IIC2283 — Diseño y Análisis de Algoritmos
Ayudantía 8

Problema 1: Cubrimiento de pasillos

Parte 1 Se necesita cubrir un pasillo de largo n con bloques, que pueden ser de largo 1, 2 o 4. Describa un algoritmo que permita determinar la cantidad de cubrimientos posibles, y analice su complejidad.

Idea de solución La cantidad de cubrimientos se puede representar con la siguiente recurrencia, que se obtiene al considerar las formas de cubrir la última posición:

$$C(n) = \begin{cases} 1 & n = 1 \\ 2 & n = 2 \\ 3 & n = 3 \\ 6 & n = 4 \\ C(n-1) + C(n-2) + C(n-4) & n > 4 \end{cases}$$

Los valores de C pueden calcularse de forma secuencial usando programación dinámica. El algoritmo funciona en tiempo $O(n)$, por que por cada subproblema se realiza una cantidad constante de operaciones, y son n subproblemas.

Parte 2 Se consideran dos tipos de bloques: los bloques blancos que tienen largo 1, y los bloques azules que tienen largo variable, pero siempre mayor o igual a un parámetro m.

Utilizando esos bloques se busca cubrir un pasillo de largo n, de forma que dos bloques azules siempre estén separados por al menos un bloque blanco. Describa y analice un algoritmo que determine la cantidad total de cubrimientos posibles dados los parámetros n y m.

Idea de solución Similar a la parte 1, pero donde la recurrencia debe considerar todas las posibilidades para el largo del bloque final.

$$C(n) = \begin{cases} 1 & n < m \\ C(n-1) + \sum_{\ell=m}^{n} C(n-\ell-1) & n \geq m \end{cases}$$

Al calcular los valores por programación dinámica, en el subproblema i se tiene que el tiempo es:

$$ts(i) = i - m$$

ya que se debe realizar esa cantidad de sumas.

Luego, el tiempo total del algoritmo está dado por:

$$\begin{aligned} T(n) &= O(m) + \sum_{i=m}^{n} ts[i] \\ &= O(m) + \sum_{i=m}^{n} (i-m) \\ &= O(m) + \sum_{i=0}^{n-m} i \\ &\in O(n^2) \end{aligned}$$

donde el término $O(m)$ corresponde a las primeras m entradas que son iguales a 1.

Problema 2: Arreglos unimodales

Decimos que un arreglo A de n enteros es *unimodal* si existe un índice i, tal que para todo $0 \leq j < i$ se cumple que $A[j] < A[j+1]$ y para todo $i < k < n$ se cumple que $A[k-1] > A[k]$.

Parte 1 Defina un algoritmo que encuentre el máximo valor en un arreglo unimodal y analice su complejidad.

Idea de solución Se utiliza un algoritmo basado en dividir para conquistar similar a búsqueda binaria.

```
function MaxUnimodal(A, i, j)
    ℓ ← ⌊(i+j)/2⌋
    if A[ℓ − 1] < A[ℓ] and A[ℓ] > A[ℓ + 1] then
        return ℓ
    else if A[ℓ − 1] < A[ℓ] and A[ℓ] < A[ℓ + 1] then
        return MaxUnimodal(A, ℓ, j)
    else
        return MaxUnimodal(A, i, ℓ)
    end if
end function
```

El algoritmo funciona en tiempo $T(n) = T(\lceil n/2 \rceil) + O(1) \in \Theta(\log(n))$.

Parte 2 Desarrolle y analice un algoritmo que encuentre el k-ésimo elemento en un arreglo unimodal.

Idea de solución Dado un arreglo unimodal, se puede encontrar el valor máximo, y luego dividir en dos arreglos ordenados.

Luego, el problema se reduce a encontrar el elemento k-ésimo de la unión de dos arreglos ordenados, que se puede resolver utilizando un algoritmo basado en dividir para conquistar.

El algoritmo compara las posiciones $k/2$ de cada uno, y luego descarta los elementos menores a la posición menor:

```
function KTH(A = [a_1, ..., a_n], B = [b_1, ..., b_m], k)
    if k = 1 then
        return mín(a_1, b_1)
    end if
    I ← ⌊k/2⌋
    if A[I] ≤ B[I] then
        return KTH([a_{I+1}, ..., a_n], B, k − I)
    else
        return KTH(A, [b_{I+1}, ..., b_n], k − I)
    end if
end function
```

En ese algoritmo se asume que los arreglos A y B son lo suficientemente largos para cualquier índice, lo que es un supuesto razonable por que siempre se puede considerar que las posiciones mayores a n o m tienen valor ∞. El tiempo del algoritmo no depende de n, y pertenece a $O(\log k)$.

Con esta idea, se puede construir un algoritmo para arreglos unimodales, donde se considere la posición $k/2$ desde cada extremo, y luego por cada una se evalúe si sobrepasa el máximo como en la parte 1. Finalmente con los valores de esas posiciones –que pueden ser ∞– se sigue como en KTH. Este algoritmo funciona en tiempo $O(\log k)$.

Parte 3 Determine si es posible adaptar el algoritmo de la parte 1 para funcionar cuando se permiten elementos repetidos contiguos. Es decir, cuando la primera parte es no decreciente, y la segunda no creciente.

Solución No es posible en tiempo $O(\log(n))$. Se puede demostrar con la técnica del adversario que todo algoritmo correcto para este problema debe consultar al menos n posiciones.

Para lograr esto, basta que el adversario siempre responda con el mismo valor, y luego, si el algoritmo no revisa todas las posiciones, entonces el adversario puede modificar alguna posición no consultada y asignarle un valor mayor al retornado por el algoritmo.

Problema 3: Saltos de línea

En un procesador de texto se necesita dividir un párrafo en lineas, tales que el largo de cada una no exceda un limite superior, que depende del tamaño de la página, los márgenes y otros elementos. De entre las distintas asignaciones posibles, se prefiere la que optimice alguna métrica de error definida por el sistema, que usualmente busca generar líneas balanceadas y cercanas al tamaño máximo.

Parte 1 Describa un algoritmo para ese problema que minimice la cantidad de líneas utilizadas, y analice su complejidad.

Idea de solución Se utiliza un algoritmo greedy, donde se agregan secuencialmente todas las palabras posibles a la primera línea, y se continúa recursivamente con las palabras restantes, desde la línea siguiente.

Se demuestra que el algoritmo minimiza la cantidad de líneas por inducción en la cantidad de líneas generadas.

Si el algoritmo greedy produce una línea, entonces el resultado es trivial. Por otra parte, si el algoritmo genera $n+1$ líneas, entonces se considera la instancia con las palabras desde la primera de la segunda línea. En este caso, por hipótesis de inducción el algoritmo es óptimo. Luego, como por construcción la primera línea tiene la mayor cantidad posible de palabras, entonces la solución formada por la primera línea, concatenada con la solución greedy desde la segunda debe ser óptima, por que la única alternativa sería traspasar palabras desde la primera línea a las siguientes, lo que no puede reducir la cantidad de líneas. Finalmente, como esta solución es igual a la solución greedy original, se obtiene el resultado buscado.

Parte 2 Se define el costo de una línea como la diferencia entre la cota para el largo, y la suma de los largos de sus palabras. Luego, se define el costo de un párrafo como la suma de los costos de todas las líneas, sin considerar la última.

Determine si el algoritmo de la parte 1 entrega una solución óptima bajo esta métrica.

Idea de solución Si consideramos todas las líneas, es decir sin ignorar la última, el costo de un párrafo está dado por:

$$\sum_i L - |\bar{w}_i|$$

donde la suma se realiza por todas las líneas i, y $|\bar{w}_i|$ representa la suma de los largos de las palabras en la línea i. Desarrollando esa suma, se obtiene que el costo es igual a:

$$pL - \sum_{w \in W} |w|$$

donde p es el número de líneas, y $|w|$ es el largo de cada palabra. Los valores de $\sum_{w \in W} |w|$ y de L solo dependen de la entrada, por lo que en este caso, el costo del párrafo solo depende de la cantidad de líneas, y es óptimo por la parte 1.

Si se ignora la última línea, entonces se puede considerar otra instancia, donde se eliminan todas las palabras de la última línea menos la primera. Luego, si existe alguna solución que tenga costo menor a la solución greedy, entonces utilizaría menos líneas que esta, lo que sería una contradicción con la parte 1.

Parte 3 Utilizando la misma definición de costo para una línea, ahora se define el costo de un párrafo como el máximo entre los costos de sus líneas, sin considerar la última. Determine si el algoritmo anterior es correcto en este caso.

Solución El algoritmo no es correcto. Un contraejemplo se obtiene con $L = 7$, y palabras

$$aa, aa, aa, aa, bbbbbb.$$

En el ejemplo la solución greedy tiene costo 5, mientras que el óptimo es 3.

En este caso, se puede calcular utilizando programación dinámica considerando como subproblemas los subarreglos desde el comienzo.

Para la posición i, se consideran todas las líneas posibles que terminan en la palabra i, y comienzan en alguna posición j, y luego, por cada posibilidad se define el costo como la combinación entre el costo hasta la posición $j-1$, con el de la línea (j, i). En este caso, se definiría como el máximo entre ambos valores. Luego, se define el costo óptimo hasta la posición i como el mínimo de todas las posibilidades descritas antes. Estos valores se calculan hasta el final del input, y luego se evalúan todas las posibilidades para la última línea. Por cada una, se considera el costo hasta la posición anterior a la línea, retornando el mínimo entre estos.