Análisis y Diseño de Algoritmos

Tarea 2

Daniel Rojo Mata
danielrojomata@gmail.com

Fecha de Entrega: 26 de agosto de 2024

Problema 1

Da un algoritmo con complejidad de tiempo $O(n)$ que reciba un arreglo A de longitud n, con enteros en sus entradas, y ordenado de manera ascendente, y devuelva un arreglo B de longitud $m \leq n$ que elimine las repeticiones en A.

Solución

Algoritmo CrearArreglo

- `Entrada:` Un arreglo A de longitud n, con enteros en sus entradas y ordenado de manera ascendente.
- `Salida:` Un arreglo B de longitud $m \leq n$ con entradas vacías.

```
1      m = 1
2      for i=2 up to n
3          if A[i] distinto a A[i-1]
4               m = m + 1
5          end if
6      end for
7      crear B de longitud m
8      return B
```

Algoritmo SinRepetidos

- `Entrada:` Un arreglo A de longitud n, con enteros en sus entradas y ordenado de manera ascendente.
- `Salida:` Un arreglo B de longitud $m \leq n$ tal que elimina las repeticiones en A.

```
1     CrearArreglo(A, n)
2     B[1] = A[1]
3     j = 2
4     for i=2 up to n
5         if A[i] distinto a A[i-1]
6             B[j] = A[i]
7             j = j + 1
8         end if
9     end for
10    return B
```

Corrección del Algoritmo

Lo que hace el algoritmo `CrearArreglo` es crear un arreglo cuya longitud es m, siendo este valor el número de elementos que no se repiten en el arreglo A.
La línea 6 hace alusión a *crear un arreglo*, se escribió de esa manera pues no en todos los lenguajes de programación se crean los arreglos de la misma manera, es por ello que se expresa así.
La salida de este algoritmo es un arreglo B cuya longitud es $m \leq n$.

Dependiendo de la manera en que se haya creado el arreglo, serán las entradas de éste, pero se cumple, que las entradas deben ser enteros por la propia forma de la creación.
En resumen B tendrá una longitud de m y, de acuerdo con la naturaleza del problema, sus entradas serán enteros (aunque inicialmente pueden estar vacías o no inicializadas).

El algoritmo `CrearArreglo` es correcto por las siguientes razones:

1. **Contar Elementos Únicos:** El algoritmo recorre el arreglo A y cuenta cuántos elementos únicos hay. Comienza con el primer elemento y cuenta cada vez que encuentra un elemento diferente al anterior. Esto asegura que cada elemento único en A se cuenta una sola vez.
2. **Determinar Longitud del Nuevo Arreglo:** La variable m guarda la cantidad de elementos únicos. Al final del recorrido, m indica el número total de elementos únicos en A.
3. **Crear el Nuevo Arreglo:** Usando el valor de m, el algoritmo crea un nuevo arreglo B que tiene exactamente la longitud necesaria para almacenar todos los elementos únicos.

Teniendo este arreglo creado, se procede a demostrar la correctitud del algoritmo `SinRepetidos`.

Se propone el siguiente invariante, en donde, por notación, $A[0 : j]$ denota todos los elementos del arreglo A desde el índice 0 hasta el índice j.

LEMA: Al término de la iteración `i`*, el subarreglo* $B[1:j]$ *contiene elementos únicos de* $A[1:i]$.

Dicho de otro modo:

LEMA: Después de la iteración `i`*, el* j*-ésimo elemento de* B *no se ha repetido.*

Demostración Lema:

Se hace uso de inducción matemática sobre el número de iteraciones.

- **Caso Base: n = 1**. Esto quiere decir que en A existe al menos un elemento. Nótese que no se inicializa el `ciclo 4-9`, por lo que solo se ejecuta la línea 2, esto es, el algoritmo hace $B[1] = A[1]$ y como no entra al ciclo solo retorna B, teniendo en éste un elemento único de A cumpliéndose el invariante.
- **Paso inductivo.**
  - **Hipótesis de Inducción:** Suponga que el algoritmo es correcto para la iteración k, donde $2 \leq k \leq n$, es decir, suponga que es cierto que:
    Al término de la iteración `k`, el subarreglo $B[1:j]$ contiene elementos únicos de $A[1:k]$

  Se demuestra la validez del invariante para la iteración $k+1$.
  Después de la iteración k, nótese que $j = k$, esto por la línea 2.

  Por la hipótesis de inducción, sabemos que el subarreglo $B[1:j]$ al término de la iteración k contiene todos los elementos únicos de $A[1:k]$. Esto implica que B no tiene duplicados hasta ese punto.

  Ahora, en la iteración $k+1$, el algoritmo compara $A[k+1]$ con $A[k]$ en la línea 5:

  $$A[k+1] \neq A[k].$$

  Si esta comparación es verdadera, entonces sabemos que $A[k+1]$ no está en el subarreglo $B[1:j]$, ya que, por hipótesis, $A[k]$ es el último elemento agregado y no hay duplicados en B. Por lo tanto, asignar:

  $$B[j] = A[k+1]$$

  asegura que $B[1:j+1]$ contenga todos los elementos únicos de $A[1:k+1]$.

Así, el invariante se mantiene, por lo que el algoritmo es correcto.

Complejidades

En el algoritmo `SinRepetidos` se hace uso de un ciclo `for` (líneas 4-9), sin embargo, el cuerpo de éste está conformado por operaciones que toman tiempo constante. Por ejemplo, en la línea 4, se está accediendo a elementos de un arreglo y se están haciendo comparaciones, en la línea 5 se hacen asignaciones y en la línea 6 sumas. Por lo que dentro del ciclo `for`, solo se ejecutan operaciones de tiempo constante, y como el cíclo se ejecuta desde $i = 2$ hasta n, entonces se hacen $n - 1$ operaciones, dando como resultado una complejidad lineal, es decir, del orden de $O(n)$.

Ahora, el algoritmo `CrearArreglo` hace uso de un ciclo `for` pero nuevamente el cuerpo conformado por operaciones constantes, es así que se tiene una complejidad de $O(n)$, luego, la creación de un arreglo de longitud n generalmente toma tiempo lineal, es decir, $O(n)$, debido a la necesidad de reservar espacio y, en algunos casos, inicializar los valores de los elementos del arreglo.

Por lo tanto la complejidad temporal total del algoritmo `SinRepetidos` es del orden de $O(n)$.

La complejidad espacial estará sujeta a la creación del arreglo, que nuevamente tomará tiempo $O(n)$, además de las constantes utilizadas, quedando al final la complejidad $O(n)$.

✓✓

Problema 2

Dado dos arreglos de enteros A y B, ambos de longitud n, decimos que A precede lexicográficamente a B, y lo denotamos $A \preceq B$, si se cumple $\forall\, 1 \leq i \leq n$, $A[i] \leq B[i]$. Y si $A \preceq B$, decimos que A estrictamente precede lexicográficamente a B, y lo denotamos $A \prec B$, si se cumple $\exists\, 1 \leq i \leq n$, $A[i] < B[i]$.

Da un algoritmo que reciba dos arreglos de enteros A y B y devuelva 1 si $A \prec B$, 2 si $A \preceq B$ pero $A \not\prec B$, y 3 en otro caso. Tu algoritmo debe tener complejidades de tiempo y espacio (auxiliar) $O(n)$ y $O(1)$, respectivamente.

Nota: Observa que la entrada al algoritmo es de tamaño $2n$, pero como las constantes no importan en notación asintótica, pedir complejidad de tiempo $O(n)$ quiere decir lineal con respecto a la entrada.

Solución

Se hace uso de dos algoritmos.

Algoritmo lexicografico

- `Entrada:` Dos arreglos de enteros A y B, ambos de longitud n.
- `Salida:` `True` si A precede lexicográficamente a B, ($A \preceq B$), `False` en caso contrario.

```
1     for i = 1 up to n do
2         if A[i] > B[i] then
3             return False
4         end if
5     end for
6     return True
```

Algoritmo estrictamente_lexicografico

- `Entrada:` Dos arreglos de enteros A y B, ambos de longitud n.
- `Salida:` 1 si $A \prec B$, 2 si $A \preceq B$ pero $\neg(A \prec B)$, y 3 en otro caso.

```
1     lexico = lexicografico(A, B)
2     if lexico then
3         for i = 1 up to n do
4             if A[i] < B[i] then
5                 return 1
6         end for
7         return 2
8     else if
9         Retornar 3
10    end if
```

Corrección del Algoritmo

Se hace uso de dos algoritmos para poder solucionar el problema.
El primero de ellos, `lexicografico`, hace la validación tal que A precede lexicográficamente a B. Para ello hace uso de un ciclo `for` recorriendo todos los elementos del arreglo, en caso de que se cumpla la condición $A[i] > B[i]$ para algún valor i desde 1 hasta n, entonces se retorna `False`, pues no se cumple la condición requerida para ser lexicográficos.
Caso contrario retorna `True`, siendo los arreglos lexicográficos.
Al hacer validaciones sencillas comparando los arreglos, el algoritmo es termina y retorna lo que se solicita, por lo que el algoritmo es correcto.

El segundo algoritmo usa el primero como punto de partida.
En la línea 1 se crea la variable `lexico`, cuyo valor asignado es la llamada a la función `lexicografico(A,B)`, por lo que el valor de `lexico` es un booleano; `True` o `False`.
Si es el caso en que `lexico = True`, entonces se procede a entrar al condicional 2, indicando que en efecto es cierto que $A \preceq B$, por lo que inicia un ciclo `for` para corroborar si pasa que $A \prec B$ o no. Para el primer caso, se hace un comparativo de $A[i] < B[i]$, y si para algún valor i entre 1 y n, se satisface la condición anterior, entonces A estrictamente precede lexicográficamente a B retornando 1. Para el segundo caso, entonces regresa 2 mostrando que no es cierto que A estrictamente precede lexicográficamente a B, esto se aprecia en el `return` de la línea 7.

Finalmente, si `lexico = False` es porque no se cumple con lo anterior escrito.

Así, al analizar los posibles casos del algoritmo y al tener un ciclo `for` comparando elementos de los arreglos, es correcto decir que el algoritmo termina y que regresa lo solicitado, por lo que el algoritmo es correcto.

Complejidades

En el primer algoritmo, `lexicografico`, se hace uso de un ciclo `for` desde 1 hasta n, pero nótese que dentro del cuerpo de éste solo se tienen operaciones constantes, pues se están haciendo operaciones de acceso y de comparación, por lo que la complejidad temporal es del orden de $O(n)$ y para este caso, no se hace uso de memoria extra, por lo que la complejidad espacial es del orden $O(1)$.

Ahora, en el algoritmo `estrictamente_lexicografico` se hace uso de un ciclo `for`, sin embargo, se realizan operaciones constantes dentro de éste. De igual forma se hacen operaciones de acceso y de comparación, por lo que la complejidad temporal también es del orden de $O(n)$. Para la complejidad temporal se considera el uso de la variable `lexico`, la cual almacena la llamada a `lexicografico`, sin embargo, almacena un valor booleano, por lo que solo usa un espacio de memoria constante.

Así, la complejidad temporal debido al uso de ambos algoritmos es $O(n)$, mientras que la complejidad espacial es del orden de $O(1)$.

✓✓

Problema 3

Soluciona el problema 1-1 del libro de algoritmos de Cormen et al., 4a. edición (es el que pide llenar una tabla que compara el crecimiento de funciones, en la página 15).
Problema: Para cada función $f(n)$ y tiempo t en la siguiente tabla, determina el tamaño máximo n de un problema que se puede resolver en el tiempo t, suponiendo que el algoritmo para resolver el problema toma $f(n)$ microsegundos

Solución

Primero se hace la conversión del tiempo

- **1 segundo** = 1 segundo = 10^6 microsegundos
- **1 minuto** = 60 segundos = 6×10^7 microsegundos
- **1 hora** = 3600 segundos = 3.6×10^9 microsegundos
- **1 día** = 86400 segundos = 8.64×10^{10} microsegundos
- **1 mes** (30 días) = 2.592×10^6 segundos = 2.592×10^{12} microsegundos
- **1 año** (365 días) = 3.1536×10^7 segundos = 3.1536×10^{13} microsegundos
- **1 siglo** (100 años) = 3.1536×10^9 segundos = 3.1536×10^{15} microsegundos

A continuación se muestran los despejes utilizados para calcular los valores en cada tabla, algunos valores se calcularon de manera numérica, además de que en otros se redondearon los valores.

1. Para $f(n) = \lg n$:
$$\lg n = t \implies n = 2^t$$
2. Para $f(n) = \sqrt{n}$:
$$\sqrt{n} = t \implies n = t^2$$

	1 segundo	1 minuto	1 hora	1 día	1 mes	1 año	1 siglo
$\lg n$	2^{10^6}	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.592\times10^{12}}$	$2^{3.1536\times10^{13}}$	$2^{3.1536\times10^{15}}$
$\sqrt{n}$	10^{12}	3.6×10^{15}	1.296×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.95×10^{30}
n	10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.592×10^{12}	3.1536×10^{13}	3.1536×10^{15}
$n \lg n$	1.44×10^5	4.77×10^6	1.14×10^8	2.6×10^9	6.65×10^{10}	7.89×10^{11}	9.96×10^{13}
n^2	10^3	7.75×10^3	6×10^4	2.94×10^5	1.61×10^6	5.61×10^6	5.61×10^7
n^3	10^2	3.91×10^2	1.53×10^3	4.29×10^3	1.37×10^4	3.17×10^4	14.7×10^5
2^n	19	25	31	36	41	44	50
$n!$	9	11	12	13	15	16	17

Cuadro 1: Tamaño más grande n que se puede resolver en tiempo t

3. Para $f(n) = n$:
$$n = t$$

4. Para $f(n) = n \lg n$: Resolvemos numéricamente para n

5. Para $f(n) = n^2$:
$$n^2 = t \implies n = \sqrt{t}$$

6. Para $f(n) = n^3$:
$$n^3 = t \implies n = \sqrt[3]{t}$$

7. Para $f(n) = 2^n$:
$$2^n = t \implies n = \log_2 t$$

8. Para $f(n) = n!$: Resolvemos numéricamente.

Problema 4

Observación: A lo largo de los ejercicios 1, 2 y 3 se utilizará la siguiente definición:

$$O\Big(g(n)\Big) = \{\ f(n) : \exists\ c, n_0 > 0\ t.q.\ \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)\} \tag{1}$$

Observación: En el ejercicio 4 utilizará la siguiente definición:

$$\Theta\Big(g(n)\Big) = \{\ f(n) : \exists\ c_1, c_2, n_0 > 0\ t.q.\ \forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)\} \tag{2}$$

Prueba lo siguiente:

1. $9n^2 + 356n + 7 \in O(n^2)$

   Solución 1

   Con base a la definición 1, se tiene que probar lo siguiente:

   $$\exists\ c, n_0 > 0\ t.q.\ \forall n \geq n_0,\ 0 \leq 9n^2 + 356n + 7 \leq c \cdot n^2$$

   Sea $n_0 = 1$. Sea $n \geq n_0 = 1$, entonces es cierto que: $n \geq 1$, por lo que se concluye:

$$n \leq n^2. \tag{3}$$

Por la afirmación en 3 se cumple:

$$7 \leq 7n \leq 7n^2$$

Así:

$$7 \leq 7n^2 \tag{4}$$

Ahora, por 3 se satisface:

$$356n \leq 356n^2 \tag{5}$$

Por otra parte, siempre es válido:

$$9n^2 \leq 9n^2 \tag{6}$$

Sumando 4, 5 y 6 se tiene la siguiente cadena de desigualdades:

$$7 + 356n + 9n^2 \leq 7n^2 + 356n^2 + 9n^2$$

$$7 + 356n + 9n^2 \leq n^2(7 + 356 + 9)$$

$$7 + 356n + 9n^2 \leq 372n^2$$

Así pues, existen n_0 y c tal que se cumple

$$\forall n \geq n_0,\ 0 \leq 9n^2 + 356n + 7 \leq c \cdot n^2$$

A saber $c = 372$ y $n_0 = 1$.

Por lo tanto; $9n^2 + 356n + 7 \in O(n^2)$ ✓✓

2. $5n^3 \log n \in O(n^4)$

Solución 2

Con base a la definición 1, se tiene que probar lo siguiente:

$$\exists\ c, n_0 > 0\ t.q.\ \forall n \geq n_0,\ 0 \leq 5n^3 \log n \leq c \cdot n^4$$

Sea $n_0 \geq 1$ y sea $n \geq n_0 = 1$, por lo que $n \geq 1$.
En particular, por la relación anterior, es cierto que:

$$n^3 > 0 \tag{7}$$

Ahora, se sabe que para todo valor de $n \in \mathbb{R}^+$ se cumple:

$$log(n) < n$$

Multiplicando por $5n^3$ ambos lados de la desigualdad y sabiendo que es un valor positivo por 7, se sigue que:

$$5n^3 log(n) < 5n^3 \cdot n$$

De donde se obtiene:

$$5n^3 log(n) < 5n^4$$

Así pues, existe $c = 5$ tal que se cumple que $5n^3 log(n) < c \cdot n^4$, para $n \geq n_0$, que era lo que se quería probar. ✓✓

3. $n^3 \log n \notin O(n^3)$

Solución 3

Se demuestra por contradicción.
Supóngase que: $n^3 \log n \in O(n^3)$, entonces, por la definición en 1, es cierto que:

$$\exists\ c, n_0 > 0\ t.q.\ \forall n \geq n_0,\ 0 \leq n^3 \log n \leq c \cdot n^3$$

Esto es:

$$n^3 \log n \leq c \cdot n^3$$

Dividiendo la relación anterior entre n^3 se tiene:

$$\log n \leq c$$

Nótese que esto no es necesariamente cierto para valores muy grandes de n, formalmente hablando, tómese el límite cuando $n \longrightarrow \infty$, entonces, la función $log(n)$ también tiende a infinito (aunque no de una manera rápida).

Así, tomando $n \longrightarrow \infty$ se tiene:

$$\infty \leq c$$

Esto quiere decir que se tendría un valor de c que también fuese infinito, lo cual no es posible. Así pues, el error se debe a la suposición inicial; $n^3 \log n \in O(n^3)$, entonces, se concluye que $n^3 \log n \notin O(n^3)$ ✓✓

4. para cualquier par de reales $a, b > 0$, $\log_a n \in \Theta(\log_b n)$.

Solución

Se sabe que:

$$\log_b(n) \leq \log_b(n) \leq \log_b(n)$$

Se multiplica por $\dfrac{1}{\log_a(n)}$ a las desigualdades anteriores.

$$\frac{1}{\log_a(n)} \cdot \log_b(n) \leq \frac{1}{\log_a(n)} \cdot \log_b(n) \leq \frac{1}{\log_a(n)} \cdot \log_b(n)$$

Luego:

$$\log_a(n) = \frac{1}{\log_a(n)} \cdot \log_b(n)$$

Por lo que:

$$\frac{1}{\log_a(n)} \cdot \log_b(n) \leq \log_a(n) \leq \frac{1}{\log_a(n)} \cdot \log_b(n)$$

Defínase $c_1 = c_2 = \dfrac{1}{\log_a(n)}$, entonces:

$$c_1 \cdot \log_b(n) \leq \log_a(n) \leq c_2 \cdot \log_b(n)$$

Es decir, se satisface lo escrito en 2.

Por lo tanto, es cierto que $\log_a(n) \in \Theta(\log_b n)$ ✓✓

Problema 5

Lee la sección *Asymptotic notation in equations and inequalities* del libro de algoritmos de Cormen et al., 4a. edición (capítulo 3, página 58), y explica por qué la siguiente expresión tiene sentido, y es correcta:

$$O(n^3 \log n) + 7n^2 + O(\log^2 n) \in O(n^3 \log n).$$

Solución:

Nótese que por definición,

$$f(n) \in O\big(g(n)\big) \text{ si y solo si } \exists\, c, n_0 > 0 \; t.q. \; \forall n \geq n_0,\; 0 \leq f(n) \leq c \cdot g(n)$$

Para el caso particular en donde $g(n) = n^3 \log n$, entonces existirá una función f_1 tal que cumple con la definición anterior; esto es, existe f_1 tal que:

$$\exists\, c, n_0 > 0 \; t.q. \; \forall n \geq n_0,\; 0 \leq f_1(n) \leq c \cdot n^3 \log n$$

De manera similar, y tomando la función $g(n) = \log^2 n$, se tiene un caso similar, solo que para una función f_2.

$$\exists\ c, n_0 > 0\ t.q.\ \forall n \geq n_0,\ 0 \leq f_2(n) \leq c \cdot \log^2 n$$

Ahora, con ayuda de las desigualdades anteriores se puede expresar la declaración inicial de la siguiente manera:

$$f_1(n) + 7n^2 + f_2(n) \in O(n^3 \log n).$$

Nótese que lo anterior es cierto pues particularmente $n^2 \in O(n^3 \log n)$, entonces, se tienen tres expresiones que cumplen con pertenecer al conjunto $O(n^3 \log n)$, lo que hace que la declaración inicial sea verdadera.

En resumen, el utilizar las expresiones $O(n^3 \log n)$ y $O(\log^2 n)$ es una manera de denotar funciones que están dentro de la propia familia, cumpliéndose la contención inicial.

✓✓