Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC1253 - Matemáticas Discretas

Tarea 5

17 de noviembre de 2021

2º semestre 2021 - Profesores M. Bucchi - G. Diéguez - F. Suárez

---

Requisitos

- La tarea es individual. Los casos de copia serán sancionados con la reprobación del curso con nota 1,1.
- **Entrega**: Hasta las 23:59:59 del 15 de noviembre a través del buzón habilitado en el sitio del curso (Canvas).
  - Esta tarea debe ser hecha completamente en LaTeX. Tareas hechas a mano o en otro procesador de texto **no serán corregidas**.
  - Debe usar el template LaTeX publicado en la página del curso.
  - Cada solución de cada problema debe comenzar en una nueva hoja. ***Hint:*** Utilice `\newpage`
  - Los archivos que debe entregar son el archivo `PDF` correspondiente a su solución con nombre `numalumno.pdf`, junto con un `zip` con nombre `numalumno.zip`, conteniendo el archivo `numalumno.tex` que compila su tarea. Si su código hace referencia a otros archivos, debe incluirlos también.
- El no cumplimiento de alguna de las reglas se penalizará con un descuento de 0.5 en la nota final (acumulables).
- No se aceptarán tareas atrasadas.
- Si tiene alguna duda, el foro de Canvas es el lugar oficial para realizarla.

Problemas

Problema 1

a) Demuestre que $\log(n!) \in \Theta(n \log(n))$.

b) Resuelva y encuentre una estimación O para la siguiente recurrencia:

$$T(n) = \begin{cases} 2 & n \leq 2 \\ T(\lfloor \sqrt{n} \rfloor) + 1 & n > 2 \end{cases}$$

Solución

a) Primero mostraremos que $\log(n!) \in O(n \log(n))$:

$$\begin{aligned} \log(n!) &= \log(1 \cdot 2 \cdot 3 \cdots n) \\ &\leq \log(n \cdot n \cdot n \cdots n) && \text{(para } n \geq 1) \\ &= \log(n^n) \\ &= n \log(n) \end{aligned}$$

Luego, con $n_0 = c = 1$, se cumple por definición que $\log(n!) \in O(n \log(n))$. Ahora hace falta demostrar que $\log(n!) \in \Omega(n \log(n))$:

$$\begin{aligned} \log(n!) &= \log(1 \cdot 2 \cdot 3 \cdots n) \\ &= \log(1) + \log(2) + \cdots + \log(n) \\ &= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \log(i) + \sum_{j=\lfloor \frac{n}{2} \rfloor + 1}^{n} \log(j) && \text{(separamos la sumatoria)} \\ &\geq \sum_{j=\lfloor \frac{n}{2} \rfloor + 1}^{n} \log(j) && \text{(nos quedamos con una mitad)} \\ &\geq \sum_{j=\lfloor \frac{n}{2} \rfloor + 1}^{n} \log(\frac{n}{2}) \\ &\geq \frac{n}{2} \log(\frac{n}{2}) \\ &\geq \frac{n}{2}(\log(n) - \log(2)) \\ &\geq \frac{n \log(n)}{4} && \text{para } n \geq 4 \end{aligned}$$

Luego, con $n_0 = 4$ y $c = \frac{1}{4}$, se cumple por definición que $\log(n!) \in \Omega(n \log(n))$, y por lo tanto podemos concluir que $\log(n!) \in \Theta(n \log(n))$.

b) Utilizaremos el reemplazo $n = 2^{2^k}$ con $k \in \mathbb{N}$.

Luego, $T(2^{2^k}) = T(2^{2^{k-1}}) + 1$, siguiendo esta ecuación de recurrencia resulta:

$$\begin{aligned} T(n) &= T(2^{2^{k-1}}) + 1 \\ T(n) &= T(2^{2^{k-2}}) + 2 \\ T(n) &= T(2^{2^{k-3}}) + 3 \\ &\vdots \\ T(n) &= T(2^{2^{k-i}}) + i \end{aligned}$$

si consideramos la iteración $i = k$:

$$T(n) = T(2^{2^{k-k}}) + k = T(2) + k = 2 + k$$

donde $k = \log_2(\log_2(n))$, luego podemos reescribir la recurrencia como:

$$T(n) = \log_2(\log_2(n)) + 2$$

Y por lo tanto concluimos que $T(n) \in O(\log(\log(n)) \mid \{2^{2^i} \mid i \in \mathbb{N}\})$. Ahora demostraremos por inducción que $T(n) \in O(\log(\log(n)))$. Para esto primero debemos estimar c y n_0:

$$\begin{aligned} T(1) &= 2 \\ T(2) &= 2 \\ T(3) &= T(1) + 2 = 4 \\ T(4) &= T(2) + 2 = 4 \\ &\vdots \\ T(8) &= T(3) + 2 = 6 \end{aligned}$$

Podríamos tomar $c = 7$ y $n_0 = 8$.

PD: $\forall n \geq 8$, $T(n) \leq 7 \cdot \log_2(\log_2(n))$. Por inducción fuerte:

BI: Además de $n = 8$, debemos mostrar la base para $3 \leq n \leq 7$, puesto que varios elementos sobre $T(8)$ dependen de estos resultados que no estamos mostrando.

$T(3) = 4 < 7 \cdot \log_2(\log_2(3))$

Para $4 \leq i \leq 7$:

$T(i) = 4 < 7 \cdot \log_2(\log_2(3)) < 7 \cdot \log_2(\log_2(i))$ (pues el logaritmo es creciente)
$T(8) = 6 < 14 = 7 \cdot \log_2(\log_2(8))$

HI: Supongamos que con $n \geq 9$, $\forall k \in \{3, \cdots, n-1\}$ se cumple que $T(k) \leq 7 \cdot \log_2(\log_2(k))$.

TI: Como $n \geq 9$:

$$
\begin{aligned}
T(n) &= T\left(\lfloor \sqrt{n} \rfloor\right) + 1 && \text{/ HI} \\
&\leq 7 \cdot \log_2(\log_2\left(\lfloor \sqrt{n} \rfloor\right)) + 1 && \text{/ log es creciente, sacamos el piso} \\
&\leq 7 \cdot \log_2(\log_2\left(\sqrt{n}\right)) + 1 \\
&= 7 \cdot \log_2(\log_2\left(n^{\frac{1}{2}}\right)) + 1 \\
&= 7 \cdot \log_2\left(\frac{\log_2(n)}{2}\right) + 1 \\
&= 7 \cdot (\log_2(\log_2(n)) - \log_2(2)) + 1 \\
&= 7 \cdot \log_2(\log_2(n)) - 6 \\
&< 7 \cdot \log_2(\log_2(n)) \square
\end{aligned}
$$

Pauta (6 pts.)

- 3 ptos por a) (1.5 pts si demuestra solo O o Ω).
- 1.5 ptos por desarrollar correctamente el remplazo de variable en $T(n)$.
- 1.5 ptos por extender el resultado a todo natural mediante inducción.

Puntajes parciales y soluciones alternativas a criterio del corrector.

Problema 2

Considere el siguiente algoritmo:

Precondiciones: un arreglo $A = [a_0, a_1, \ldots, a_{n-1}]$, un natural n (largo del arreglo) y un natural m.

Postcondiciones: el arreglo contiene una permutación de los valores originales tal que primero aparecen todos los elementos en A menores a m, seguidos de todos los elementos en A iguales a m, y terminando con todos los elementos en A mayores que m.

TierSort(A, n, m)

```
1:  i = 0
2:  j = 0
3:  k = n − 1
4:  while j <= k do
5:    if A[j] > m then
6:      aux = A[j]
7:      A[j] = A[k]
8:      A[k] = aux
9:      k−−
10:   else if A[j] = m then
11:     j++
12:   else
13:     aux = A[i]
14:     A[i] = A[j]
15:     A[j] = aux
16:     i++
17:     j++
18:   end if
19: end while
```

a) (5 ptos.) Demuestre que TierSort es correcto.

b) (1 pto.) Determine la complejidad de TierSort en el mejor y peor caso.

Solución

a) Para demostrar la correctitud del algoritmo, primero demostraremos que el algoritmo termina en exactamente n iteraciones, cuando $j = k + 1$:

Note que la condición de **término** del *loop* se puede reescribir como $k - j < 0$. Antes de la primera iteración $k - j = n - 1$. En cada iteración solo se puede ejecutar una de las líneas 9, 11 ó 17, lo cual implica que en cada iteración, o bien aumenta j en 1, o reduce k en 1. Esto se traduce a que $k - j$ reduce su valor en exactamente 1 durante cada

iteración, y por tanto que el loop alcanzará su condición de término cuando $k-j=-1$. Para esto, es necesario reducir el valor de $k-j$ en n, y por lo tanto concluimos que luego de n iteraciones se alcanzará la condición de término. Hemos demostrado entonces que el algoritmo termina, y además que cuando lo hace, $j=k+1$.

Ahora solo nos falta mostrar la correctitud parcial del algoritmo. Considere la siguiente afirmación:

Al comienzo de cualquier iteración, todos los elementos en $A[0:i]$ son menores que m, todos los elementos en $A[i:j]$ son exactamente iguales a m, y todos los elementos en $A[k+1:n]$ son mayores que m.

Demostraremos mediante inducción que la proposicón anterior es un invariante para el *loop* del algoritmo `TierSort`.

B.I. Antes de la primera iteración, las variables son inicializadas como $i=0$, $j=0$ y $k=n-1$. Luego, $A[0:i]$, $A[i:j]$ y $A[k+1:n]$ son todos arreglos vacíos. De esta forma, la propiedad cumple trivialmente.

H.I. Sean i_c, j_c y k_c los valores de las variables i, j y k respectivamente antes de iniciar la c-ésima iteración. Suponemos entonces que la propiedad enunciada se cumple para i_c, j_c y k_c.

T.I. Durante la iteración c puede pasar cualquiera de tres cosas:

- $A[j_c] > m$
  En este caso, se ejecutará el código entre las líneas 6 y 9. Luego, la única variable de interés modificada es k, la cual reduce en 1. Por **H.I.**, y dado que i y j no son modificadas durante la c-ésima iteración, podemos concluir que $A[0:i_{c+1}]$, $A[i_{c+1}:j_{c+1}]$ cumplen la propiedad enunciada. Dado que k reduce en 1 durante esta iteración, la **H.I.** nos permite concluir que $A[k_{c+1}+2:n]$ solo tiene elementos mayores que m, por lo que solo falta probar que $A[k_{c+1}+1] > m$. Notemos que antes de reducir el valor de k, se asigna $A[k]=A[j]$ (mediante la variable aux). En otras palabras $A[k_{c+1}+1]=A[j_c]$. Recordemos ademas, que $A[j_c] > m$, y por lo tanto podemos concluir que $A[k_{c+1}+1] > m$. Luego, la propiedad enunciada se mantiene para el comienzo de la siguiente iteración.

- $A[j_c] = m$
  En este caso, se ejecutará solo la línea 11. Luego, la única variable de interés modificada es j, la cual aumenta en 1. Por **H.I.** y dado que i y k no son modificadas durante la c-ésima iteración, podemos concluir que $A[0:i_{c+1}]$, $A[k_{c+1}+1:n]$ cumplen la propiedad enunciada. Dado que j aumenta en 1 durante esta iteración, la **H.I.** nos permite concluir que $A[j_{c+1}-1:n]$ tiene elementos iguales a m, por lo que solo falta probar que $A[j_{c+1}-1]=m$. Notemos que esto es lo mismo que probar que $A[j_c]=m$, lo cual sabemos es cierto. Luego, la propiedad enunciada se mantiene para el comienzo de la

siguiente iteración.

- $A[j_c] < m$
  En este caso, se ejecutará el código entre las línas 13 y 17. Luego, i y j son modificadas. Por **H.I.** y dado que k no es modificada durante la c-ésima iteración, podemos concluir que $A[k_{c+1}+1 : n]$ cumple la propiedad enunciada. Dado que i aumenta en 1 durante esta iteración, la **H.I.** nos permite concluir que $A[0 : i_{c+1}-1]$ tiene elementos iguales a m, por lo que solo falta probar que $A[i_{c+1}-1] < m$. Notemos que antes de aumentar el valor de i, se asigna $A[i] = A[j]$. En otras palabras $A[i_c] = A[j_c]$. Recordemos además que $A[j_c] < m$, y por lo tanto podemos concluir que $A[i_{c+1} - 1] < m$. Dado que j aumenta en 1 durante esta iteración, la **H.I.** nos permite concluir que $A[i_c : j_{c+1} - 1]$ tiene elementos iguales a m al comienzo de la iteración. Notemos que mediante la variable aux, se realiza la asignación $A[j_c] = A[j_i]$. Esto es lo mismo que decir $A[j_{c+1} - 1] = m$, lo cual nos permite concluir entonces que $A[i_{c+1} : j_{c+1}]$ tiene solo valores iguales a m. Luego, la propiedad enunciada se mantiene para el comienzo de la siguiente iteración.

Por inducción entonces se ha demostrado que después de cada iteración (o antes de partir una siguiente) se cumple que todos los elementos en $A[0 : i]$ son menores que m, todos los elementos en $A[i : j]$ son exactamente iguales a m, y todos los elementos en $A[k + 1 : n]$ son mayores que m. Por último, recordemos que la condición de término del loop se alcanza cuando $j = k + 1$ y por lo tanto, las divisiones anteriores son exhaustivas y cubren exactamente todo el arreglo

Concluimos entonces que el algoritmo es correcto y al terminar se cumplen las post condiciones.

b) En primer lugar, notemos que el algoritmo es iterativo, así que podemos abordarlo directamente contando instrucciones. Contaremos la cantidad de comparaciones que realiza el algoritmo para un arreglo de tamaño n.

Notemos que hay solo tres comparaciones dentro del algoritmo: en las líneas 4, 5 y 10. Todas forman pate del *while loop* principal del algoritmo.

En el *mejor* caso, dentro del *loop* solo se ejecutará la comparación de la línea 5. Un ejemplo de esto, es cuando el arreglo de entrada está formado solo por enteros menores que m. Por otra parte, en el *peor* caso, siempre se ejecutarán las dos comparaciones dentro del *loop*. Un ejemplo de esto es un arreglo que está formado solo por enteros mayores o iguales a m.

Recordemos, de la parte (a), que el cuerpo del *loop* se ejecuta exactamente n veces. Dado esto, la línea 4 se ejecutará $n + 1$ veces: 1 por cada iteración, más la verificación que romperá el *loop*. Similarmente, la línea 5 se ejecutará exactamente n veces en cualquier caso. La línea 10 no se ejecutará nunca en el mejor caso, mientras que en el peor caso se ejecutará exactamente n veces (una vez por iteración).

Finalmente, podemos concluir que en la ejecución del algoritmo `TierSort` con un arreglo de tamaño n, se realizan $2n + 1$ comparaciones en el mejor caso, mientras que en el peor caso, se realizan $3n + 1$ comparaciones.

Por lo tanto, concluimos que el algoritmo tiene complejidad $\Theta(n)$.

Pauta (6 pts.)

- 1 pto. por demostrar que el algoritmo termina.
- 1 pto. por enunciar el invariante
- 3 ptos por demostrar que el invariante es correcto usando inducción.
- 0.5 ptos por demostrar correctamente la complejidad para cada uno de los casos (mejor y peor).

Puntajes parciales y soluciones alternativas a criterio del corrector.