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):

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

Backtraking 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 atras los cantidatos cuando sabemos que no son validos(que no pertenecen al espacio de soluciones validas del problema).

Para este problema queremos buscar el minimo de numeros sin pintar para una secuencia de numeros de largo n. Ya que un numero puede estar pintado de rojo, azul o sin pintar, y como tenemos n numeros, podemos tener como maximo 3^n posibles combinaciones, aunque podrian ser menos ya que de esas combinaciones hay posibilidades que no son validas 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, deberiamos encontrar la que menos numeros tiene sin pintar y esa va a ser la solucion a nuestro problema(se nos pide la cantidad no que combinacion de rojos y azules).

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

Se implemento este algoritmo de manera recursiva, dado el arreglo de numeros, el indice, y cual fue el ultimo rojo y ultimo azul pintado(si hay), devuelve la minima cantidad de numero sin pintar dentro de las posibilidades. Pedimos el ultimo rojo y el ultimo azul porque no me importa saber como los pinte si no los ultimos ya que eso me va a dar la limitante de si el proximo numero lo puedo pintar de rojo o azul(y descartamos los que no pude pintar). Entonces esta funcion recursiva guarda los resultados del indice siguiente para las 3 posibilidades(rojo, azul y sin pintar), se fija cual es la menor y devuelve esa(si es la que no se pinta tiene que sumarle uno), excepto que sea el caso baso en el que no hay indice siguiente y devolvemos 0 si podemos pintarlo(0 elementos sin pintar en un arreglo que contiene solo al ultimo elemento pintado de cualquier color) o 1 si no lo pudimos pintar, y despues se van a ir llenando las anteriores llamadas con estos casos bases.

Un poco de codigo:

```
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)
```

```
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
```

Analisis 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 asi 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 optimo pintando de rojo nos ahorramos 2/3 de las difurcaciones al no calcular si pintamos de azul y si no lo pintamos y ademas si estamos al limite de numeros sin pintar comparados con el optimo 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 (Programacion dinamica):

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

Principio de optimalidad de Bellman: Un problema satisface el principio si una subsolucion optima debe ser solucion del subproblema asociado a esa subsolucion. En este problema podriamos decir que para una secuencia de largo n un subproblema sea la subsecuencia de 0 a n - 1, y si tenemos esta subsolucion, entonces, la solucion al problema original va a ser menor o igual a esta subsolucion, 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 tambien supongamos que la solucion al problema es un x' tal que $x' = x - 2$, si pasa esto podriamos quedarnos con esa combinacion de colores(con la que nos da x') sacarle el ultimo numero y tendriamos una nueva respuesta x'' tal que $x'' = x' - 1$ ya que solamente sacamos el ultimo numero, pero entonces $x'' < x$ osea que x no era optima, por ende entramos en un absurdo. Que esto lo supusimos al decir que habia una solucion mucho mejor que la de la solucion. Osea que de una subsolucion no podemos empeorar la solucion o solo la empeoramos en una unidad, la dejamos igual cuando el nuevo numero lo podemos pintar desde alguna de la combinaciones optimas anteriores, o empeora en uno si no existe combinacion optima anterior desde la cual no podamos pintar este numero de rojo o de azul(notar que ahora hay combinaciones que por ahi no son optimas 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 cumpliria el principio ya que, por lo ultimo que dijimos, cuando no hay una combinacion optima anterior puede ser que las nuevas soluciones no esten asociadas a las del subproblema, entonces no cumpliria el principio.

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

El algoritmo dado se va a basar en eso, va a terminar calculando todas las combinaciones de ultimo rojo y ultimo azul para ver cual es la optima. Como calculamos cada una de estas combinaciones? recientemente dijimos que para pintar un numero de forma optima teniamos que buscar en la combinacion de ultimo rojo ultimo azul valida que sea optima. Entonces si queremos pintar el ultimo indice de rojo, sea r, y como ultimo azul un indice a con $0 \leq a < r$, deberiamos buscar el r' tal que sea optimo y que $secuencia_{r'} < secuencia_r$, para esto vamos a tener que recorrer de 0 a $r - 1$ buscando el que cumpla estas condiciones. Otro tema importante de la programacion dinamica es como guardamos los datos, una manera es tener una matriz de 2 dimensiones, en el que el indice de la fila representa el indice del ultimo azul y el indice de la columnas el del rojo, osea que sea M la matriz, $M_{r,b}$ nos da cual la subsolucion para esa combinacion de rojo y azul. Un caso que deje afuera pero hay que tener en cuenta es que hay una combinacion que es en la que no se pinta ninguno de rojo o de azul, o no se pinta ninguno, este indice se va a representar en la ultima fila y en la ultima columna, resumiendo para un problema de n numeros vamos a tener una matriz M de $(n + 1xn + 1)$ donde la la fila n+1 y la columna n+1 representan cuando el ultimo azul o ultimo rojo respectivamente son nulos.

Este es nuestro algoritmo, calculamos la matriz y buscamos en ella cual es el optimo, ahora tenemos que decidir en que 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 tambien, podemos ir de una manera constructiva, empezando en los casos bases, que son los cuando pintamos nada mas el primero de un color(que sabemos que va a dejar a todos los otros numeros siguientes sin pintar entonces la subsolucion es n - 1) y si no pintamos ninguno(osea quedan n sin pintar), esto era el indice 0, despues pasamos para el indice 1 y calculamos todas las combinaciones nuevas, osea ya teniamos 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 invalido porque no podemos pintar un mismo numero de dos colores). Una vez calculada toda la matriz buscamos el optimo dentro de esta y esa es nuestra solucion.

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
res ← BT(0, arreglo, n, NULL, NULL, n, 0)
`/* Ahora recorremos todos los indices */`
for $i \leftarrow 1$ **to** n **do**
`/* Y para cada indice, calculamos el optimo si pintaramos este de azul o de rojo y las combinaciones del otro */`
for $j \leftarrow o$ **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 elejimos 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 $ri \leftarrow 0$ **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

buscarOptimoPintandoDeRojo (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 $ai \leftarrow 0$ **to** *a - 1* **do**
if *ai != r* ***and*** *arreglo[ai] > arreglo[a]* ***and*** *m[r][ai] < optimoActual* **then**
optimoActual ← m[r][ai]

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 funcion auxiliar para buscar el optimo), eso ya nos da una complejidad de $O(n^3)$ y despues buscar el optimo dentro de la matriz tambien es $O(n^2)$ y despues 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 codigo dado no cortamos nunca y siempre completamos la matriz entonces para cualquier caso el codigo va a hacer $O(n^3)$. Una alternativa de analisis es considerar los casos que tenemos, podriamos decir que tenemos 3 variables en cada caso, el ultimo azul, el ultimo rojo, y el indice, entonces como cada una de estas variables esta acotada por n tenemos n^3 casos y son los que terminamos calculando.