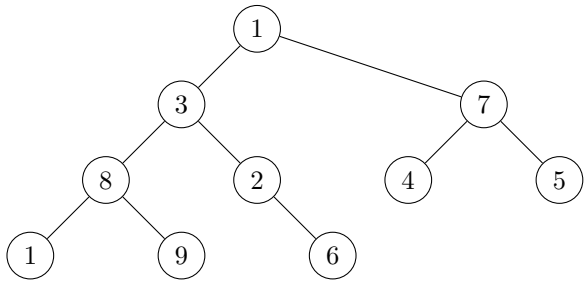
Estructuras de Datos y Algoritmos 1 - ST0245
Segundo Parcial 002 - Martes

Nombre .....
Departamento de Informática y Sistemas
Universidad EAFIT

Mayo 11 de 2021

1 Árboles 30%

En empresas como Bancolombia, los árboles se utilizan para simbolizar decisiones; por ejemplo, las condiciones que se deben analizar para decidir si a un cliente se le puede otorgar un crédito o no. Veamos un ejercicio interesante sobre árboles binarios –muy común en entrevistas según el portal *LeetCode*. Dado un árbol binario, es necesario encontrar el sub-árbol que tiene el mayor promedio. El promedio de un sub-árbol, es la suma del valor de su nodo raíz más la suma del valor de todos sus nodos descendientes, divido por el número de nodos que aportan a tal suma. Para el siguiente árbol, la respuesta es 9 que corresponde al sub-árbol donde la raíz es el nodo 9 y no tiene hijos ni izquierdo ni derecho. El siguiente código resuelve el problema, pero le faltan algunas líneas.



Si trabajas en Java, considera el siguiente código:

```
1  class Node {
2    Node left; Node right; int val;
3    Node(int val) { this.val = val;
4  } }
5
6  class Pair {
7    int sum; int size;
8    Pair(int sum, int size) {
9      this.sum = sum; this.size = size;
10 } }
11
12 class Algorithm {
13  static double promMax = 0.0;
14  static double solve(Node root) {
15   solveTemp(root); return promMax;
16  }
17  static Pair solveTemp(Node root) {
18   if (root == null) return new Pair(0, 0);
19   Pair l = solveTemp(root.left);
20   Pair r = solveTemp(root.right);
21   int sum = ......................;
22   int sz = 1 + l.size + r.size;
23   promMax = Math.max(promMax, ......................);
24   return new Pair(sum, sz);
25 }}
```

(A) (10%) Completa la línea 21 .....

(B) (10%) Completa la línea 23 .....

(C) (10%) ¿Cuál es la ecuación de recurrencia, para el peor caso, del algoritmo anterior? T(n) = ...

Si trabajas en Python, considera el siguiente código:

```
1  class Node:
2    def __init__(self, val):
3      self.val = val
4      self.left = None
5      self.right = None
6
7  promMax = 0.0
8  theSum = 0
9  theSize = 1
10
11 def solve(root):
12   solveTemp(root)
13   return promMax
14
15 def solveTemp(root):
16   global promMax
17   if root == None:
18       return (0, 0) #theSum, theSize
19   l = solveTemp(root.left)
20   r = solveTemp(root.right)
21   sum = ......................
22   sz = 1 + l[theSize] + r[theSize]
23   promMax = max(promMax, ........................)
24   return (sum, sz) #theSum, theSize
```

(A) (10%) Completa la línea 21 .....

(B) (10%) Completa la línea 23 .....

(C) (10%) ¿Cuál es la ecuación de recurrencia, para el peor caso, del algoritmo anterior? T(n) = ...

2 Pilas 20%

En la vida real, determinar si una cadena con paréntesis, llaves y corchetes es válida es de interés para grandes empresas como *Microsoft* (creador del lenguaje C#) u *Oracle* (dueño del lenguaje Java). Además, es un ejercicio muy común en entrevistas para estas empresas según el portal *Geeks for Geeks*. Veamos el problema. Dada una cadena que contiene –únicamente– los caractéres "(", "{", "[", ")", "}", "]", determina si ésta es válida. La cadena es válida si para cada caractér que abre, siempre hay uno que cierra en el orden correcto. Como un ejemplo "([ ])" es válida, pero "[(])" no es válida porque el paréntesis ")" debe cerrar antes que el corchete "]". ¡Sólo nos falta completar unas líneas! ¡Ánimo!

Si trabajas en Java, considera este código:

```
1  boolean solve(String exp) {
2    LinkedList<Character> s = new LinkedList();
3    for(int i=0;i<exp.length();++i){
4      char ei = exp.charAt(i);
5      if(ei == '(' | ei == '{' | ei == '[')
6        ..........................;
7      if(ei == ')' | ei == '}' | ei == ']'){
8        if(s.size() == 0 | !valid(s.get(s.size()-1), ei))
9          return false;
10       else{
11         ..........................;
12   } } }
13   return s.size() == 0;
14 }
15 boolean valid(char o, char c) {
16   return (o == '(' && c == ')') |
17          (o == '[' && c == ']') |
18          (o == '{' && c == '}');
19 }
```

En Java, la librería `LinkedList` se puede usar para implementar una pila. En esa librería, `push(e)` agrega el elemento e a la pila y `pop( )` elimina el elemento que se encuenta en el tope de la pila.

Si trabajas en Python, considera este código:

```
1  def solve(exp):
2    s = deque()
3    for i in range(0, len(exp)):
4      ei = exp[i]
5      if ei == "(" or ei == "{" or ei == "[":
6        ........................
7      if ei == ")" or ei == "}" or ei == "]":
8        if len(s) == 0 or not valid(s[len(s)-1], ei):
9          return False
10       else:
11         ........................
12   return len(s) == 0
13
14 def valid(o, c):
15   return (o == "(" and c == ")") or \
16          (o == "[" and c == "]") or \
17          (o == "{" and c == "}")
```

En Python, la librería `deque` se usa para implementar una pila. En `deque`, `append()` agrega al final y `pop()` elimina el último elemento.

(A) (10%) Completa la línea 6 .....

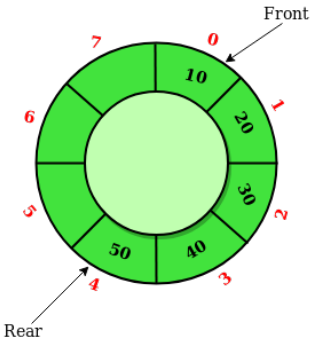
Y Completa la línea 11 .....

(B) (10%) Cuál es la complejidad asintótica, en el peor de los casos, del algoritmo anterior.

O(_____), donde e es el número de caracteres de `exp`. NO existe n.

3 Colas 20%

En la vida real, las colas circulares se usan para planificación de procesos en sistemas operativos como Microsoft Windows 10, Linux Ubuntu 21.04 y Mac OS X. Para lograrlo, se utilizan algoritmos como el de *Round Robin* o versiones mejoradas de este algoritmo. Por ahora, vamos a limitarnos a implementar una cola circular, que nos servirá mucho para trabajar en empresar como Microsoft, Redhat, Oracle o Apple. Sólo nos falta una línea y la complejidad, ¿nos ayudas a completarlas? ¡Por favor! La imagen a continuación muestra un ejemplo de una cola circular.



Si trabajas en Java, considera este código:

```
1  public class CircularQueue {
2      int[] queue;
3      int front;
4      int rear;
5      int currentSize;
6      int size;
7
8      public CircularQueue(int size) {
9          this.queue = new int[size];
10         this.front = 0;
11         this.rear = 0;
12         this.size = size;
13         this.currentSize = 0;
14     }
15     public boolean add(int x) {
16         if (currentSize == size)
17             throw new IllegalStateException("The queue is full: front size: " + rear);
18         queue[rear++] = x;
19         ......................
20         currentSize++;
21         return true;
22     }
23     public int peek() {
24         if (front == rear)
25             throw new IllegalStateException("The queue is empty");
26         return queue[front];
27     }
28     public int poll() {
29         if (front > rear)
30             throw new IllegalStateException("The queue is empty");
31         currentSize--;
32         return queue[front++];
33 }   }
```

En una cola circular, el método `add(e)` agrega un elemento al final de la cola (en inglés, *rear*) el método `poll()` elimina el primer elemento que llegó a la cola (en inglés, *front*) y el método `peek()` permite ver el primer elemento que llegó a la cola, sin eliminarlo. El operador a % b es el residuo de dividir a en b.

(A) (10%) Completa la línea 19 .....

(B) (10%) Cuál es la complejidad asintótica, en el peor de los casos, del algoritmo anterior.

O(_____), donde n es el número de elementos que hay en cola.

Si trabajas en Python, considera este código:

```
1  import numpy as np
2  class CircularQueue:
3      def __init__(self, size):
4          self.queue = np.zeros(size)
5          self.front = 0
6          self.rear = 0
7          self.size = size
8          self.currentSize = 0
9
10     def add(self, x):
11         if self.currentSize == self.size:
12             raise Exception("The queue is full: front size: " + str(self.rear))
13         self.queue[self.rear] = x
14         self.rear += 1
15         ......................
16         self.currentSize += 1
17         return True
18
19     def peek(self):
20         if self.front == self.rear:
21             raise Exception("The queue is empty");
22         return self.queue[self.front]
23
24     def poll(self):
25         if self.front > self.rear:
26             raise Exception("The queue is empty");
27         self.currentSize -= 1
28         self.front += 1
29         return self.queue[self.front-1]
```

En una cola circular, el método `add(e)` agrega un elemento al final de la cola (en inglés, *rear*) el método `poll()` elimina el primer elemento que llegó a la cola (en inglés, *front*) y el método `peek()` permite ver el primer elemento que llegó a la cola, sin eliminarlo. El operador a % b es el residuo de dividir a en b.

(A) (10%) Completa la línea 15 .....

(B) (10%) Cuál es la complejidad asintótica, en el peor de los casos, del algoritmo anterior.

O(_____), donde n es el número de elementos que hay en cola.

4 Tablas de Hash 20%

Cuando un sitio web te solicita tu login y password para autenticarte, el password que se almacena en el servidor es el hash del password. De esa manera, si un atacante accede a los passwords almacenados en el servidor, estos no le serán útiles para autenticarse en el sitio web. Consideremos otro problema con tablas de hash. Dada una cadena que contiene caracteres alfabéticos en minúsculas, tenemos que eliminar como máximo un carácter de esta cadena de manera que la frecuencia de cada carácter distinto sea la misma en la cadena. Sólo nos falta completar una línea y la complejidad, ayúdanos por favor.

Entrada: str = "xyyz"
Salida: Sí
Podemos eliminar el carácter 'y' de la cadena anterior cadena para que la frecuencia de cada carácter sea la misma.
Entrada: str = "xyyzz"
Salida: Sí
Podemos eliminar el carácter 'x' de la cadena cadena para que la frecuencia de cada carácter sea la misma.

Entrada: str = "xxxxyyzz"

Salida: No

No es posible hacer que la frecuencia de cada carácter sea la misma simplemente eliminando como máximo un carácter de la cadena anterior.

Si trabajas en Java, considera este código:

```
1  public class Hashing {
2      static final int M = 26;
3      static int getIdx(char ch) { // Metodo para obtener el indice de un caracter
4          return (ch - 'a');
5      }
6      // Retorna si todos los elementos del arreglo freq que ocurren mas de una vez son iguales
7      static boolean allSame(int freq[], int N){
8          int same = 0;
9          int i; // Selecciona el primer elemento que ocurre mas de una vez
10         for (i = 0; i < N; i++) {
11             if (freq[i] > 0) {
12                 same = freq[i];
13                 break;
14         }   }
15         for (int j = i + 1; j < N; j++)
16             if (freq[j] > 0 && freq[j] != same)
17                 return false;
18         return true;
19     }
20     // Retorna verdadero si podemos hacer que todas las frecuencias sean iguales
21     static boolean possibleSameCharFreqByOneRemoval(String str) {
22         int l = str.length();
23         int[] freq = new int[M]; // Crea el arreglo de frecuencias
24         for (int i = 0; i < l; i++)
25             freq[getIdx(str.charAt(i))]++;
26         if (allSame(freq, M)) // Si todas las frecuencias son iguales,
27             return true;      // retorna verdadero
28             int i = getIdx(c); //  Intenta bajar la frecuencia de cada caracter en uno
29             if (freq[i] > 0) { // Verifica si un caracter ocurre una sola vez en la cadena
30                 freq[i]--;
31                 if (allSame(freq, M))
32                     return true;
33                 freq[i]++;
34         }   }
35         .........................
36 }   }
```

(A) (10%) Completa la línea 35 .....

(B) (10%) ¿Cuál es la complejidad asintótica, para el peor de los casos, del último algoritmo?
O(.....) Donde l es el número de elementos de la cadena. NO hay n en este problema

Si trabajas en Python, considera este código:

```
1  M = 26
2  # Funcion para obtener el indice de un caracter
3  def getIdx(ch):
4      return (ord(ch) - ord('a'))
5  # Retorna si todos los elementos del arreglo freq que ocurren mas de una vez son iguales
6  def allSame(freq, N):
7      # Selecciona el primer elemento que ocurre mas de una vez
8      for i in range(0, N):
9          if(freq[i] > 0):
10             same = freq[i]
11             break
12     for j in range(i + 1, N):
13         if(freq[j] > 0 and freq[j] != same):
14             return False
15     return True
16 # Retorna verdadero si podemos hacer que todas las frecuencias sean iguales
17 def possibleSameCharFreqByOneRemoval(str1):
18     l = len(str1)
19     # Crea el arreglo de frecuencias
20     freq = [0] * M
21     for i in range(0, l):
22         freq[getIdx(str1[i])] += 1
23     # Si todas las frecuencias son iguales, retorna verdadero
24     if(allSame(freq, M)):
25         return True
26
27     # Intenta bajar la frecuencia de cada caracter en uno
28     for i in range(0, 26):
29         # Verifica si un caracter ocurre una sola vez en la cadena
30         if(freq[i] > 0):
31             freq[i] -= 1
32             if(allSame(freq, M)):
33                 return True
34             freq[i] += 1
35     ................
```

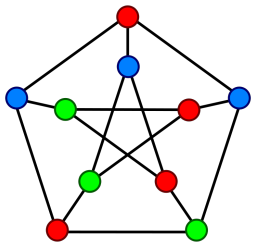
(A) (10%) Completa la línea 35 .....

(B) (10%) ¿Cuál es la complejidad asintótica, para el peor de los casos, del último algoritmo?
O(.....) Donde l es el número de elementos de la cadena. NO hay n en este problema

5 Grafos 10%

El coloramiento de grafos es un problema fundamental en la ingeniería de sistemas y matemática: A partir de este problema se pueden resolver muchas tareas, como lo son asignación de salones, de jaulas en zoológicos, de semáforos o colorear un mapa. Por eso, hoy vamos a desarrollar un algoritmo para colorear un grafo con la mínima cantidad de colores que, infortunadamente, no siempre entrega la mínima cantidad de colores por ser ávaro. Ayúdanos a resolver el problema de encontrar la mínima cantidad de colores para pintar un grafo cualquiera. Colorear un grafo quiere decir asignar colores a sus vértices de tal forma que dos vértices adyacentes NO queden con el mismo color. El algoritmo y un ejemplo están a continuación.

- Colorear el vértice 0 con el color 0.
- Por cada uno de los restantes n–1 vertices del grafo,
  - Tomar un vértice u.
  - Colorear u con el menor color que aún no ha sido utilizado para colorear sus vértices adyacentes.
  - Si todos los colores previos ya han sido utilizados, colorear u con un nuevo color.



Si trabajas en Java, considera este código:

```
1  int colorear(boolean[][] grafo) {
2      int n = grafo.length;
3      int[] color = new int[n + 1];
4      for(int i = 0; i < n; ++i)
5        color[i] = -1;
6      color[0] = 0;
7      boolean[] colordisponible = new boolean[n + 1];
8      for(int i = 0; i < n; ++i)
9        colordisponible[i] = true;
10     for(int u = 1; u < n; ++u) {
11       for(int i = 0; i < n; ++i)
12         if(grafo[u][i])
13           if(color[i] != -1)
14             colordisponible[color[i]] = false;
15       int aval;
16       for(aval = 0; aval < n; ++aval)
17        if(colordisponible[aval])
18          break;
19       color[u] = aval;
20     }
21     for(int i = 0; i < n; ++i)
22       colordisponible[i] = true;
23     boolean[] usado = new boolean[n + 1];
24     int res = 0;
25     for(int i = 0; i < n; ++i)
26       if(!usado[color[i]]){
27         usado[color[i]] = true;
28         res++;
29       }
30     return res+1;
31 }
```

(A) (10%) ¿Cuál es la complejidad asintótica, en el peor de los casos, del algoritmo anterior? Donde V es el número de vértices del grafo.

i) $O(V)$
ii) $O(V \times \log(V))$
iii) $O(V^2)$
iv) $O(1)$
v) $O(V^3)$
vi) $O(\log(V))$

Si trabajas en Python, considera este código:

```
1  def colorear(grafo):
2      n = len(grafo)
3      color = np.zeros(n+1)
4      for i in range(0,n):
5        color[i] = -1
6      color[0] = 0
7      colordisponible = np.zeros(n + 1)
8      for i in range(0,n):
9        colordisponible[i] = 1
10     for u in range (1,n):
11       for i in range(0,n):
12         if grafo[u][i] == 1:
13           if color[i] != -1:
14             colordisponible[int(color[i])] = 0
15       color[u] = 0
16       for aval in range(0,n):
17        if colordisponible[aval] == 1:
18          color[u] = aval
19          break
20     for i in range(0,n):
21       colordisponible[i] = 1
22     usado = np.zeros(n+1)
23     res = 0
24     for i in range(0,n):
25       if usado[int(color[i])] == 0:
26         usado[int(color[i])] = 1
27         res = res +1
28     return res+1
```

(A) (10%) ¿Cuál es la complejidad asintótica, en el peor de los casos, del algoritmo anterior? Donde V es el número de vértices del grafo.

i) $O(V)$

ii) $O(V \times \log(V))$

iii) $O(V^2)$

iv) $O(1)$

v) $O(V^3)$

vi) $O(\log(V))$

6 Meme sobre Complejidad (2% extra)

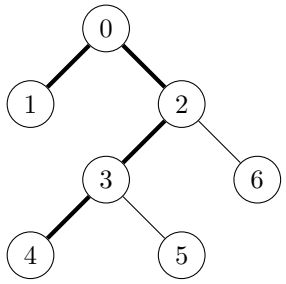
(2 %) En la vida real, los memes se utilizan en presentaciones orales como una estrategia para *romper el hielo*. Con base en los errores que has cometido durante el semestre sobre complejidad, escribe un texto para el siguiente meme:

....
....
....
....
....
....
....

Como un ejemplo, ten en cuenta la regla de la suma, regla del producto, notación O o ecuaciones de recurrencia.

7 (Opcional) Árboles 10%

En la vida real, en arquitecturas de red para sistemas con memoria distribuida, es de suma importancia el diámetro de un árbol. Como bien sabes, un *árbol* es un grafo no dirigido con n vértices y $n-1$ artistas donde el peso de cada arista es 1. Considera un árbol g con raíz v. Sabemos que d_i, usando algoritmo de *búsqueda primero en amplitud* (en Inglés, BFS), es la distancia más corta del nodo v al nodo i. En este ejercicio, vamos a encontrar el *diámetro* de un árbol de n nodos. El diámetro de un árbol se define como el número máximo de aristas del camino más corto entre cualesquiera dos vértices del árbol. Como un ejemplo, en el siguiente árbol, se resalta su diámetro, el cual es de longitud 4. Como un árbol es un grafo, en este ejercicio, el árbol se representa como un grafo utilizando listas de adyacencia.



Si trabajas en Java, considera este código:

La siguiente es la implementación del algoritmo de BFS para calcular las distancias más cortas del nodo v a todos los demás nodos de un árbol

```
1    //Retorna di: las distancias mas cortas del nodo v al nodo i
2    int[] bfs(ArrayList<Integer>[] g, int v) {
3      int[] d = new int[g.length];
4      Arrays.fill(d, Integer.MAX_VALUE);
5      d[v] = 0;
6      Queue<Integer> q;
7      q = new LinkedList<Integer>();
8      q.add(v);
9      while(!q.isEmpty()){
10       int s = q.poll();
11       Iterator<Integer> i=g[s].listIterator();
12       while(i.hasNext()){
13         int n = i.next();
14         if(d[s] + 1 < d[n]){
15           d[n] = d[s] + 1;
16           q.add(n);
17     } } }
18     return d;
19   }
```

Para encontrar el diámetro de un árbol se puede usar el siguiente algoritmo:

- Tomar un vértice v como raíz del árbol g y encontrar d_i, para todo i, usando `bfs(g, v)`.
- Encontrar un vértice u como nodo inicial tal que $d_u \geq d_t$ para todo t. Sea f_i la distancia más corta del nodo u a cualquier nodo i del árbol, el diámetro del árbol es $\max_i f_i$

```
1    int diametro(ArrayList<Integer> g[]){
2      int v, u, w;
3      v = u = w = 0;
4      int[] d = bfs(g, v);
5      int n = d.length;
6      for(int i = 0; i < n; ++i)
7        if( ..... ) u = i;
8      int[] f = bfs(g, u);
9      for(int i = 0; i < n; ++i)
10       if( ..... ) w = i;
11     return f[w];
12   }
```

(A) (10%) ¿Cuál es la complejidad asintótica, en el peor de los casos, del algoritmo anterior? Donde n es el número de vértices del árbol.

i $O(\log_2 n)$
ii $O(n)$
iii $O(n^2)$
iv $O(n \log_2 n)$
v $O(n \times \sqrt{n})$

Y Completa las líneas 7, 10 ....., ...

Si trabajas en Python, considera este código:

La siguiente es la implementación del algoritmo de BFS para calcular las distancias más cortas del nodo v a todos los demás nodos de un árbol

```
1  #Retorna di: las distancias mas cortas del nodo v al nodo i
2  def bfs(g, v):
3      # Rellena un arreglo con "infinito" con el num vertices de g
4      d = np.full(len(g), np.iinfo(np.int32).max)
5      d[v] = 0
6      q = deque()
7      q.append(v)
8      while(len(q) != 0):
9        s = q.pop()
10       for n in g[s]:
11         if d[s] + 1 < d[n]:
12           d[n] = d[s] + 1
13           q.append(n)
14     return d
```

Para encontrar el diámetro de un árbol se puede usar el siguiente algoritmo:

- Tomar un vértice v como raíz del árbol g y encontrar d_i, para todo i, usando `bfs(g, v)`.
- Encontrar un vértice u como nodo inicial tal que $d_u \geq d_t$ para todo t. Sea f_i la distancia más corta del nodo u a cualquier nodo i del árbol, el diámetro del árbol es $\max_i f_i$

```
1  def diametro(g):
2      v = u = w = 0
3      d = bfs(g, v)
4      n = d.size
5      for i in range(0,n):
6        if ..... :
7            u = i
8      f = bfs(g, u)
9      for i in range (0,n):
10       if ..... :
11           w = i
12     return f[w]
```

(A) (10%) ¿Cuál es la complejidad asintótica, en el peor de los casos, del algoritmo anterior? Donde n es el número de vértices del árbol.

i $O(\log_2 n)$

ii $O(n)$

iii $O(n^2)$

iv $O(n \log_2 n)$

v $O(n \times \sqrt{n})$

Y Completa las líneas 6, 10 ....., ...