arreglo[], n) → res: int</p> <p>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</p> <pre> /* 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 res ← BT(0, arreglo, n, NULL, NULL, n, 0) /* 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) </pre>
<p>buscarOptimoPintandoDeRojo (int r, a, arreglo[], m, n) → res: int</p> <p>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 elejimos como ultimo pintado de rojo al indice r y de azul a a</p> <pre> /* 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 </pre>
<p>buscarOptimoPintandoDeAzul (int a, j, arreglo[], m, n) → res: int</p> <p>Data: a = El indice que estamos pintando de azul r = El indice que fijamos de rojo</p> <pre> /* 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 </pre>