Pregunta 1: Justifique si son verdaderos o falsos:

1. $n^3 \in O(n^3 \log n)$
2. $\log n \in O(\sqrt{n})$
3. $n^2 \in \omega(n \log n)$
4. $n^{\log n} \in O(n^2)$
5. $2^n \in o(3^n)$

Pregunta 2:

- Describa lo que hace el algoritmo y analice la complejidad temporal y espacial del algoritmo
- Demuestre la correctitud del algoritmo

Algorithm 1 Quickselect

```
1:  procedure QUICKSELECT(A, low, high, k)
2:      if low = high then
3:          return A[low]
4:      end if
5:      pivot_index ← Partition(A, low, high)
6:      if k = pivot_index then
7:          return A[k]
8:      else if k < pivot_index then
9:          return QUICKSELECT(A, low, pivot_index − 1, k)
10:     else
11:         return QUICKSELECT(A, pivot_index + 1, high, k)
12:     end if
13: end procedure
```

Algorithm 2 Partition

```
1:  procedure PARTITION(A, low, high)
2:      pivot ← A[high]
3:      i ← low − 1
4:      for j ← low to high − 1 do
5:          if A[j] ≤ pivot then
6:              i ← i + 1
7:              SWAP(A[i], A[j])
8:          end if
9:      end for
10:     SWAP(A[i + 1], A[high])
11:     return i + 1
12: end procedure
```

Pregunta 3: Analizaremos 2 algoritmos que resuelven el problema de la suma maxima de un subarreglo. Este problema consiste en dado un arreglo de números enteros `A[]`, se debe encontrar el subarreglo (una secuencia continua de elementos) que tenga la suma más alta y devolver dicha suma.

- Analice la complejidad temporal del algoritmo **Maxsubsum** y del algoritmo de **Kadane**
- Demuestre la correctitud del algoritmo de **Kadane**

Algorithm 3 Maxsubsum

```
1: procedure MaxSubarraySum(A, left, right)
2:     if left = right then
3:         return A[left]
4:     end if
5:     mid ← ⌊(left + right)/2⌋
6:     left_max ← MaxSubarraySum(A, left, mid)
7:     right_max ← MaxSubarraySum(A, mid + 1, right)
8:     left_sum ← −∞
9:     total ← 0
10:    for i ← mid downto left do
11:        total ← total + A[i]
12:        if total > left_sum then
13:            left_sum ← total
14:        end if
15:    end for
16:    right_sum ← −∞
17:    total ← 0
18:    for i ← mid + 1 to right do
19:        total ← total + A[i]
20:        if total > right_sum then
21:            right_sum ← total
22:        end if
23:    end for
24:    crossing_max ← left_sum + right_sum
25:    return máx(left_max, right_max, crossing_max)
26: end procedure
```

Algorithm 4 Kadane

```
1: procedure Kadane(A, n)
2:     current_max ← A[0]
3:     global_max ← A[0]
4:     for i ← 1 to n − 1 do
5:         current_max ← máx(A[i], current_max + A[i])
6:         if current_max > global_max then
7:             global_max ← current_max
8:         end if
9:     end for
10:    return global_max
11: end procedure
```

Solución 1

1. $n^3 \in O(n^3 \log n)$

   - **Método de Desigualdades:**
     Queremos demostrar que existe una constante c y un n_0 tal que para todo $n \geq n_0$:

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

     Sabemos que $\log n \geq 1$ para $n \geq 1$. Esto implica que $n^3 \leq n^3 \log n$. Por lo tanto, podemos tomar $c = 1$ y $n_0 = 1$.
     Conclusión: Verdadero, $n^3 \in O(n^3 \log n)$.
   - **Método de Límites:**
     Calculamos el límite:

     $$\lim_{n\to\infty} \frac{n^3}{n^3 \log n} = \lim_{n\to\infty} \frac{1}{\log n}.$$

     Como $\log n \to \infty$ cuando $n \to \infty$, el límite tiende a 0. Esto indica que $n^3 \in O(n^3 \log n)$.
     Conclusión: Verdadero, $n^3 \in O(n^3 \log n)$.

2. $\log n \in O(\sqrt{n})$

   - **Método de Desigualdades:**
     Queremos encontrar una constante c y un n_0 tal que para todo $n \geq n_0$:

     $$\log n \leq c \cdot \sqrt{n}.$$

     Sabemos que $\log n$ crece más lentamente que $\sqrt{n}$ para $n \geq 1$. Por lo tanto, podemos elegir $c = 1$ y $n_0 = 1$.
     Conclusión: Verdadero, $\log n \in O(\sqrt{n})$.
   - **Método de Límites:**
     Calculamos el límite:

     $$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}}.$$

     Usamos L'Hôpital:

     $$\lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = \lim_{n\to\infty} \frac{2\sqrt{n}}{n} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0.$$

     Como el límite es 0, $\log n \in O(\sqrt{n})$.
     Conclusión: Verdadero, $\log n \in O(\sqrt{n})$.

3. $n^2 \in \omega(n \log n)$

   - **Método de Desigualdades:**
     Queremos demostrar que para cualquier constante c y algún n_0, se cumple:

     $$n^2 > c \cdot n \log n \quad \text{para todo } n \geq n_0.$$

     Sabemos que n crece más rápido que $\log n$, para $n > 0$, entonces:

     $$n > \log n$$
     $$n^2 > n \log n$$

     Por lo que esta desigualdad se cumple para $n_0 = 1$ y $c = 1$.
     Conclusión: Verdadero, $n^2 \in \omega(n \log n)$.

- **Método de Límites:**
  Calculamos el límite:
  $$\lim_{n\to\infty} \frac{n^2}{n\log n} = \lim_{n\to\infty} \frac{n}{\log n}.$$
  Como este límite tiende a ∞, $n^2 \in \omega(n\log n)$.
  Conclusión: Verdadero, $n^2 \in \omega(n\log n)$.

4. $n^{\log n} \in O(n^2)$

   - **Método de Desigualdades:**
     Queremos demostrar que NO existe un c, n_0 tal que:
     $$n^{\log n} \leq c \cdot n^2, \quad n \geq n_0 \tag{1}$$
     Sea $c \in \mathbb{R}$, notemos que para $n \geq c^2$
     $$\log n \geq 2\log c$$
     $$n^{\log n} \geq c \cdot n^2$$
     Por lo tanto, para cualquier c no existe un n_0 tal que se cumpla (1)
     Conclusión: Falso, $n^{\log n} \notin O(n^2)$.
   - **Método de Límites:**
     Calculamos el límite:
     $$\lim_{n\to\infty} \frac{n^{\log(n)}}{n^2} = \lim_{n\to\infty} n^{\log(n)-2}.$$
     Como $\log(n) - 2$ es una funcion crecciente y positiva (para $n > 10^2$), y n tambien es creciente, el limite tiende a ∞.
     Conclusión: Falso, $n^{\log n} \notin O(n^2)$.

5. $2^n \in o(3^n)$

   - **Método de Desigualdades:**
     Queremos demostrar que existe un c, n_0, tal que se cumple:
     $$2^n < c \cdot 3^n \quad \text{para todo } n \geq n_0.$$
     Dividiendo ambos lados por 3^n, obtenemos:
     $$\left(\frac{2}{3}\right)^n < c.$$
     Como $\left(\frac{2}{3}\right)^n \leq 1$ cuando $n \geq 1$, la desigualdad se cumple para cualquier c positivo. Por lo tanto la desigualdad se cumple para $c = 1$ y $n_0 = 1$
     Conclusión: Verdadero, $2^n \in o(3^n)$.
   - **Método de Límites:**
     Calculamos el límite:
     $$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0.$$
     Como el límite es 0, $2^n \in o(3^n)$.
     Conclusión: Verdadero, $2^n \in o(3^n)$.

Solución 2

Descripción

El algoritmo `Quickselect` encuentra el k-ésimo elemento más pequeño en una lista desordenada. Se basa en el principio de dividir y conquistar, dividiendo el arreglo de entrada en dos sublistas.

- El algoritmo selecciona un pivote a través del algoritmo `Partition`, y lo utiliza para dividir el arreglo en dos partes: los elementos menores que el pivote y los elementos mayores.
- Dependiendo de la posición del pivote, el algoritmo decide si buscar en la parte izquierda o derecha del pivote, o si ha encontrado el k-ésimo elemento.
- Este proceso se repite recursivamente hasta que se encuentra el k-ésimo elemento.

Análisis de la complejidad

- Complejidad temporal: `Quickselect` es un algoritmo de dividir y conquistar, el cual divide el problema en dos partes. Dependiendo de la posición del k-ésimo elemento respecto al pivote, solo una de estas partes se considera en la siguiente llamada recursiva. Esto lleva a la siguiente relación de recurrencia:

  $$T(n) = T(n') + O(n)$$

  Donde $O(n)$ representa el costo del algoritmo `Partition`, y n' es el tamaño del subarreglo en el que se continúa la búsqueda, este depende de la posición del pivote:

  - En el mejor caso $n' \sim \frac{n}{2}$
  - En el peor caso es cuando el pivote está cerca de un extremo, es decir, $n' \sim n - 1$

  Ahora queremos obtener el caso promedio. Al seleccionar siempre como pivote al último elemento del arreglo desordenado, el elemento seleccionado puede tomar cualquier posicion una vez ordenado, con igual probabilidad. Dado un arreglo de n elementos, la posición p del pivote en el arreglo ordenado se considera una variable aleatoria que puede tomar valores de 0 a $n - 1$.

  La esperanza $E[p]$ de la posición del pivote se calcula como:

  $$E[p] = \sum_{i=0}^{n-1} i \cdot P(i),$$

  donde $P(i) = \frac{1}{n}$ es la probabilidad de que el elemento en la posición i sea elegido como pivote. Por lo tanto, la fórmula se convierte en:

  $$E[p] = \frac{1}{n} \sum_{i=0}^{n-1} i = \frac{1}{n} \cdot \frac{(n-1)n}{2} = \frac{n-1}{2} \approx \frac{n}{2}.$$

  Esto implica que, en promedio, el pivote divide el arreglo en dos partes de tamaños aproximadamente iguales. Entonces se tiene:

  $$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

  Luego, según el teorema maestro, con $a = 1$, $b = 2$ y $d = 1$. Comparamos d con $\log_b a = \log_2 1 = 0$. Nos encontramos en el tercer caso, entonces:

  $$T(n) = O(n)$$

- Complejidad espacial: La complejidad espacial (adicional) del algoritmo Quickselect depende del lenguaje en el que se implemente. Dentro de cada llamada de Quickselect, se utiliza $O(1)$ variables adicionales antes de retornar. En el peor caso el algoritmo debe realizar $O(n)$ llamadas recursivas, sin embargo, estas llamas son lo último que realiza antes de retornar, por lo que no requiere mantener en memoria las $O(1)$ variables del nivel de recursión actual.

  - Caso 1: El lenguaje libera la memoria del nivel de recursión anterior: El algoritmo utiliza $O(1)$ memoria adicional.
  - Caso 2: El lenguaje no libera la memoria del nivel de recursión anterior: El algoritmo utiliza $O(n)$ memoria adicional.

  Tip adicional. Es posible implementar la funcion sin recursión, utilizando un loop. Haciendo esto forzamos que la memoria adicional sea $O(1)$.

Demostración de correctitud

Para demostrar la correctitud del algoritmo Quickselect, definimos el siguiente invariante: Después de cada partición, si el subarreglo está dividido por un pivote en la posición p, todos los elementos a la izquierda del pivote son menores o iguales al pivote, y todos los elementos a la derecha del pivote son mayores al pivote. Luego, queremos demostrar que este invariante se mantiene a lo largo de todas las iteraciones, y que al finalizar el ciclo, se retorna el k-ésimo elemento más pequeño de la lista

1. Inicialización: Al comienzo tenemos el arreglo completo y no hemos hecho ninguna partición. Como no hemos separado aún los elementos, el invariante se cumple.

2. Mantenimiento: Para demostrar que el invariante se cumple durante el ciclo, realizamos inducción sobre el tamaño del subarreglo n.

   - Paso base: cuando $n = 1$, entonces el sub arreglo tiene solo un elemento. Si $low = high$, el algoritmo retorna el único elemento del arreglo. En este caso el invariante se cumple ya que no hay otros elementos en el arreglo.
   - Paso inductivo: Suponemos que el invariante es cierto para todos los subarreglos de tamaño menor que n, es decir, los elementos a la derecha son mayores y a su izquierda son menores.

     Ahora queremos demostrar que se cumple para un subarreglo de tamaño n. Para esto, sabemos que después de la partición, el pivote está en la posición correcta en el arreglo. Luego, tenemos los 3 casos:

     a) Si $k = pivot\_index$, el algoritmo retorna el pivote, que es el k-ésimo menor elemento, ya que todos los elementos a su izquierda son menores y los elementos a su derecha son mayores.

     b) Si $k < pivot\_index$, el algoritmo recursivamente busca en la sublista izquierda, que contiene los elementos menores al pivote. Dado que el tamaño de esta sublista es menor que n, la hipótesis de inducción se aplica, garantizando que la búsqueda encontrará el k-ésimo menor elemento en esta parte.

     c) Si $k > pivot\_index$, busca en la sublista derecha, que contiene los elementos mayores al pivote. De nuevo, la hipótesis de inducción se aplica para asegurar que se encontrará el k-ésimo menor elemento.

     Por lo tanto, se garantiza que el invariante se cumple a lo largo del ciclo.

3. Terminación: El algoritmo termina cuando el índice del pivote p coincide con el valor k. Cuando el índice del pivote es igual a k, el invariante garantiza que todos los elementos a la izquierda del pivote son menores o iguales, y todos los elementos a la derecha son mayores. Por lo tanto, el pivote es exactamente el k-ésimo elemento más pequeño.

Solución 3

Análisis de Complejidad Temporal de Maxsubsum

El algoritmo **Maxsubsum** es un algoritmo de dividir y conquistar que divide el problema en dos mitades y luego calcula la suma máxima cruzando el punto medio. Esto lleva a la siguiente relación de recurrencia:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Donde:

- $2T\left(\frac{n}{2}\right)$ es el costo de resolver los dos subproblemas de tamaño $n/2$.
- $O(n)$ es el costo de combinar los resultados al encontrar la suma máxima cruzando el punto medio.

Podemos aplicar el Teorema Maestro para resolver la relación de recurrencia. El teorema se aplica a las recurrencias de la forma:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Comparando la relación de recurrencia del algoritmo **Maxsubsum** con esta forma, identificamos los siguientes parámetros:

- $a = 2$: El número de subproblemas es 2.
- $b = 2$: Cada subproblema tiene un tamaño de $n/2$.
- $d = 1$: El costo no recursivo es lineal, es decir, $O(n)$.

Ahora, calculamos $n^{\log_b a}$:

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Esto corresponde al **caso 2 del Teorema Maestro**, donde $n^d = n^{\log_b a}$. En este caso, la complejidad es:

$$T(n) = O(n \log n)$$

Demostración de Correctitud del Algoritmo Kadane

Invariante de ciclo

El invariante de ciclo en el algoritmo de **Kadane** es el siguiente:
Al comienzo de cada iteración del ciclo en la línea 4, para cada índice i:

- `current_max` contiene la suma máxima de cualquier subarreglo que termina en el índice $i - 1$.
- `global_max` contiene la suma máxima de cualquier subarreglo que se ha encontrado hasta el índice $i - 1$.

Objetivo: Mostrar que este invariante se mantiene a lo largo de todas las iteraciones del ciclo, y que al finalizar el ciclo, `global_max` contendrá la suma máxima del subarreglo para todo el arreglo A.

Prueba del Invariante de ciclo

1. **Inicialización:** Antes de la primera iteración del ciclo, es decir, cuando $i = 1$:

   - `current_max` se inicializa como $A[0]$. Esto es correcto porque el único subarreglo posible que termina en el índice 0 es el subarreglo que contiene únicamente el elemento $A[0]$, cuya suma es $A[0]$.
   - `global_max` también se inicializa como $A[0]$. Esto es correcto porque, después de procesar el primer elemento, el subarreglo máximo encontrado es precisamente $A[0]$.

   Por lo tanto, el invariante de ciclo es cierto al inicio.

2. **Mantenimiento:** Supongamos que el invariante de ciclo es cierto al comienzo de la i-ésima iteración. Ahora demostraremos que también es cierto al comienzo de la siguiente iteración $(i + 1)$.

   a) En la línea 5, se actualiza `current_max` con el valor máx$(A[i],$ `current_max` $+ A[i])$. Esto garantiza que `current_max` contenga la suma máxima de cualquier subarreglo que termine en el índice i, ya que hay dos posibilidades:

      1) El subarreglo óptimo que termina en i incluye solo el elemento $A[i]$, en cuyo caso, la maxima suma del mejor subarreglo que termina en i es $A[i]$.
      2) El subarreglo óptimo que termina en i extiende el subarreglo máximo que termina en $i - 1$, en cuyo caso, la maxima suma del subarreglo que termina en i es `current_max` $+ A[i]$.

   b) En la línea 6, si el nuevo valor de `current_max` es mayor que el valor actual de `global_max`, entonces actualizamos `global_max`. Esto asegura que `global_max` siempre contenga la suma máxima de cualquier subarreglo encontrado hasta el índice i.

   Por lo tanto, el invariante de ciclo se mantiene al final de cada iteración.

3. **Terminación:** Al finalizar el ciclo (es decir, después de procesar todos los elementos del arreglo A), el valor de i será n. En este punto, `global_max` contiene la suma máxima de cualquier subarreglo de A, ya que hemos verificado cada subarreglo posible que termina en cada índice y mantenido el valor máximo en `global_max`.