Estructuras de Datos y Algoritmos 1 - ST0245
Segundo Parcial 001 - Jueves

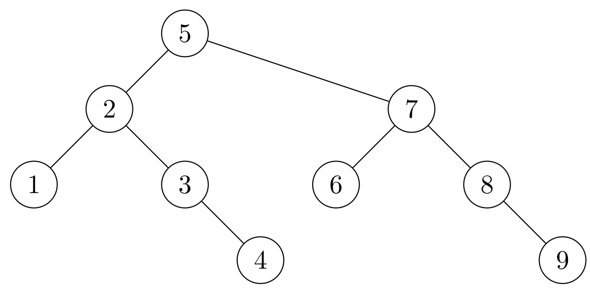
Nombre ....

Departamento de Informática y Sistemas
Universidad EAFIT

Mayo 13 de 2021

1 Árboles 30%

En la vida real, los árboles binarios de búsqueda (BST) se usan en bases de datos relacionales como MySQL de Oracle, SQL Server de Microsoft o DB2 de IBM. ¿Te gustaría trabajar en una de estas empresas? Bueno, vamos a resolver un problema con BSTs. Dado un conjunto de números a, donde $a_i \leq a_j \iff i < j$, crea un BST con el conjunto de números a. Si estás muy emocionado con bases de datos, imagina que a son las llaves primarias de una tabla. Si hay múltiples respuestas, cualquiera de ellas es válida. Para el conjunto de números $a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$, la respuesta es la siguiente:



La respuesta a este problema está en el siguiente algoritmo, pero faltan algunas líneas.

Si trabajas en Java, considera el siguiente código:

```
1  class Node {
2    int data;
3    Node left;
4    Node right
5    Node(int d){data = d;}
6  }
7  private Node solve(int[] a, int l, int r) {
8    if (l > r) {
9      return null;
10   }
11   int m = l + (r - l) / 2;
12   Node root = new Node(a[m]);
13   root.left = .......................;
14   root.right = .......................;
15   return root;
16 }
17 public Node solve(int[] a){
18   return solve(a, 0, a.length - 1);
19 }
```

(A) (10%) Completa la línea 13 ....

(B) (10%) Completa la línea 14 ....

(C) (10%) ¿Cuál es la ecuación de recurrencia para el peor de los casos?
T(n) = ... donde n es el número de elementos de a

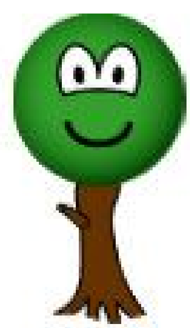
Si trabajas en Python, considera el siguiente código:

```
1  class Node:
2    def __init__(self,d):
3      self.data = d
4      self.left = None
5      self.right = None
6
7  def solveAux(a, l, r):
8    if l > r:
9      return None
10
11   m = int(l + (r - l) / 2)
12   root = Node(a[m])
13   root.left = .......................
14   root.right = ......................
15   return root
16
17 def solve(a):
18   return solveAux(a, 0, len(a) - 1)
```

(A) (10%) Completa la línea 13 ....

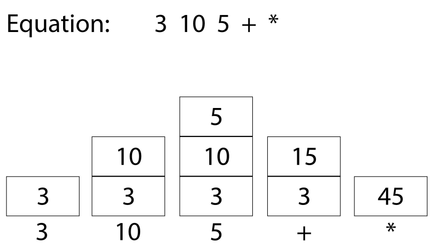
(B) (10%) Completa la línea 14 ....

(C) (10%) ¿Cuál es la ecuación de recurrencia para el peor de los casos?
T(n) = ... donde n es el número de elementos de a



2 Pilas 20%

La *Notación polaca inversa* es una notación matemática en la cual los operadores siguen sus operandos. En la vida real, la notación polaca inversa fue ampliamente utilizada, en los años 70s y 80s, por las calculadoras científicas Hewlett-Packaar. Por ejemplo, la expresión en notación polaca 5 31 4 + $\times$, equivale a la expresión $(31 + 4) \times 5$. Dado un arreglo de operadores y operandos (*tokens*) que representan una expresión en notación polaca, considerando sólo los operadores "(+, *, -, /)", determina el resultado de la expresión. Utilizando una pila, es posible operar una expresión en esta notación, así:



Ya hemos adelantado algo de trabajo, pero falta completar algunas líneas, por favor.

Si trabajas en Java, considera el siguiente código:

```
1  int solve(String[] tokens){
2    String operators = "+-*/";
3    Stack<String> aStack = new Stack<String>();
4    for(String t : tokens){
5      if(!operators.contains(t)){
6        ..............;
7      }else{
8        int a = .................;
9        int b = .................;
10       int index = operators.indexOf(t);
11       if(index == 0)
12         aStack.push(String.valueOf(a+b));
13       else if(index == 1)
14         aStack.push(String.valueOf(b-a));
15       else if(index == 2)
16         aStack.push(String.valueOf(a*b));
17       else // index == 3
18         aStack.push(String.valueOf(b/a));
19     }
20   }
21   return ...............;
22 }
```

(A) (10%) Completa la línea 6

____

(B) (10%) Completa las líneas 8 y 9

____

____

Y completa la línea 21

____

En Java, el método `String.valueOf(a)` convierte el entero a en una cadena de caracteres y el método `Integer.valueOf(b)` convierte la cadena b en un entero.

Si trabajas en Python, considera el siguiente código:

```
1  def solve(tokens):
2    operators = "+-*/"
3    aStack = deque()
4    for t in tokens:
5      if not t in operators:
6        ..................
7      else:
8        a = .........
9        b = ..........
10       index = operators.index(t)
11       if index == 0:
12        aStack.append(str(a+b))
13       elif index == 1:
14        aStack.append(str(a-b))
15       elif index == 2:
16        aStack.append(str(a*b))
17       else: #index == 3
18        aStack.append(str(b/a))
19   return ...............
```

(A) (10%) Completa la línea 6

____

(B) (10%) Completa las líneas 8 y 9

____

____

Y completa la línea 19

____

En Python, la librería `deque` se usa para implementar una pila. En `deque`, `append()` agrega al final y `pop()` elimina el elemento del final. Además, en Python, la función `str(a)` convierte el entero a en una cadena de caracteres y la función `int(b)` convierte la cadena b en un entero.

3 Colas 20%

En la vida real, hacemos filas de espera (*Queues*) para entrar a un parqueadero, para ser atendido en un banco o para comprar un producto en un supermercado. Implementar filas de espera, en un programa informático, es importante para simular, por ejemplo, el impacto que tendría –sobre los tiempos de espera– el contratar un nuevo cajero. Por eso, estas implementaciones son de suma importancia para empresas como Bancolombia. La siguiente clase implementa una fila de espera para un máximo de n elementos. Tan sólo hace falta calcular la complejidad asintótica, para el peor de los casos, de las operaciones entrar a la fila de esperar (*enqueue*) y salir de la fila de espera (*dequeue*).

Si trabajas en Java, considera el siguiente código:

```java
class Queue {
  int e[];
  int index;
  boolean started;

  Queue(int n) {
    e = new int[n];
    index = 0;
    started = false;
  }

  public boolean isEmpty() {
    return (index == 0 && !started) | index <= 0;
  }

  void enqueue(int x){
    if (index == e.length) {
      throw new RuntimeException("Full");
    }
    started = true;
    e[index] = x;
    index++;
  }

  int dequeue(){
    if (isEmpty())
      throw new RuntimeException("Empty");
    int x = e[0];
    for (int i = 0; i < index - 1; i++) {
      e[i] = e[i + 1];
    }
    index--;
    return x;
  }
}
```

(A) (10%) Calcula la complejidad asintótica, para el peor de los casos, de `enqueue(x)`

O(____)

(B) (10%) Calcula la complejidad asintótica, para el peor de los casos, de `dequeue(x)`

O(____)

Si trabajas en Python, considera el siguiente código:

```
import numpy as np
class Queue:

  def __init__(self, n):
    self.e = np.zeros(n)
    self.index = 0;
    self.started = False;

  def isEmpty(self):
    return (self.index == 0 and not self.started) or self.index <= 0

  def enqueue(self,x):
    if self.index == self.e.size:
      raise Exception("Full")
    self.started = True
    self.e[self.index] = x
    self.index = self.index + 1

  def dequeue(self):
    if self.isEmpty():
      raise Exception("Empty")
    x = self.e[0]
    for i in range (0, self.index - 1):
      self.e[i] = self.e[i + 1]
    self.index = self.index - 1
    return x
```

(A) (10%) Calcula la complejidad asintótica, para el peor de los casos, de `enqueue(x)`

O(____________________________________)

(B) (10%) Calcula la complejidad asintótica, para el peor de los casos, de `dequeue(x)`

O(____________________________________)

4 Tablas de Hash 20%

En la vida real, el siguiente problema es frecuente en entrevistas para conseguir un trabajo en empresas como Microsoft, Google y Amazon, según el portal *Geeks for Geeks*. El problema es el siguiente. Dadas dos cadenas S y P, encuentra la ventana más pequeña de S formada por todos los caracteres de P. Tu tarea es completar la función `findSubString()` que toma dos cadenas S y P como parámetros de entrada y devuelve la ventana más pequeña de la cadena S que tenga todos los caracteres de la cadena P. En caso de que haya varias ventanas de la misma longitud, devuelve la que tenga el menor índice inicial. Devuelve "-1" en caso de que no haya ninguna ventana de este tipo. A continuación, algunos ejemplos:

- **Entrada:** S = "timetopráctica" y P = "toc" **Salida:** toprac **Explicación:** "toprac" es la subcadena más pequeña subcadena en la que se puede encontrar "toc".
- **Entrada:** S = "zoomlazapzo" y P = "oza" **Salida:** apzo **Explicación:** "apzo" es la subcadena más pequeña en la que se puede encontrar "oza".

Si trabajas en Java, considera el siguiente código:

```java
1      static final int no_of_chars = 256;
2      // Encontrar la ventana mas pequenha con los caracteres de pat
3      static String findSubString(String str, String pat) {
4          int len1 = str.length();
5          int len2 = pat.length();
6          // Mirar si la longitud de la cadena str es menor a la del patron pat
7          if (len1 < len2)
8              return "-1";
9          int hash_pat[] = new int[no_of_chars];
10         int hash_str[] = new int[no_of_chars];
11         // Guardar las ocurrencias de caracteres del patron pat
12         for (int i = 0; i < len2; i++)
13             hash_pat[pat.charAt(i)]++;
14         int start = 0, start_index = -1,
15             min_len = Integer.MAX_VALUE;    // MAX_VALUE es "infinito"
16         int count = 0;
17         for (......................) {
18             // Contar las ocurrencias de los caracteres en la cadena
19             hash_str[str.charAt(j)]++;
20             // Si los caracteres esta en el patron pat y cadena str, contar
21             if (hash_str[str.charAt(j)] <= hash_pat[str.charAt(j)])
22                 count++;
23             // Si los caracteres de pat son tantos como el contador count
24             if (count == len2) {
25                 // Intentar buscar una ventana mas pequenha
26                 while (....................................) {
27                     if (hash_str[str.charAt(start)] > hash_pat[str.charAt(start)])
28                         hash_str[str.charAt(start)]--;
29                     start++;
30                 }
31                 // Actualizar el tamanho de la ventana
32                 int len_window = j - start + 1;
33                 if (min_len > len_window) {
34                     min_len = len_window;
35                     start_index = start;
36     }   }   }
37         // Si no se encuentra la ventana
38         if (start_index == -1)
39             return "-1";
40         // Retorna la subcadena entre el indice de start con longitud min_len
41         return str.substring(start_index, start_index + min_len);
42     }
```

(A) (10%) Completa, por favor, la línea 17

____

(B) (10%) Completa, por favor, la línea 27

____

Si trabajas en Python, considera el siguiente código:

```
1  no_of_chars = 256
2
3  #Encontrar la ventana mas pequenha con los caracteres de pat
4  def findSubString(string, pat):
5      len1 = len(string)
6      len2 = len(pat)
7      # Mirar si la longitud de la cadena str es menor a la del patron pat
8      if len1 < len2:
9          return "-1"
10     hash_pat = [0] * no_of_chars
11     hash_str = [0] * no_of_chars
12     # Guardar las ocurrencias de caracteres del patron pat
13     for i in range(0, len2):
14         hash_pat[ord(pat[i])] += 1
15     start, start_index, min_len = 0, -1, float('inf') #inf es infinito
16     count = 0  # count of characters
17     for ......................:
18         # Contar las ocurrencias de los caracteres en la cadena
19         hash_str[ord(string[j])] += 1
20         # Si los caracteres esta en el patron pat y cadena str, contar
21         if (hash_str[ord(string[j])] <= hash_pat[ord(string[j])]):
22             count += 1
23
24         # Si los caracteres de pat son tantos como el contador count
25         if count == len2:
26             # Intentar buscar una ventana mas pequenha
27             while ......................:
28                 if (hash_str[ord(string[start])] > hash_pat[ord(string[start])]):
29                     hash_str[ord(string[start])] -= 1
30                 start += 1
31             # Actualizar el tamanho de la ventana
32             len_window = j - start + 1
33             if min_len > len_window:
34                 min_len = len_window
35                 start_index = start
36     # Si no se encuentra la ventana
37     if start_index == -1:
38         return "-1"
39     # Retorna la subcadena entre el indice de start con longitud min_len
40     return string[start_index: start_index + min_len]
```

En Python, `ord(a)` convierte una cadena a con un único caracter en su equivalente numérico en código ASCII.

(A) (10%) Completa, por favor, la línea 17

____

(B) (10%) Completa, por favor, la línea 27

____

5 Grafos 10%

El **algoritmo de Floyd-Warshall** encuentra la distancia mínima entre cada par de vértices i, j de un grafo con pesos. Este algoritmo muchas veces es usado para determinar la clausura transitiva de un grafo, que es útil en Lenguajes Formales y Compiladores. Dada la representación usando **matrices de adyacencia** g de un grafo, donde la posición i, j de la matriz es ∞ si el vértice i no está conectado con el vértice j o la posición i, j es un entero $a_{i,j}$ si existe una conexión entre el vértice i y j, queremos encontrar la distancia mínima entre el vértice u y el vértice v. Ayúdanos a completar el siguiente algoritmo.

Si trabajas en Java, considera el siguiente código:

```java
int calcular(int[][]g,int u,int v){
  int n = g.length;
  int[][] d = new int[n][n];
  for(int i = 0; i < n; ++i){
   for(int j = 0; j < n; ++j){
    d[i][j] = g[i][j];
   }
  }
  for(int k = 0; k < n; ++k){
   for(int i = 0; i < n; ++i){
    for(int j = 0; j < n; ++j){
      int ni = d[i][j];
      int nj = d[i][j];
      int nk = d[k][j];
      int res = Math.min(ni, nj + nk);
      d[i][j] = res;
    }
   }
  }
  return d[u][v];
}
```

(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 el siguiente código:

```python
def calcular(g, u, v):
  n = len(g)
  d = np.zeros((n,n))
  for i in range(0,n):
    for j in range(0,n):
      d[i][j] = g[i][j]
  for k in range(0,n):
    for i in range(0,n):
      for j in range(0,n):
        ni = d[i][j]
        nj = d[i][k]
        nk = d[k][j]
        res = min(ni, nj + nk)
        d[i][j] = res
  return d[u][v]
```

(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)

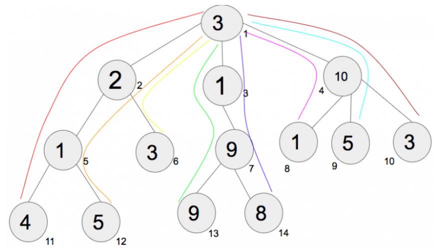
(2 %) En la vida real, los memes se utilizan en presentaciones orales como una estrategia para *romper el hielo*. Con base en los errores que has cometido durante el semestre sobre complejidad, escribe un texto para el siguiente meme:

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

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

7 (Opcional) Árboles 10%

Dado un árbol con N nodos y $N-1$ aristas, calcula, por favor, la suma máxima de los valores de los nodos desde la raíz hasta cualquiera de las hojas sin volver a visitar ningún nodo. A continuación tenemos un diagrama de un árbol con $N = 14$ nodos y $N-1 = 13$ aristas. Los valores en los nodos son 3, 2, 1, 10, 1, 3, 9, 1, 5, 3, 4, 5, 9 y 8 respectivamente para los nodos 1, 2, 3, 4 ... 14. El siguiente diagrama muestra todos los caminos desde la raíz hasta las hojas.



En el dibujo anterior, todos los caminos están marcados por diferentes colores, así:

- Ruta 1(rojo, 3-2-1-4) : suma de todos los valores de los nodos = 10
- Ruta 2(naranja, 3-2-1-5) : suma de todos los valores de los nodos = 11
- Ruta 3(amarillo, 3-2-3) : suma de los valores de todos los nodos = 8
- Ruta 4(verde, 3-1-9-9) : suma de los valores de todos los nodos = 22
- Ruta 5(violeta, 3-1-9-8) : suma de los valores de todos los nodos = 21
- Ruta 6(rosa, 3-10-1) : suma de los valores de todos los nodos = 14
- Ruta 7(azul, 3-10-5) : suma de los valores de todos los nodos = 18
- Ruta 8(marrón, 3-10-3) : suma de los valores de todos los nodos = 16

La respuesta para el árbol del dibujo es 22, ya que la ruta 4 tiene la máxima suma de valores de los nodos en su camino desde una raíz hasta las hojas.

Si trabajas en Java, considera el siguiente código:

```java
    static int[] dp = new int[100];
    // Realiza un recorrido en profundidad para guardar el valor maximo en dp[]
    private static void dfs(int[] a, Vector<Integer>[] v, int u, int parent){
        // Inicializamos dp[u] como a[u]
        dp[u] = a[u - 1];
        // Almacena el maximo valor de los nodos
        int maximum = 0;
        // Recorre el arbol
        for (int child : v[u]) {
            // Si el hijo es padre, continuamos el ciclo
            if (child == parent)
                continue;
            // llamamos dfs para un recorrido nuevo
            dfs(a, v, child, u);
            // Guarda el maximo entre un nodo previo y el nuevo visitado
            maximum = Math.max(maximum, dp[child]);
        }
        // Agrega el maximo valor al nodo padre
        dp[u] += maximum;
    }
    // Funcion que retona el maximo valor de un arbol con nodo a y aristas v
    public static int maximumValue(int[] a, Vector<Integer>[] v) {
        dfs(a, v, 1, 0);
        return dp[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 árbol.

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 el siguiente código:

```python
dp = [0]*100
# Realiza un recorrido en profundidad para guardar el valor maximo en dp[]
def dfs(a, v, u, parent):
    # Inicializamos dp[u] como a[u]
    dp[u] = a[u - 1]
    # Almacena el maximo valor de los nodos
    maximum = 0
    # Recorre el arbol
    for child in v[u]:
        # Si el hijo es padre, continuamos el ciclo
        if child == parent:
            continue
        # llamamos dfs para un recorrido nuevo
        dfs(a, v, child, u)
        # Guarda el maximo entre un nodo previo y el nuevo visitado
        maximum = max(maximum, dp[child])
    # Agrega el maximo valor al nodo padre
    dp[u] += maximum

# Funcion que retona el maximo valor de un arbol con nodo a y aristas v
def maximumValue(a, v):
    dfs(a, v, 1, 0)
    return dp[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 árbol.

i) $O(V)$

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

iii) $O(V^2)$

iv) $O(1)$

v) $O(V^3)$

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