Estructuras de Datos 2 - ST0247 - 022
Examen Parcial 2 (lunes)

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

Mayo 08, 2017

Criterios de calificación

- Selección múltiple con única respuesta
  - Respuesta correcta: 100 %
  - Respuesta incorrecta: 0 %
- Completar código
  - Respuesta correcta 100 %
  - Respuesta incorrecta o vacía 0 %

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

1. Alg. Voraces 20 %

El problema del **agente viajero** consiste en responder la siguiente pregunta: Dada una lista de ciudades y las distancias entre cada par de ciudades, ¿cuál es la ruta posible más corta que visita a cada ciudad exactamente una vez y regresa a la ciudad de origen?

Un algoritmo voraz para solucionar este problem es el **algoritmo del vecino más cercano** también llamado el **Algoritmo de Christofides**. El algoritmo empieza en la primera ciudad y selecciona en cada iteración una nueva ciudad, no visitada, que sea la más cercana a la inmediamente anterior. A continuación una implementación del algoritmo del vecino más cercano que recibe como parámetro un grafo representado como matriz de adyacencia.

La persona que hizo este código es un ingeniero matemático. En Matlab los índices empiezan en 1. Por esta razón, el camino que encuentra el algoritmo empieza con la ciudad numerada con 1 y trabaja con ciclos `while` en lugar de `for`. Adicionalmente, el programador inicia los índices en 1, por ejemplo `i = 1`. Por si fuera poco, la variable `min` no era necesaria inicializarla fuera del bloque `while` y `minFlag` hubiera sido mejor declararla dentro del bloque `while`. Finalmente, el arreglo de visitados sería más eficiente haberlo hecho con tipo `boolean`. No obstante, el programa funciona en Java.

```
01 public void tsp(int adjacencyMatrix[][]) {
02  Stack<Integer> stack = new Stack<Integer>();
03  int numberOfNodes=adjacencyMatrix[1].length-1;
04  int[] visited = new int[numberOfNodes + 1];
05  visited[1] = 1;
06  stack.push(1);
07  int element, dst = 0, i;
08  int min = Integer.MAX_VALUE;
09  boolean minFlag = false;
10  System.out.print(1 + "\t");
11  while (!stack.isEmpty()) {
12   element = stack.peek();
13   i = 1;
14   min = Integer.MAX_VALUE;
15   while (i <= numberOfNodes) {
16    if (adjacencyMatrix[element][i] > 0
```

```
17        && visited[i] == 0) {
18         if (________> _______) {
19          min = adjacencyMatrix[element][i];
20          dst = i;
21          minFlag = true;
22         }
23    }
24    i++;
25   }
26   if (minFlag) {
27    visited[dst] = 1;
28    stack.push(dst);
29    System.out.print(dst + "\t");
30    minFlag = false;
31    continue;
32   }
33   stack.pop();
34  }
}
```

`Continue` es una palabra reservada en Java que permite terminar una iteración de un ciclo abruptamente y pasar a la siguiente iteración del ciclo.

1.1 (10 %) Complete el espacio en la línea 18

_____ > _____

1.2 (10 %) ¿Cuál es la complejidad asintótica para el peor de los casos?

_____

2. Prog. Dinámica 30 %

La **distancia de Levenshtein** es el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra. Se entiende por operación, una inserción, eliminación o la sustitución de un carácter. Es útil en programas que determinan cuán similares son dos cadenas de caracteres, como es el caso de los correctores de ortografía. Como un ejemplo, la distancia de Levenshtein entre “casa” y “calle” es de 3 porque se necesitan al menos tres operaciones para convertir uno en el otro:

1. casa → cala (sustitución de ’s’ por ’l’)
2. cala → calla (inserción de ’l’ entre ’l’ y ’a’)
3. calla → calle (sustitución de ’a’ por ’e’)

A continuación está una implementación del algoritmo en Java:

```
int minimum(int a, int b, int c) {
  return Math.min(Math.min(a, b), c);
}

int Levenshtein(String lhs, String rhs) {
  int[][] distance = new int[lhs.length() + 1]
                           [rhs.length() +1];
  for (int i = 0; i <= lhs.length(); i++)
    distance[i][0] = i;
  for (int j = 1; j <= rhs.length(); j++)
    distance[0][j] = j;
  for (int i = 1; i <= lhs.length(); i++)
    for (int j = 1; j <= rhs.length(); j++)
     distance[i][j] = minimum(
      distance[i - 1][j] + 1,
      distance[i][j - 1] + 1,
      distance[i - 1][j - 1] +
      ((lhs.charAt(i - 1) == rhs.charAt(j - 1)) ? 0 : 1));
  return distance[lhs.length()][rhs.length()];
}
```

En Java, el operador incógnita (?) funciona de la siguiente forma: Si `algo` es verdadero, entonces retorna un `valor`, de lo contrario retorna otro `valor`, así: `algo? un valor : otro valor`.

2.1 (15 %) Complete la siguiente tabla, siguiendo el algoritmo de programación dinámica de la distancia de Levenshtein:

		c	a	l	l	e
c						
a						
s						
a						

2.2 (15 %) Complete la siguiente tabla, siguiendo el algoritmo de programación dinámica de la distancia de Levenshtein, para encontrar la distancia entre madre y mama:

		m	a	d	r	e
m						
a						
m						
a						

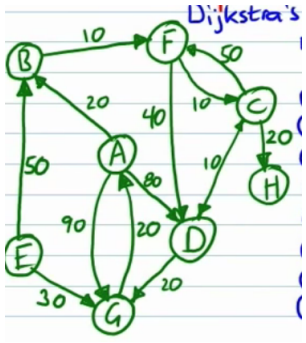
3. Algo. voraces 30 %

El algoritmo de Dijkstra sirve para encontrar el camino más corto de un vértice a todos los demás de un grafo. A continuación una implementación en Java.

```
int minVertex (int [] dist, boolean [] v) {
 int x = Integer.MAX_VALUE; //Infinity
 int y = -1;
 for (int i=0; i<dist.length; i++)
  if (!v[i] && dist[i]<x)
     y=i; x=dist[i];
 return y;
}

int [] dijsktra(Graph dg, int source) {
  int [] dist = new int [dg.size()];
  int [] pred = new int [dg.size()];
  boolean [] visited = new boolean [dg.size()];
  for (int i=0; i<dist.length; i++)
    dist[i] = Integer.MAX_VALUE;
  dist[source] = 0;
  for (int i=0; i<dist.length; i++) {
    int next = minVertex (dist, visited);
    visited[next] = true;
    ArrayList<Integer> n =
        dg.getSuccessors (next);
    for (int j=0; j<n.size(); j++) {
      int v = n.get(j);
      int d = dist[next] +
          dg.getWeight(next,v);
      if (dist[v] > d) {
        dist[v] = d;
        pred[v] = next;
      }}}
  return pred;
}
```

Considere el siguiente grafo:



4.1 (20 %) Complete, por favor, la siguiente tabla, usando el algoritmo de Dijkstra para encontrar el camino más corto del vértice A a todos los demás. En la tabla, la palabra "a" significa "hasta".

Paso	a	B	C	D	E	F	G	H
1	A	20,A	∞	80, A	∞	∞	90, A	∞
2	B	20,A	∞	80, A	∞	30,B	90,A	∞
3								
4								
5								
6								
7								
8								

4.2 (10 %) ¿Cuál es el camino más corto de A a G?

---

4. Prog. dinámica 20 %

El problema de la subsecuencia común más larga es el siguiente. Dadas dos secuencias, encontrar la longitud de la secuencia más larga presente en ambas. Una subsecuencia es una secuencia que aparece en el mismo orden relativo, pero no necesariamente de forma contigua. Como un ejemplo, "abc", "abg", "bdf", "aeg" y "acefg" son subsecuencias de "abcdefg". Entonces, para una cadena de longitud n existen 2^n posibles subsecuencias. Este problema es utilizado en la implementación del comando `diff`, para comparación de archivos, disponible en sistemas Unix. También tiene muchas aplicaciones en bioinformática.

Considere los siguientes ejemplos para el problema:

- Para “ABCDGH” y “AEDFHR” es “ADH” y su longitud es 3.
- Para “AGGTAB” y “GXTXAYB” es “GTAB” y su longitud es 4.

Una forma de resolver este problema es usando *backtracking*, como un ejemplo, para las cadenas “AXYT” y “AYZX”, dada una función recursiva `lcs` que resuelve el problema, se obtendría el siguiente árbol (parcial) de recursión:

```
                lcs("AXYT", "AYZX")
                /                 \
    lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
_____________________________________________________
    lcs("AXY", "AYZX")
      /           \
lcs("AX", "AYZX") lcs("AXY", "AYZ")
_____________________________________________________
    lcs("AXYT", "AYZ")
     /              \
lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

Usando backtracking para ese problema, el problema `lcs(“AXY”, “AYZ”)` se resuelve dos veces. Si dibujamos el árbol de recursión completo, veremos que aparecen más y más problemas repetidos, así como en el caso de serie de Fibonacci. Este problema se puede solucionar guardando la soluciones, que ya se han calculado para los subproblemas, en una tabla; es decir, usando programación dinámica, como en el algoritmo siguiente:

```
01 // Precondición: Ambas cadenas x, y son no vacías
02 public static int lcsdyn(String x, String y) {
03  int i,j;
04  int lenx = x.length();
05  int leny = y.length();
06  int[][] table = new int[lenx+1][leny+1];
07
08  // Inicializa la tabla para guardar los prefijos
09  // Esta inicialización para las cadenas vacías
10  for (i=0; i<=lenx; i++)
11    table[i][0] = 0;
12  for (i=0; i<=leny; i++)
13    table[0][i] = 0;
14
15  // Llena cada valor de arriba a abajo
16  // y de izquierda a derecha
17  for (i = 1; i<=lenx; i++) {
18     for (j = 1; j<=leny; j++) {
19       // Si el último caracter es igual
20       if (x.charAt(i-1) == y.charAt(j-1))
21        table[i][j] = 1+table[i-1][j-1];
22       // De lo contrario, tome el máximo
23       else // de los adyacentes
24        table[i][j] = Math.max(table[i][j-1],
                                 table[i-1][j]);
25       System.out.print(table[i][j]+" ");
26    }
27    System.out.println();
28  }
29
30  // Retornar la respuesta (tipo entero, ojo)
31  _______________________________
32  }
```

5.1 (10 %) ¿Cuál es la complejidad asintótica para el peor de los casos? OJO, n no es una variable en este problema.

_____

5.2 (10 %) Complete la línea 31

_____