Estructuras de Datos 2 - ST0247
Examen Parcial 1 - Martes (033)

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

Marzo 21 de 2019

NOTAS IMPORTANTES:
- Responda en la hoja de PREGUNTAS
- Marque la hoja de PREGUNTAS

1. Fuerza bruta 20 %

Un problema frecuente es buscar si una una cadena llamada patrón (`pat`) se encuentra dentro de otra cadena llamado texto (`txt`). Esto es lo que sucede en muchos programas cuando le damos *Edición, Buscar*. El objetivo es encontrar la posición en la que aparece por primera vez `pat` en `txt`.

Como un ejemplo, la cadena "atr" aparece dentro de la cadena "patrón" en la posición 1. Como otro ejemplo, la cadena "mat" no aparece en la cadena "patrón". Cuando no aparece, el algoritmo retorna la longitud de la cadena, es decir, en este caso, 5.

Una forma de resolverlo es por fuerza bruta, probando todas las posibles posiciones en las que `pat` puede aparecer dentro de `txt`, como se muestra a continuación:

```
01 int indexOf(String pat, String txt) {
02  int m = pat.length();
03  int n = txt.length();
04  int i, j;
05  for (i = 0, j = 0; i < n && j < m; i++) {
06   if (txt.charAt(i) == pat.charAt(j)) j++;
07   else {
08     i -= j;
09     j = 0;
10   }
11  }
12  if (j == m) return ______  // encontrado
13  else        return n;      // no encontrado
14 }
```

A (10 %) Complete la línea 12

_____

B (10 %) ¿Cuál es la complejidad asintótica, para el peor de los casos, del algoritmo? (En términos de n y m)

O(____)

2. Backtracking 20 %

Para el grafo anterior, complete la salida que darían los siguientes algoritmos:

A (10 %) Complete el orden en que se recorren los nodos usando **búsqueda en profundidad** (en Inglés DFS) a partir de cada nodo. Si hay varias opciones de recorrer el grafo con DFS, elija el vértice más pequeño.

$0 \rightarrow$
$1 \rightarrow 5 \rightarrow 7$
$2 \rightarrow$
$3 \rightarrow$
$4 \rightarrow$
$5 \rightarrow$
$6 \rightarrow$

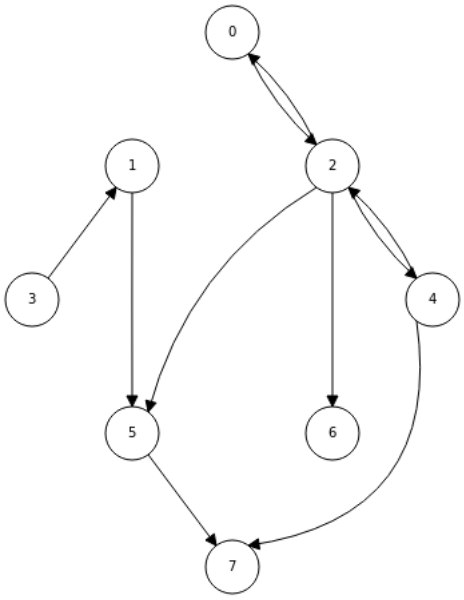
$7 \rightarrow$

B (10 %) Complete el orden en que se recorren los nodos usando **búsqueda en amplitud** (en Inglés BFS) a partir de cada nodo. Si hay varias opciones de recorrer el grafo con BFS, elija siempre el vértice más pequeño.

$0 \rightarrow$
$1 \rightarrow$
$2 \rightarrow 0 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$
$3 \rightarrow$
$4 \rightarrow$
$5 \rightarrow$
$6 \rightarrow$
$7 \rightarrow$

3. Implementación grafos 20 %

Considere el siguiente grafo:



A (10 %) Complete la representación de **matrices de adyacencia**. Si no hay arco, por simplicidad, deje el espacio en blanco. No coloque ceros.

	0	1	2	3	4	5	6	7
0			1					
1						1		
2								
3								
4								
5								
6								
7								

B (10 %) Complete la representación de **listas de adyacencia**. Como el grafo no tiene pesos, sólo se colocan los sucesores en la lista de adyacencia. Los sucesores o vecinos son los vértices a los que se puede llegar a partir de un vértice directamente, no transitivamente.

$0 \rightarrow [2]$
$1 \rightarrow [5]$
$2 \rightarrow$
$3 \rightarrow$
$4 \rightarrow$
$5 \rightarrow$
$6 \rightarrow$
$7 \rightarrow$

4. Backtracking 30 %

Sea un laberinto una matriz de $n \times n$. Una rata tiene que encontrar la ruta para salir. La rata empieza en la esquina superior izquierda (`matriz[0][0]`) y debe ir hasta la esquina inferior derecha (`matriz[N-1][N-1]`) para salir del laberinto. Algunas celdas están bloquedas y por esas no puede pasar. La rata se puede mover en cuatro direcciones: izquierda (*left*), derecha (*right*), abajo (*down*) y arriba (*up*).

La entrada del algoritmo es una matriz donde los valores son cero o uno. Un valor de cero significa que la celda está bloqueda, y un valor de 1 significa que la celda está libre y la rata se puede mover a esa celda.

La matriz `solucion` se utiliza para almacenar la solución. Es importante que la rata sepa cuál es el movimiento inmediatamente anterior para no termi-

nar en un ciclo infinito, por ejemplo, moviéndose izquierda y derecha, sucesivamente, por siempre.

```
01 boolean findPath(int[][] maze, int x,
     int y, int N, String direction) {
02   if(x==N-1 && y==N-1){
03     solution[x][y] = 1;
04     return true;
05   }
06   if (.....................) {
07     solution[x][y] = 1;
08     if(direction!="up" &&
      findPath(maze, x+1, y, N, "down")){
09       return true; }
10     if(direction!="left" &&
      findPath(maze, x, y+1, N,"right")){
11       return true; }
12     if(direction!="down" &&
      findPath(maze, x-1, y, N, "up")){
13       return true; }
14     if(direction!="right" &&
      findPath(.....................)){
15       return true; }
16     solution[x][y] = 0;
17     return false;
18   }
19   return .............;
20 }
21 public boolean isSafeToGo(int[][] maze,
                  int x, int y, int N) {
22   if (x >= 0 && y >= 0 && x < N  &&
       y < N && maze[x][y] != 0) {
23     return true;
24   }
25   return false;
26 }
27 }
```

A (10 %) Completa la línea 6:

_____

B (10 %) Completa la línea 14:

_____, _____, _____, _____, _____

C (10 %) Completa la línea 19:

_____

5. Voraces 10 %

Dados dos arreglos de enteros a, b, de tamaño n cada uno, encuentre dos permutaciones a', b' tal que $\sum_{i=0}^{n-1} |a'_i - b'_i|$ sea tan pequeño como sea posible.

Nota: En Java, `Math.abs(n)` calcula el valor absoluto de n.

```
1    int solve(int[] a, int[] b){
2      int n = a.length;
3      //Paso 1: Procesamiento inicial
4      .............;
5      .............;
6      int res = -1;
7      for(int i = 0; i < n; ++i){
8        //Desicion voraz
9        res += Math.abs(a[i] - b[i]);
10     }
11     return res;
12   }
```

a (10 %) Complete las líneas 4, 5

....., ...