

Padrón:

Apellido y Nombre:

Correo electronico:

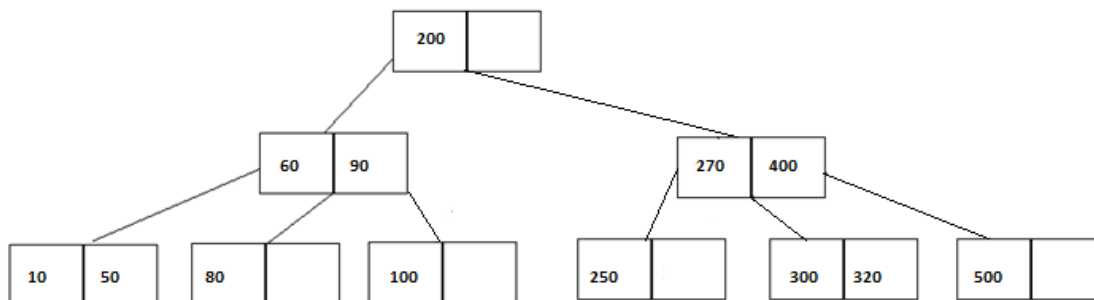
1) Conceptos básicos de complejidad

Colocar V o F, justificando (la justificación es necesaria para la puntuación del ítem)

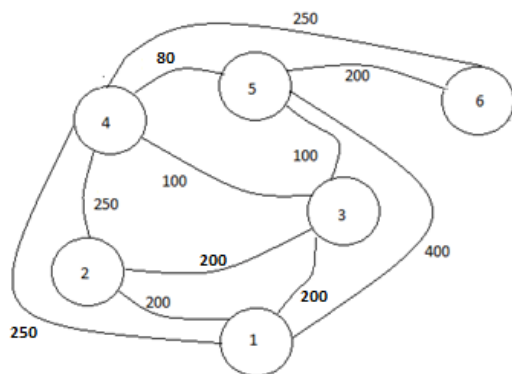
Afirmación	Indicar V o F
$\Theta(f(n)) \subset \Theta(g(n)) \Rightarrow \Omega(g(n)) \subset \Omega(f(n))$	
Todo algoritmo $\Omega(n * \log n)$ pertenece a $O(n)$	
Si $T(n) = T(n-1) + 1$ y $T(1) = 0$ entonces $T(n)$ pertenece a $O(n^2)$	
El reordenamiento de los datos de un array para que verifique las condiciones de heap de mínimo sobre un array no puede hacerse (para el peor caso) con un coste inferior a $O(n * \log n)$	
Si $T(n) = 3 * T(n/2) + n^2$, siendo $T(1)=1$, entonces $T(n)$ pertenece a $O(n^3)$	

2) TDA Conjunto:

- a) Diseñar un algoritmo que permita determinar cuántas hojas tiene un ABB. Indicar eficiencia
- b) Considere esta secuencia de datos: 10, 25, 35, 30, 2, 1. Muestre gráficamente cómo quedan almacenados: en un árbol heap de máximo si se incorporan de a uno. Definir árbol heap e indicar usos.
- c) En el siguiente árbol B, realizar gráficamente de forma sucesiva sucesivamente el alta de 25, y luego la baja de 500 y 250

**3) TDA Grafo:**

a)



En el grafo de la izquierda

- a) Obtenga el árbol de expansión de coste mínimo. Describa el algoritmo utilizado con detalle. Indique cuáles son las estructuras usadas (para implementar el grafo y/o adicionales requeridas por el algoritmo usado). En particular, indique cómo organiza los candidatos el algoritmo que describió

- b) Explicar qué es un punto de articulación o vértice de corte. Dar un ejemplo

4) Estrategias de resolución de problemas: Caracterice y ejemplifique la estrategia "Divide y Vencerás"

Solución:

Punto 1.A

La afirmación dada es:

$$\Theta(f(n)) \subset \Theta(g(n)) \implies \Omega(g(n)) \subset \Omega(f(n))$$

Esta afirmación es falsa. Para demostrarlo, proporcionaré un contraejemplo.

Contraejemplo:

Considera $f(n) = n^2$ y $g(n) = n^3$. Aquí, tenemos que:

$$\Theta(f(n)) = \Theta(n^2)$$

$$\Theta(g(n)) = \Theta(n^3)$$

Ahora, la afirmación inicial $\Theta(f(n)) \subset \Theta(g(n))$ es verdadera, ya que n^2 crece más lentamente que n^3 .

Sin embargo, al considerar Ω , tenemos:

$$\Omega(g(n)) = \Omega(n^3)$$

$$\Omega(f(n)) = \Omega(n^2)$$

En este caso, $\Omega(g(n)) \not\subset \Omega(f(n))$ ya que n^3 no es asintóticamente inferior a n^2 .

Por lo tanto, la afirmación dada es falsa, y el contraejemplo muestra que hay casos en los que $\Theta(f(n)) \subset \Theta(g(n))$ pero $\Omega(g(n)) \not\subset \Omega(f(n))$.

Para demostrar que $f(n) = n^2$ pertenece a $\Theta(g(n) = n^3)$, necesitamos encontrar constantes c_1, c_2 , y n_0 que cumplan con la definición de la notación Θ .

Primero, consideramos la parte inferior de la definición:

$$c_1 \cdot n^3 \leq n^2$$

Para simplificar, podemos dividir ambos lados por n^2 :

$$c_1 \cdot n \leq 1$$

Podemos elegir $c_1 = 1$ y cualquier n_0 ya que la desigualdad es cierta para cualquier $n \geq 1$.

Ahora, consideramos la parte superior de la definición:

$$c_2 \cdot n^3 \geq n^2$$

Dividimos ambos lados por n^2 :

$$c_2 \cdot n \geq 1$$

Podemos elegir $c_2 = 1$ y cualquier n_0 ya que la desigualdad es cierta para cualquier $n \geq 1$.

Entonces, en este caso, $c_1 = 1, c_2 = 1$, y $n_0 = 1$ cumplen con la definición de Θ . En otras palabras, n^2 está en $\Theta(n^3)$.

Sí, eso es correcto. La notación $\Theta(f(n))$ indica que la función $f(n)$ está acotada superior e inferiormente por ciertas constantes multiplicativas y funciones.

Formalmente, una función $g(n)$ pertenece a $\Theta(f(n))$ si existen constantes positivas c_1, c_2 , y n_0 tales que:

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n) \quad \text{para todo } n \geq n_0$$

Esto significa que $g(n)$ está limitada tanto superior como inferiormente por $f(n)$ multiplicado por constantes. En otras palabras, $f(n)$ y $g(n)$ tienen el mismo crecimiento asintótico.

Por lo tanto, cuando se dice que $\Theta(f(n)) \subset \Theta(g(n))$, se está indicando que la función $f(n)$ está acotada superior e inferiormente por ciertas constantes multiplicativas y funciones, y estas cotas también son válidas para $g(n)$, lo que implica que $g(n)$ también tiene el mismo crecimiento asintótico que $f(n)$.

Punto 1.B

La afirmación dada es:

$$\Omega(n \log n) \subseteq O(n)$$

Esta afirmación es falsa. La notación Ω implica un límite inferior asintótico, mientras que la notación O implica un límite superior asintótico. En otras palabras, $\Omega(n \log n)$ indica que la función crece al menos tan rápido como $n \log n$, mientras que $O(n)$ indica que la función está acotada superiormente por n .

Dado que $\Omega(n \log n)$ implica un crecimiento más rápido que lineal ($O(n)$), la afirmación no es cierta en general. Hay algoritmos cuya complejidad es $\Omega(n \log n)$ pero no son $O(n)$. Por ejemplo, algoritmos de ordenación eficientes como el algoritmo Merge Sort o el algoritmo Heap Sort tienen una complejidad de $\Omega(n \log n)$, pero no son lineales.

En resumen, la afirmación dada es falsa en general. No todos los algoritmos con complejidad $\Omega(n \log n)$ pertenecen a $O(n)$.

Punto 1.C

Para analizar si $T(n) = T(n - 1) + 1$ con $T(1) = 0$ pertenece a $O(n^2)$, intentemos encontrar una cota superior en términos de n^2 .

La ecuación de recurrencia $T(n) = T(n - 1) + 1$ indica que en cada paso se suma 1 al valor anterior de $T(n)$. Dado que $T(1) = 0$, podemos expresar $T(n)$ de la siguiente manera:

$$T(n) = T(n - 1) + 1 = T(n - 2) + 2 = \dots = T(1) + (n - 1) = n - 1$$

Ahora, veamos si $T(n)$ puede estar acotada superiormente por $O(n^2)$. Supongamos que existe una constante positiva c tal que $T(n) \leq c \cdot n^2$.

$$n - 1 \leq c \cdot n^2$$

Dividamos ambos lados por n :

$$1 - \frac{1}{n} \leq c \cdot n$$

A medida que n tiende a infinito, el término $\frac{1}{n}$ se vuelve insignificante, y la desigualdad se reduce a $1 \leq c \cdot n$.

Para que esta desigualdad sea cierta para todos los n suficientemente grandes, c debe ser mayor o igual a 1.

En resumen, $T(n) = n - 1$ no pertenece a $O(n^2)$. Más bien, $T(n)$ pertenece a $O(n)$ ya que es lineal con respecto a n , y la constante c puede ser elegida como 1 en este caso.

Entonces, la afirmación original de que $T(n) = T(n - 1) + 1$ con $T(1) = 0$ pertenece a $O(n^2)$ es falsa, pero sí pertenece a $O(n)$.

Punto 1.D

La afirmación que has presentado es correcta. El reordenamiento de los datos de un array para que cumpla con las condiciones de un heap de mínimo (o heap binario de mínimo) tiene un límite inferior de complejidad de $O(n \log n)$ para el peor caso.

En un heap de mínimo, para cada nodo i distinto de la raíz, se debe cumplir que el valor almacenado en el nodo i sea mayor o igual al valor almacenado en el padre de i . Dado que los elementos en el array no cumplen inicialmente esta propiedad, es necesario reorganizar el array mediante operaciones de intercambio (swap) para establecer la estructura de heap de mínimo.

La construcción de un heap de mínimo se puede hacer de manera eficiente utilizando un algoritmo como el algoritmo de "Heapify" o el algoritmo "Bottom-Up Heap Construction", ambos con una complejidad de $O(n)$. Sin embargo, cuando se trata de una secuencia arbitraria de datos, el límite inferior para cualquier algoritmo de ordenamiento basado en comparaciones es $O(n \log n)$ según el teorema de la información de decisión.

Por lo tanto, en el peor caso, no se puede reordenar un array para cumplir con las condiciones de un heap de mínimo en un tiempo inferior a $O(n \log n)$.

Punto 1.E

Para analizar la complejidad asintótica de $T(n) = 3 \cdot T\left(\frac{n}{2}\right) + n^2$ con $T(1) = 1$, podemos utilizar el método de sustitución o el teorema maestro.

El teorema maestro se aplica a recurrencias de la forma $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, donde $a \geq 1, b > 1$, y $f(n)$ es una función asintóticamente positiva. En este caso, $a = 3, b = 2$, y $f(n) = n^2$.

El teorema maestro tiene tres casos:

1. Si $f(n) = O(n^{\log_b a - \epsilon})$ para alguna constante positiva ϵ , entonces $T(n) = \Theta(n^{\log_b a})$.
2. Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.
3. Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguna constante positiva ϵ , y $a \cdot f\left(\frac{n}{b}\right) \leq k f(n)$ para alguna constante $k < 1$ y n suficientemente grande, entonces $T(n) = \Theta(f(n))$.


En este caso, $a = 3, b = 2$, y $f(n) = n^2$. La función n^2 no satisface ninguno de los casos 1 y 3, pero sí se ajusta al caso 2. Por lo tanto, podemos decir que $T(n) = \Theta(n^{\log_2 3} \cdot \log n)$.

Dado que $\log_2 3 < 2$, podemos decir que $T(n) = O(n^2)$. Por lo tanto, la afirmación es cierta, y podemos concluir que $T(n)$ pertenece a $O(n^3)$.

Punto 2.A

Para determinar cuántas hojas tiene un Árbol Binario de Búsqueda (ABB), puedes hacer un recorrido en profundidad (DFS) a través del árbol y contar las hojas. Aquí te doy un ejemplo de un algoritmo en pseudocódigo:

plaintext

 Copy code

```
Función ContarHojas(nodo):  
    Si nodo es nulo:  
        Devolver 0  
    Si nodo no tiene hijos (nodo es una hoja):  
        Devolver 1  
    Sino:  
        ContarHojasIzquierda = ContarHojas(nodo.hijoIzquierdo)  
        ContarHojasDerecha = ContarHojas(nodo.hijoDerecho)  
        Devolver ContarHojasIzquierda + ContarHojasDerecha
```

Este algoritmo cuenta las hojas recursivamente. La eficiencia de este algoritmo está determinada por el número total de nodos en el árbol, ya que cada nodo se visita exactamente una vez. Si el árbol tiene n nodos, entonces la complejidad de tiempo del algoritmo es $O(n)$, ya que cada nodo se procesa una vez.

Ahora, vamos a asumir que el ABB está equilibrado para obtener una cota superior en términos de altura del árbol. En un ABB equilibrado, la altura es $\log n$, donde n es el número de nodos en el árbol. Por lo tanto, la complejidad de espacio es $O(\log n)$ debido a la recursión en la pila de llamadas.

En resumen, la eficiencia del algoritmo para contar las hojas de un ABB es $O(n)$ en términos de complejidad de tiempo y $O(\log n)$ en términos de complejidad de espacio (bajo la asunción de que el árbol está equilibrado).


Punto 2.B

Un árbol heap es una estructura de datos de árbol binario que satisface la propiedad de heap. En un heap de máximo, para cada nodo i , el valor almacenado en i es mayor o igual al valor almacenado en sus hijos.

Vamos a mostrar gráficamente cómo quedan almacenados los datos dados (10, 25, 35, 30, 2, 1) en un árbol heap de máximo cuando se incorporan de a uno:

1. Incorporar 10:


markdown

 Copy code

```
10
```

2. Incorporar 25:


markdown

 Copy code

```
  25
 /
10
```

3. Incorporar 35:

markdown

 Copy code

```
  35
 /  \
10   25
```

4. Incorporar 30:



5. Incorporar 2:



6. Incorporar 1:



Este árbol cumple con la propiedad de heap de máximo: el valor en cada nodo es mayor o igual que los valores en sus hijos.

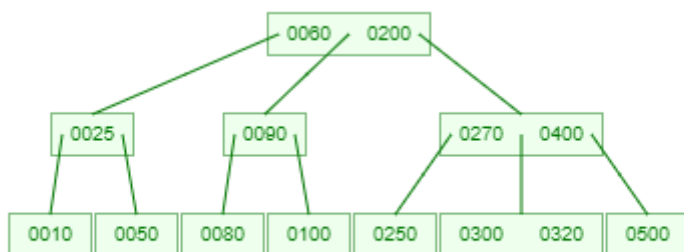
Usos de un árbol heap de máximo:

- **Cola de prioridad:** Los árboles heap se utilizan comúnmente en implementaciones de colas de prioridad, donde el nodo de mayor valor (o prioridad) se encuentra en la raíz.
- **Algoritmos de ordenación:** Los heaps se utilizan en algoritmos de ordenación eficientes como Heap Sort.
- **Implementación de colas de prioridad en algoritmos de búsqueda:** Por ejemplo, en el algoritmo de Dijkstra para encontrar el camino más corto en un grafo.

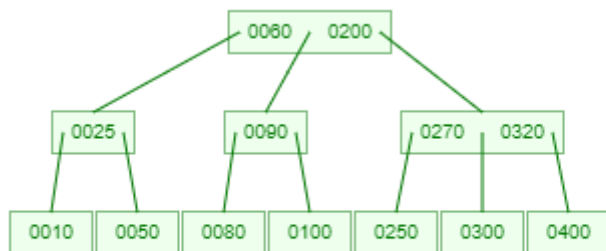
En resumen, un árbol heap es una estructura de datos importante que se utiliza en diversos algoritmos y aplicaciones, especialmente cuando es necesario mantener y acceder eficientemente al elemento de mayor valor. En el caso del árbol heap de máximo, este valor se encuentra en la raíz del árbol.

Punto 2.C

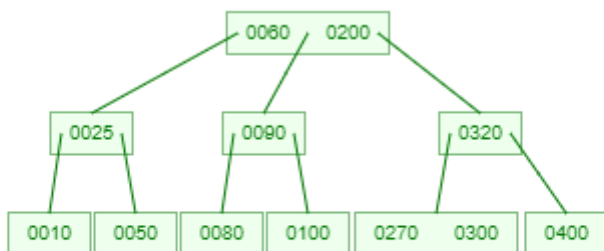
Inserto el 25



Elimino el 500



Elimino el 250



Punto 3.B

Un punto de articulación, también conocido como vértice de corte, es un vértice en un grafo cuya eliminación aumenta el número de componentes conectados del grafo. En otras palabras, la eliminación de un punto de articulación desconecta el grafo o lo divide en dos o más componentes conectados.

Ejemplo:

Consideremos el siguiente grafo no dirigido:



En este grafo, los vértices son A, B, C, D y E. Ahora, supongamos que eliminamos el vértice B:



Después de eliminar el vértice B, el grafo se divide en dos componentes conectados: {A, C} y {D, E}. Por lo tanto, el vértice B es un punto de articulación en este grafo, ya que su eliminación cambia la conectividad del grafo.

Los puntos de articulación son especialmente importantes en la teoría de grafos y en la planificación de redes, ya que representan lugares críticos que, si fallan, pueden dividir una red en partes no conectadas. La identificación de puntos de articulación es útil en el diseño de redes robustas y en la optimización de la conectividad en sistemas distribuidos.

Punto 4:

La estrategia "Divide y Vencerás" es un enfoque algorítmico que resuelve un problema dividiéndolo en subproblemas más pequeños, resolviendo cada subproblema de manera recursiva y luego combinando las soluciones de los subproblemas para obtener la solución final del problema original. Esta estrategia se utiliza comúnmente para mejorar la eficiencia de algoritmos, especialmente en problemas que pueden dividirse en partes más pequeñas y solucionarse de manera independiente.


Características clave de "Divide y Vencerás":

1. **División del Problema:** Divide el problema en subproblemas más pequeños y manejables. Esta división puede realizarse de manera recursiva hasta que los subproblemas sean lo suficientemente simples para ser resueltos directamente.
2. **Conquista:** Resuelve los subproblemas de manera recursiva. Cada subproblema se resuelve independientemente hasta que se alcance un caso base.
3. **Combinación de Soluciones:** Combina las soluciones de los subproblemas para obtener la solución final del problema original.

Ejemplo: Merge Sort (Ordenamiento por Mezcla):

Un ejemplo clásico de la estrategia "Divide y Vencerás" es el algoritmo de ordenamiento por mezcla (Merge Sort). Aquí está el pseudocódigo y una breve descripción:

plaintext

 Copy code

```
Función MergeSort(arr):  
    Si longitud(arr) <= 1:  
        Devolver arr  
  
    // Paso de División  
    Dividir arr en dos mitades, izquierda y derecha  
  
    // Paso de Conquista  
    izquierda = MergeSort(izquierda)  
    derecha = MergeSort(derecha)  
  
    // Paso de Combinación  
    Devolver Combina(izquierda, derecha)
```

En Merge Sort, el arreglo se divide en mitades, luego cada mitad se ordena de manera recursiva mediante llamadas a `MergeSort`. Finalmente, las mitades ordenadas se combinan en una nueva lista ordenada a través de la función `Combina`. Este proceso de dividir, conquistar y combinar asegura que el arreglo completo esté ordenado al final del algoritmo.

La eficiencia de Merge Sort es $O(n \log n)$, lo que lo hace más eficiente que algunos algoritmos de ordenamiento tradicionales, especialmente para grandes conjuntos de datos. Este ejemplo ilustra cómo "Divide y Vencerás" puede mejorar la eficiencia al descomponer un problema en partes más pequeñas y resolverlas de manera independiente.