

Padrón:

Apellido y Nombre:

Correo electrónico:

Cuatrimestre:

1) Conceptos básicos de complejidad

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

	Afirmación	Indicar V o F
1	$T(n) = 2T(n-1) + n$ y $T(0) = 1$ entonces $T(n)$ pertenece a $O(n)$ para el peor caso	
2	$T(n) = 2T(n/2) + n^2$ y $T(1) = 1$ entonces $T(n)$ pertenece a $O(n)$ para el peor caso	

B) Dé un ejemplo de algoritmo que responda a esta ecuación de recurrencia: $T(n) = T(n-1) + n^2$ y $T(0) = 1$.

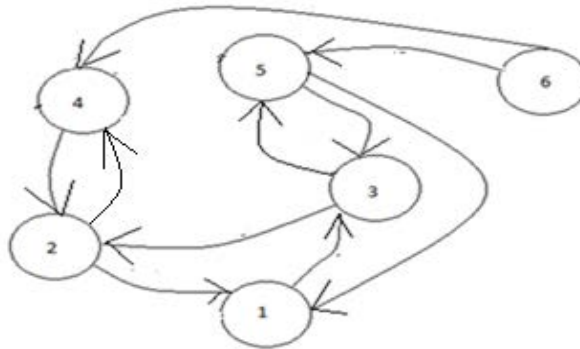
2) TDA Conjunto:

- Diseñar un algoritmo que permita determinar cuántas hojas tiene un ABB. Indicar eficiencia
- Considere esta secuencia de datos: 25, 20, 15, 30, 35, 32. Muestre gráficamente cómo quedan almacenados en un AVL.
- ¿Cómo se puede armar de manera eficiente un heap de mínimo sobre este array? Mostrar gráficamente el proceso. Describir el algoritmo

20	35	10	15	5	4	20
----	----	----	----	---	---	----

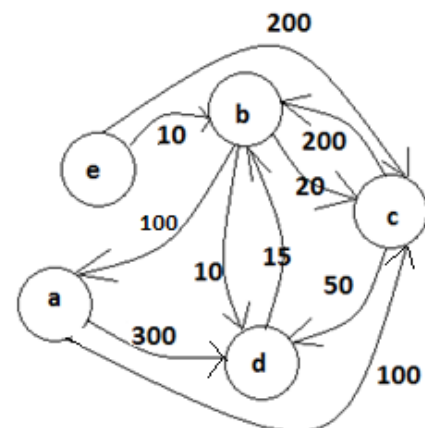
3) TDA Grafo:

a) El siguiente grafo



¿Es conexo? ¿Cuáles son sus componentes conexas? Realice un recorrido en profundidad del mismo
¿Es posible recorrerlo respetando las precedencias? Si es posible, indique un recorrido en esas condiciones. Si no es posible, explique por qué.

b) Para el siguiente grafo, indicar los costes de los caminos mínimos con origen en E. Describa el algoritmo con detalle y las estructuras usadas en la implementación para mejorar la eficiencia cuando el número de nodos es alto respecto del número de enlaces.



4) Estrategias de resolución de problemas: Caracterice y ejemplifique la estrategia "Programación Dinámica". Mencione algún algoritmo que la utilice.

Para determinar si $T(n)$ pertenece a $O(n)$ en el peor caso, podemos utilizar el método de sustitución. Aquí tienes el proceso:

Ecuación de Recurrencia:

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

Condiciones Iniciales:

$$T(0) = 1$$

Método de Sustitución:

Supongamos que $T(k)$ pertenece a $O(k)$ para algún $k < n$. Entonces, probamos a demostrar que $T(n)$ también pertenece a $O(n)$.

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

Supongamos $T(k)$ pertenece a $O(k)$ para algún $k < n$, es decir, existe una constante c tal que $T(k) \leq ck$.

Entonces, sustituimos en la ecuación de recurrencia:

$$T(n) \leq 2(c(n-1)) + n$$

$$T(n) \leq 2cn - 2c + n$$

Asumamos que $n > 2c$, de manera que $2c$ no domine el término n en la expresión anterior. Entonces, podemos decir:

$$T(n) \leq 3cn - 2c$$

Entonces, hemos mostrado que si $T(k)$ pertenece a $O(k)$ para algún $k < n$, entonces $T(n)$ también pertenece a $O(n)$.

Base de Inducción:

- Base: Para $n = 0$, $T(0) = 1$ (la condición inicial), y 1 pertenece a $O(0)$ (ya que 1 es una constante).
- Inducción: Hemos demostrado que si $T(k)$ pertenece a $O(k)$, entonces $T(n)$ también pertenece a $O(n)$.



Por lo tanto, podemos concluir que $T(n)$ pertenece a $O(n)$ en el peor caso.

Ecuación de Recurrencia:

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

Condiciones Iniciales:

$$T(1) = 1$$

Método de Sustitución:

Supongamos que $T(k)$ pertenece a $O(k^2)$ para algún $k < n$. Entonces, probamos a demostrar que $T(n)$ también pertenece a $O(n^2)$.

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

Supongamos $T(k)$ pertenece a $O(k^2)$ para algún $k < n$, es decir, existe una constante c tal que $T(k) \leq ck^2$.

Entonces, sustituimos en la ecuación de recurrencia:

$$T(n) \leq 2c\left(\frac{n}{2}\right)^2 + n^2$$

Simplificamos:

$$T(n) \leq \frac{c}{2}n^2 + n^2$$

$$T(n) \leq \frac{3c}{2}n^2$$

Asumamos que $n > 1$, de manera que $\frac{3c}{2}n^2$ no domine el término n^2 en la expresión anterior.

Entonces, hemos mostrado que si $T(k)$ pertenece a $O(k^2)$ para algún $k < n$, entonces $T(n)$ también pertenece a $O(n^2)$.


Base de Inducción:

- Base: Para $n = 1$, $T(1) = 1$ (la condición inicial), y 1 pertenece a $O(1)$ (ya que 1 es una constante).
- Inducción: Hemos demostrado que si $T(k)$ pertenece a $O(k^2)$, entonces $T(n)$ también pertenece a $O(n^2)$.

Por lo tanto, podemos concluir que $T(n)$ pertenece a $O(n^2)$ en el peor caso.

La ecuación de recurrencia $T(n) = T(n - 1) + n^2$ describe un algoritmo cuyo tiempo de ejecución está definido en función del cuadrado de la entrada. Aquí tienes un ejemplo de un algoritmo recursivo que sigue esta ecuación de recurrencia:

python

 Copy code

```
def ejemplo_algoritmo(n):
    if n == 0:
        return 1
    else:
        return ejemplo_algoritmo(n - 1) + n**2

# Ejemplo de uso:
resultado = ejemplo_algoritmo(5)
print(resultado)
```

Este algoritmo es una función recursiva que se llama a sí misma con $n - 1$ y suma el cuadrado de n al resultado de la llamada recursiva. La condición base es cuando n es igual a 0, en cuyo caso devuelve 1.

Este código sigue la estructura de la ecuación de recurrencia $T(n) = T(n - 1) + n^2$ con la condición inicial $T(0) = 1$. Cada llamada recursiva reduce n en 1 y suma el cuadrado de n .

Punto 2.C

Para construir eficientemente un heap de mínimo a partir de un array dado, puedes utilizar el algoritmo conocido como "heapify". Aquí tienes el procedimiento para convertir el array en un heap de mínimo:

1. **Comenzar desde el último nodo no hoja y trabajar hacia atrás:**

- Los nodos desde $\frac{n}{2} - 1$ hasta 0 son los únicos que pueden tener hijos. Estos nodos son los candidatos iniciales para aplicar el procedimiento de "heapify".

2. **Aplicar el procedimiento de "heapify" a cada nodo no hoja:**

- Para cada nodo no hoja, compáralo con sus hijos (si existen) y asegúrate de que el nodo sea menor que sus hijos. Si no es así, intercambia el nodo con el hijo de menor valor y continúa aplicando el procedimiento en el hijo.

3. **Repetir el proceso hasta llegar al nodo raíz:**

- Repite el procedimiento hasta llegar al nodo raíz. Después de este paso, el array se ha convertido en un heap de mínimo.

Aquí está el pseudocódigo para el procedimiento:

Aquí está el pseudocódigo para el procedimiento:

plaintext

Copy code

```
funcion heapify(arr, n, i):
    menor = i
    izquierda = 2 * i + 1
    derecha = 2 * i + 2

    // Comparar con el hijo izquierdo
    si izquierda < n y arr[izquierda] < arr[menor]:
        menor = izquierda

    // Comparar con el hijo derecho
    si derecha < n y arr[derecha] < arr[menor]:
        menor = derecha

    // Si el menor no es el nodo actual, intercambiar y seguir heapificando
    si menor != i:
        intercambiar(arr[i], arr[menor])
        heapify(arr, n, menor)

// Construir el heap de mínimo
funcion construir_heap(arr):
    n = longitud(arr)

    // Comenzar desde el último nodo no hoja y trabajar hacia atrás
    para i desde  $\lfloor \frac{n}{2} - 1 \rfloor$  hasta 0 decrementando:
        heapify(arr, n, i)
```

Este algoritmo garantiza que cada nodo sea menor o igual que sus hijos, construyendo así un heap de mínimo eficientemente. En tu caso, puedes aplicar

`construir_heap([20, 35, 10, 15, 5, 4, 20])` para convertir el array dado en un heap de mínimo.

Un grafo se considera conexo cuando existe al menos un camino entre cada par de vértices en el grafo. En otras palabras, un grafo es conexo si para cualquier par de vértices u y v , hay al menos un camino que conecta u y v .

Hay dos tipos principales de conexidad que se suelen considerar:

1. Conexidad en un Grafo No Dirigido:

- **Grafo Conexo:** Un grafo no dirigido es conexo si hay un camino entre cada par de vértices.
- **Componentes Conexas:** Si un grafo no dirigido no es conexo, puede descomponerse en componentes conexas. Cada componente conexa es un subgrafo conexo en sí mismo.

2. Conexidad en un Grafo Dirigido:

- **Grafo Fuertemente Conexo:** Un grafo dirigido es fuertemente conexo si hay un camino dirigido entre cada par de vértices.
- **Componentes Fuertemente Conexas:** Similar a los grafos no dirigidos, si un grafo dirigido no es fuertemente conexo, se puede descomponer en componentes fuertemente conexas.

Métodos para Verificar Conexidad:

- **Búsqueda en Profundidad (DFS) o Búsqueda en Anchura (BFS):** Estos algoritmos pueden utilizarse para explorar el grafo y verificar si se puede llegar a todos los vértices desde un vértice inicial.
- **Matriz de Adyacencia o Lista de Adyacencia:** Al inspeccionar la representación del grafo, puedes determinar si hay caminos entre todos los pares de vértices.

En resumen, la conexión en un grafo implica la existencia de caminos entre todos los pares de vértices, y la existencia de tales caminos puede verificarse mediante métodos algorítmicos o inspeccionando la estructura del grafo.

El recorrido en profundidad (DFS, por sus siglas en inglés, Depth-First Search) es un algoritmo utilizado para explorar y visitar todos los nodos (vértices) de un grafo. El DFS se basa en la idea de explorar tan profundamente como sea posible a lo largo de cada rama antes de retroceder.

Aquí te presento una descripción general del algoritmo de DFS:

1. Inicialización:

- Selecciona un nodo inicial para comenzar la exploración. Marca este nodo como visitado.

2. Exploración:

- Desde el nodo actual, elige un vecino no visitado y muévete a ese vecino.
- Marca el vecino como visitado.
- Repite este proceso recursivamente para el vecino seleccionado.

3. Retroceso (Backtracking):

- Cuando no hay más vecinos no visitados desde el nodo actual, retrocede al nodo anterior (el nodo desde el cual se llegó al nodo actual) y busca otros vecinos no visitados desde ese nodo.
- Repite este proceso hasta que todos los nodos hayan sido visitados.

4. Repetición:

- Si quedan nodos no visitados, selecciona uno de ellos como nuevo nodo inicial y repite el proceso.

El DFS se puede implementar de manera recursiva o mediante el uso de una pila para mantener un seguimiento de los nodos a visitar. La versión recursiva es más simple de entender, mientras que la versión con pila a menudo se utiliza en la práctica para evitar el riesgo de desbordamiento de la pila en grafos grandes.

El DFS se puede implementar de manera recursiva o mediante el uso de una pila para mantener un seguimiento de los nodos a visitar. La versión recursiva es más simple de entender, mientras que la versión con pila a menudo se utiliza en la práctica para evitar el riesgo de desbordamiento de la pila en grafos grandes.

Aquí tienes un ejemplo de pseudocódigo para DFS recursivo:

plaintext

Copy code

```
DFS(nodo):  
    Marcar el nodo como visitado  
    Para cada vecino no visitado del nodo:  
        Llamar recursivamente a DFS(vecino)
```

El DFS se utiliza comúnmente para la búsqueda de componentes conexas, detección de ciclos en grafos, y muchas otras aplicaciones en teoría de grafos.

Recorrer un grafo respetando las precedencias implica visitar los nodos de manera que se respeten las relaciones de precedencia definidas entre ellos. En otras palabras, se asegura de que se visiten primero los nodos que deben ser procesados antes de otros según las restricciones de precedencia del grafo.

La noción de precedencia se refiere a una relación de orden entre los nodos del grafo, donde algunos nodos deben ser procesados antes que otros. Este concepto es comúnmente aplicado en situaciones donde hay dependencias o restricciones temporales entre las tareas representadas por los nodos del grafo.

Un ejemplo común es el grafo de dependencias de tareas, donde los nodos son tareas y las aristas representan las dependencias entre ellas. En este contexto, recorrer el grafo respetando las precedencias significa que una tarea no puede ser realizada hasta que todas sus tareas predecesoras hayan sido completadas.

El algoritmo topológico de ordenación es un método específico para recorrer un grafo dirigido acíclico (DAG) respetando las precedencias. Este algoritmo encuentra un orden lineal de los nodos que respeta las relaciones de precedencia.

El proceso de recorrer el grafo respetando las precedencias generalmente involucra el uso de técnicas específicas para garantizar que se cumplan las restricciones impuestas por la estructura del grafo y las relaciones de precedencia entre los nodos.

El algoritmo de camino mínimo es un conjunto de algoritmos utilizados para encontrar el camino más corto entre dos puntos en un grafo ponderado. Dos de los algoritmos más conocidos para resolver este problema son el algoritmo de Dijkstra y el algoritmo de Bellman-Ford.

1. Algoritmo de Dijkstra:

- **Objetivo:** Encontrar el camino más corto desde un nodo de inicio a todos los demás nodos en un grafo ponderado (con pesos no negativos).
- **Funcionamiento:**
 - 1.1. Se asigna una distancia inicial de 0 al nodo de inicio y de infinito a todos los demás nodos.
 - 1.2. Selecciona el nodo con la distancia mínima y examina sus vecinos.
 - 1.3. Actualiza las distancias de los nodos vecinos si se encuentra un camino más corto.
 - 1.4. Marca el nodo actual como visitado y repite los pasos 2 y 3 hasta visitar todos los nodos o llegar al nodo de destino.

2. Algoritmo de Bellman-Ford:

- **Objetivo:** Encontrar el camino más corto desde un nodo de inicio a todos los demás nodos en un grafo ponderado (permite pesos negativos pero no ciclos negativos).
- **Funcionamiento:**
 - 2.1. Se asigna una distancia inicial de 0 al nodo de inicio y de infinito a todos los demás nodos.
 - 2.2. Se actualizan las distancias iterativamente considerando todas las aristas en el grafo.
 - 2.3. Se repite el paso 2 $V-1$ veces (V es el número de nodos) para asegurar convergencia.
 - 2.4. Si se encuentra una distancia más corta en la iteración final, el grafo tiene un ciclo negativo.

Estos algoritmos son fundamentales para problemas como la navegación de mapas, redes de comunicación, logística y muchos otros donde se busca el camino más corto entre dos puntos en un grafo ponderado. La elección entre Dijkstra y Bellman-Ford depende de las características específicas del grafo en cuestión y de si se permiten o no pesos negativos.

La programación dinámica es una técnica de resolución de problemas que involucra la descomposición de un problema en subproblemas más pequeños y la resolución de cada subproblema solo una vez, almacenando sus soluciones para evitar recálculos innecesarios. Esta estrategia se utiliza para optimizar la eficiencia de algoritmos y reducir la complejidad temporal de problemas.

La programación dinámica generalmente sigue los siguientes pasos:

1. Caracterización de la Estructura Óptima:

- Identificar la estructura óptima de la solución al problema y determinar cómo se relacionan las soluciones a subproblemas más pequeños.

2. Definición de la Función de Recurrencia:

- Formular una función de recurrencia que exprese la solución del problema en función de las soluciones de sus subproblemas más pequeños.

3. Identificación de Subproblemas Solapados:

- Determinar si hay subproblemas que se resuelven más de una vez. Si es así, la programación dinámica evita los recálculos almacenando y reutilizando los resultados de subproblemas anteriores.

4. Determinación del Orden de Resolución:

- Decidir el orden en que se resuelven los subproblemas. Puede ser ascendente (de los más pequeños a los más grandes) o descendente (de los más grandes a los más pequeños).

5. Almacenamiento de Resultados:

- Almacenar las soluciones de los subproblemas resueltos para evitar cálculos redundantes.

6. Recursión o Iteración:

- Implementar la solución mediante un enfoque recursivo o iterativo, según la naturaleza del problema y las preferencias de implementación.

Ejemplos comunes de problemas que se resuelven eficientemente mediante programación dinámica incluyen el cálculo del n-ésimo número de Fibonacci, la resolución del problema de la mochila, y la búsqueda del camino más corto en un grafo ponderado.

La programación dinámica puede ser un enfoque muy poderoso para resolver problemas complejos y optimizar algoritmos, pero requiere un cuidadoso análisis de la estructura del problema y una formulación adecuada de las funciones de recurrencia.

Aquí tienes un ejemplo de pseudocódigo para calcular el n-ésimo número de Fibonacci utilizando programación dinámica y memorización:

plaintext

Copy code

```
funcion fibonacci(n):
    si n <= 1:
        devolver n

    // Inicializar una tabla de memorización para almacenar resultados
    tabla = nueva tabla con tamaño n + 1
    para cada elemento en la tabla:
        tabla[elemento] = -1

    // Llamar a la función auxiliar que utiliza la memorización
    devolver fibonacci_aux(n, tabla)

funcion fibonacci_aux(n, tabla):
    // Si ya hemos calculado el resultado, devolverlo desde la tabla
    si tabla[n] != -1:
        devolver tabla[n]

    // Calcular Fibonacci para n utilizando resultados previamente calculados
    resultado = fibonacci_aux(n - 1, tabla) + fibonacci_aux(n - 2, tabla)

    // Almacenar el resultado en la tabla de memorización
    tabla[n] = resultado

    devolver resultado

// Ejemplo de uso
n = 5
resultado = fibonacci(n)
imprimir "El", n, "-ésimo número de Fibonacci es:", resultado
```

Este pseudocódigo utiliza una función `fibonacci` que inicializa una tabla de memorización y luego llama a una función auxiliar `fibonacci_aux` que realiza el cálculo utilizando la memorización para evitar recalcular resultados ya conocidos. La tabla de memorización se inicializa con valores `-1` y se utiliza para almacenar los resultados intermedios. Este enfoque mejora la eficiencia del cálculo al evitar redundancias en la recursión.

