Algoritmos y Estructuras de Datos III

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

Abril 2017

Trabajo Práctico 1

Alumno	LU	Correo electrónico
Seijo, Jonathan Adrián	592/15	jon.seijo@gmail.com

Índice

1. Introducción

1.1. Explicación del problema

Dada una secuencia A de números, se quieren pintar cada uno de ellos con rojo, azul o dejarlos sin pintar. Una aclaración importante es que los elementos de A no pueden modificarse, ni tampoco cambiarse su orden inicial. Lo unico que puede hacerse con ellos es colorearlos (o no).

Para que una secuencia de colores se considere **válida** es necesario que se cumplan ciertas condiciones:

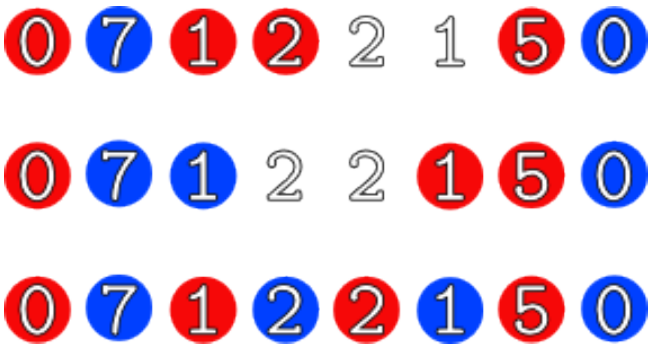
1. Todos los elementos de color rojo están ordenados por valor de forma estrictamente creciente
2. Todos los elementos de color azul están ordenados por valor de forma estrictamente decreciente

(Estrictamente significa que no hay numeros consecutivos iguales)

Las secuencias de colores válidas pueden tener diferentes cantidades de elementos sin pintar. El objetivo del problema es encontrar la **mínima cantidad de elementos sin pintar** de todas las secuencias válidas que pueden formarse a partir de A.

1.2. Ejemplo

Supongamos que $A = [0, 7, 1, 2, 2, 1, 5, 0]$. Veamos *algunas* de las posibles secuencias de colores válidas:



Consideremos los colores del tercer caso para ver que es una secuencia válida.

1. **Rojos:** [0, 1, 2, 5] (estrictamente crecientes)
2. **Azules:** [7, 2, 1, 0] (estrictamente decrecientes)

Podemos ver que diferentes formas de pintar de rojo y azul nos obligan a dejar algunos elementos sin pintar para que la secuencia sea válida. En el caso de este ejemplo la **mínima cantidad de elementos sin pintar** que puede obtenerse de A es **0**, como puede verse en la tercer combinación.

2. Backtracking

2.1. Solución naive

Llamo A a la secuencia de números que quiero pintar, y n a la cantidad de elementos en A. De todas las secuencias válidas de colores que puedo formar quiero saber cual es la mínima cantidad de elementos que puedo dejar sin pintar.

Una forma natural de pensar la solución es la siguiente: genero todas las formas de pintar posibles, y veo cual es el mínimo sin pintar que puede usarse para las secuencias que son válidas. Esa es la idea central detrás de ambos algoritmos de backtracking. Veamos entonces una posible implementación, la forma *naive*.

El primer elemento puede ser Rojo, Azul o Ninguno. Dado el color del primero, el segundo elemento puede también ser Rojo, Azul o Ninguno. Fijados el primero y el segundo, el tercero puede ser tomar cualquiera de las tres posibilidades, y así siguiendo.

Una vez fijos los colores de los n elementos, reviso si la secuencia de colores que se formó es válida. (Esto es, que los elementos rojos estén ordenados crecientemente y los azules decrecientemente, ambos de forma estricta).

Si la secuencia formada era válida, entonces cuento la cantidad de elementos sin pintar, y devuelvo ese número. La respuesta final es se consigue tomando el mínimo de todos los mínimos.

Como detalle de implementación, en caso de que la secuencia formada no sea válida, devuelvo un valor infinito para que no afecte al valor mínimo solución. Esta solución existe porque no pintar ningún elemento de ningún color es siempre una solucion válida **finita**

2.2. Pseudocódigo

```
procedure BACKTRACK(secuencia(Colores) colores, int actual)
    if actual = n then
        if EsValido(colores) then
            return CantSinPintar(colores)
        else
            return ∞
    else
        colores[actual] ← Rojo
        minimoConRojo ← backtrack(colores, actual + 1)

        colores[actual] ← Azul
        minimoConAzul ← backtrack(colores, actual + 1)

        colores[actual] ← Ninguno
        minimoSinPintar ← backtrack(colores, actual + 1)

        return Min(minimoConRojo, minimoConAzul, minimoSinPintar)
```

Auxiliares:

```
procedure EsValida(secuencia(Colores) colores)
    bool rojoValido ← EsCreciente(DameRojos(colores))          ▷ O(n)
    bool azulValido ← EsDecreciente(DameAzules(colores))       ▷ O(n)
    return (rojoValido ∧ azulValido)
```

```
procedure CantSinPintar(secuencia(Colores) colores)
    return Tamaño(DameSinPintar(colores))                      ▷ O(n)
```

Para resolver el problema original, llamo a la función con los siguientes parámetros:

```
procedure Resolver naive(secuencia(int) A)
    backtrack(secuenciaColoresVacia(n), 0)
```

2.3. Complejidad

El algoritmo presentado visita todas las posibles combinaciones de colores. Cada uno de los elementos tiene tres posibilidades, y como hay n elementos, la cantidad de combinaciones posibles es 3^n. Por lo tanto, visitar todas las posibilidades es $O(3^n)$.

Además, para cada combinación, se revisa en $O(n)$ si es una secuencia valida o no. Por lo tanto, la complejidad total del algoritmo es $O(n * 3^n)$

Otra forma de verlo es pensando en el arbol de recursión que se va formando al llamar a la función. Cada nivel representa al elemento iésimo de A, y cada nodo representa el color del elemento. De todo nodo se desprenden tres posibilidades hasta llegar al nivel n. Al llegar a una hoja, se decide si la secuencia *hasta esa hoja* es válida, en tiempo lineal.

Sabiendo que en un árbol ternario el nivel **i** tiene 3^i nodos, y sabiendo que el arbol tiene n niveles, la cantidad de nodos que se visitan es:

$$\sum_{i=0}^{n} 3^i = \frac{3^{n+1}}{2} = O(3^n)$$

El costo de las visitas de nodos no es el total, pues para cada una de las hojas se verifica si la secuencia obtenida es válida o no. Las hojas se encuentran en el útimo nivel n, entonces el árbol de recursión tiene 3^n hojas, donde cada hoja tiene costo $O(n)$. El costo de operar en las hojas entonces es $3^n * O(n) = O(n * 3^n)$

El costo **total** es la suma entre visitar todos los nodos y operar en las hojas, es decir:

$$O(3^n) + O(n * 3^n) = O(n * 3^n)$$

2.4. Solución con poda

El algoritmo anterior funciona, pero puede mejorarse si tenemos en cuenta algunas observaciones:

1. Pueden filtrarse las secuencias válidas a medida que vamos construyendo la secuencia de colores.
2. Recorriendo de izquierda a derecha, lo único que se necesita para saber si es válido pintar cierto elemento de algun color, es ver el valor del último elemento antes del actual que fue pintado de ese mismo color.
3. Puede conocerse la cantidad de elementos sin pintar a medida que se va pintando la secuencia.
4. La cantidad de elementos sin pintar **no** aumenta si se pinta rojo o azul.
5. Si la cantidad de elementos sin pintar es mayor al mínimo entontrado, entonces no vale la pena seguir explorando esa secuencia, pues la cantidad de elementos sin pintar no puede disminuir.

2.5. Pseudocódigo

Siguiendo la idea principal del algoritmo anterior, pero teniendo en cuenta las nuevas ideas, podemos mejorar nuestro algoritmo:

```
procedure BACKTRACK(int actual, int ultRojo, int ultAzul, int cantSinPintar)
    if actual = n then
        minimoTotal ← cantSinPintar
        return minimoTotal
    else
        if (ultRojo = n) ∨ (A[actual] > A[ultRojo]) then
            minRojo ← backtrack(actual + 1, actual, ultAzul, cantSinPintar)

        if (ultAzul = n) ∨ (A[actual] < A[ultAzul]) then
            minAzul ← backtrack(actual + 1, ultRojo, actual, cantSinPintar)

        if cantSinPintar + 1 < minimoTotal then
            minSinPintar ← backtrack(actual + 1, ultRojo, ultAzul, cantSinPintar + 1)

        return Min(minimoConRojo, minimoConAzul, minimoSinPintar)
```

Para resolver el problema original, llamo a la funcion con los siguientes parámetros:

```
procedure RESOLVER PODA(secuencia(int) A)
    backtrack(0, n, n, 0)
```

Como detalle de implementación, cuando el último rojo (o azul) es n, significa que no hay ningún rojo (o azul) aún, porque los índices del arreglo comienzan en cero.

2.6. Complejidad con poda

Haciendo un análisis similar al del algoritmo naive, pensando en el arbol de recursión, tenemos que como cota superior se visitan todos los nodos del árbol. La validez de la secuencia se comprueba a medida que se desciende en la recursión, pues no se entra al siguiente paso si la propiedad de los colores no se cumple. Además, se va contando la cantidad de elementos sin pintar a medida que se eligen colores, por lo que tampoco se usa tiempo en eso.

Como todo el costo se encuentra en la cantidad de nodos que se visita, y como la cantidad máxima de nodos es $O(3^n)$, el costo de obtener la respuesta usando este algoritmo con poda es $O(3^n)$, mejorando la complejidad del anterior.

3. Programación Dinámica

3.1. Solución topdown

Aca va la explicacion de la solucion y el por que el algoritmo es correcto

3.2. Pseudocódigo topdown

```
procedure RESOLVER TOPDOWN(secuencia(int) A)
    Matriz3 DP ← Matriz3(n, −1)                    ▷ Matriz de 3 dimensiones, llena con −1
    for ultRojo ∈ [0..n] do
        for ultAzul ∈ [0..n] do
            minSinPintar ← Min(minSinPintar, Solución(n − 1, ultRojo, ultAzul))
    return minSinPintar
```

```
procedure SOLUCIÓN(int actual, int ultRojo, int ultAzul)
    if actual = −1 then return 0
    if DP[actual][ultRojo][ultAzul] ≠ −1 then return DP[actual][ultRojo][ultAzul]

    minRojo ← TopdownCasoRojo(actual, ultRojo, ultAzul)
    minAzul ← TopdownCasoAzul(actual, ultRojo, ultAzul)
    minSinPintar ← TopdownCasoSinPintar(actual, ultRojo, ultAzul)

    return Min(minRojo, minAzul, minSinPintar)
```

procedure TopdownCasoRojo(int *actual*, int *ultRojo*, int *ultAzul*) ▷ Si no hay rojo o si el actual es azul, entonces no puedo considerar que el actual sea rojo
  if $(ultRojo = n) \vee (actual = ultAzul)$ **then**
    return ∞
  else ▷ Si soy el último rojo, ó si puedo serlo porque cumplo la propiedad:
    if $(actual = ultRojo) \vee (i < ultRojo \wedge A[i] < A[ultRojo])$ **then**
      return Solución($actual - 1$, *actual*, *ultAzul*)
    else
      return ∞

procedure TopdownCasoAzul(int *actual*, int *ultRojo*, int *ultAzul*)
▷ Si no hay azul o si el actual es rojo, entonces no puedo considerar que el actual sea azul
  if $(ultAzul = n) \vee (actual = ultRojo)$ **then**
    return ∞
  else ▷ Si soy el último azul, ó si puedo serlo porque cumplo la propiedad:
    if $(actual = ultAzul) \vee (i < ultAzul \wedge A[i] > A[ultAzul])$ **then**
      return Solución($actual - 1$, *ultAzul*, *actual*)
    else
      return ∞

procedure TopdownCasoSinPintar(int *actual*, int *ultRojo*, int *ultAzul*)
▷ No puede pasar que el actual sea rojo o azul, pero que lo quiera dejar sin pintar
  if $(actual = ultRojo) \vee (actual = ultAzul)$ **then**
    return ∞
  else
    return 1+ Solución($actual - 1$, *ultRojo*, *ultAzul*)

3.3. Complejidad

Aca va la justificacion de por que el algoritmo topdown es n^3, consultar mis notas para orientacion