Examen parcial 2 de Análisis y Diseño de Algoritmos. Fecha: Sábado 11 de Noviembre de 2017.

Nombre del alumno: \_David Alfonso Velasco Sedano\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Aciertos: \_\_\_\_\_\_\_\_\_\_

I.- Se desea armar la torre de bloques más alta posible, dado un conjunto de bloques con diferente peso y capacidad de carga que incluye a su propio peso. Considere el ejemplo. El bloque C sólo puede cargar 1 unidad de peso adicional.

Torre válida de máxima altura:

|  |
| --- |
| C |
| A |
| B |

Torre no válida:

|  |
| --- |
| B |
| C |

Torre válida:

|  |
| --- |
| C |
| A |

|  |  |  |
| --- | --- | --- |
| Bloque | Peso | Capacidad |
| A | 1 | 7 |
| B | 5 | 10 |
| C | 3 | 4 |

*Restricciones (no se probará con datos fuera de las restricciones):*

1. No se esperan más de 30 bloques de entrada.
2. Todos los pesos son enteros positivos menores que 200.
3. La capacidad de un bloque es un entero positivo mayor a su peso y no mayor que 300.

I.- Implementa un algoritmo *voraz* que reciba los datos de los bloques y devuelva los datos de una torre válida con la máxima altura encontrada. No se espera la solución óptima. Como estrategia, el mejor candidato a incluirse en la solución será aquél con mayor capacidad disponible (excluyendo su peso). Para el ejemplo anterior y usando dicha estrategia, el algoritmo voraz deberá devolver la torre: [A, B].

*Método a implementar*: static int [] torreGreedy (int[] pesos, int[] capacidades)

*Argumentos de entrada*: arreglo de pesos, arreglo de capacidades. Todos son números enteros. Los arreglos recibidos son del mismo tamaño. La posición *k* de cada arreglo refiere a los datos del bloque *k*.

*Salida*:arreglo de índices de bloques que representa la mejor torre obtenida, y se llena del bloque inferior al superior. Para el ejemplo mostrado, el arreglo de salida debe ser: {0, 1}.

public static int[] torreGreedy (int[] pesos, int[] capacidades) {

LinkedList<Element> universe = createList(pesos, capacidades);

LinkedList<Element> tower = new LinkedList<Element>();

int totalCapacity = -1;

int index = 0;

while(universe.size() > 0) {

index = select(universe);

Element candidate = universe.remove(index);

if(totalCapacity < 0) {//First iteration

totalCapacity = candidate.capacity;

tower.add(candidate);

continue;

}

int cap = totalCapacity - candidate.weight;

if(cap >= 0) {

totalCapacity = cap;

tower.add(candidate);

}

}

return toIntArray(tower);

}

**[Ver adjunto Examen2\_Prob1.java para ver toda la solución]**

*Valor del ejercicio: 25 puntos*

II.- Implementa un algoritmo *backtracking* que, dados los mismos datos que el ejercicio anterior, devuelva la mayor altura posible. Este algoritmo deberá explorar a lo largo de todas las soluciones potenciales. El resultado debe ser el óptimo.

Método a implementar: static int alturaBT(int[] pesos, int[] capacidades)

*Argumentos de entrada*: arreglo de pesos, arreglo de capacidades; tal y como se definieron en la sección I.

*Salida*: número entero que denota la máxima altura posible. Para el ejemplo mostrado, devuelve 3, que corresponde a la altura de la torre: [B, A, C] = {1, 0, 2}.

public static int[] alturaBT (int[] pesos, int[] capacidades) {

LinkedList<Element> result = new LinkedList<Element>();

for(int i = 0; i < pesos.length; i++) {

LinkedList<Element> temp = new LinkedList<Element>();

temp.add(new Element(i, pesos[i], capacidades[i]));

LinkedList<Element> propousal = alturaBT(createList(pesos, capacidades), temp);

if(propousal.size() >= result.size())

result = (LinkedList<Element>)propousal.clone();

}

return toIntArray(result);

}

**[Ver adjunto Examen2\_Prob2.java para ver toda la solución]**

*Valor del ejercicio: 25 puntos*

III.- Implementa un algoritmo que resuelva el siguiente problema.

Un guía de turistas tiene la misión de llevar a 12 turistas desde un pueblo de partida a uno destino, en una región del país donde existen muchos pueblos pintorescos comunicados mediante carreteras de dos carriles. Como el guía no tiene vehículo, tiene que subirlos a un camión rural. El problema es que los camiones sólo van de un pueblo a otro, y tienen capacidad limitada. Como no existen camiones disponibles desde un pueblo a todos los demás, y cada camión puede tener diferente capacidad, el guía de turistas tiene que buscar una secuencia de camiones que lo lleve del pueblo inicial al destino, tal que el camión más pequeño en la ruta pueda llevar a todos los turistas. Implementa el siguiente método que determine si es posible encontrar dicha ruta. Por simplicidad, los pueblos se identifican con índices.

Método a implementar: static boolean existeRuta(int[][]mapa, int T, int inicio, int destino)

*Argumentos de entrada*: mapa de carreteras como una matriz de adyacencia, número de turistas a pasear T, ciudad inicial, ciudad destino.

*Salida*: *verdadero*, si es posible ir de la ciudad inicial a la ciudad destino; *falso*, en otro caso.

10

1

20

0

15

2

3

25

Para el mapa de la figura, con T = 12, partida = 0 y destino = 3, el guía sólo puede elegir la ruta 0–2–3. La ruta 0–1–3 no se puede elegir porque el camión que va de 1 a 3 tiene capacidad para 10 pasajeros y existen 12 turistas. El mapa de entrada se representa mediante una matriz de adyacencia de un grafo no dirigido tal que, si no existe un camión entre las ciudades *j*, *k*, *mapa*[*j*][*k*] = 0; en caso contrario, *mapa*[*j*][*k*] almacena la capacidad del camión.

public static boolean existeRuta(int[][] mapa, int T, int inicio, int destino) {

if(inicio == destino) {

return true;

}

int[] distances = new int[mapa.length];

boolean[] processed = new boolean[mapa.length];

int[] voyage = new int[mapa.length];

for(int i = 0; i < distances.length; i++) {

distances[i] = Integer.MAX\_VALUE;

voyage[i] = -1;

}

ArrayDeque<Integer[]> pendingNodes = new ArrayDeque<Integer[]>();

pendingNodes.push(new Integer[]{inicio, -1});

while(!pendingNodes.isEmpty()) {

Integer[] node = pendingNodes.pop();

if(processed[node[0]])

continue;

processed[node[0]] = true;

voyage[node[0]] = node[1];

for(int i = 0; i < mapa.length; i++) {

if(mapa[node[0]][i] > 0 && T <= mapa[node[0]][i]) {

pendingNodes.add(new Integer[]{i, node[0]});

}

}

}

return voyage[destino] > 0 ? true : false;

}

**[Ver adjunto Examen2\_Prob3.java para ver toda la solución]**

*Valor del ejercicio: 25 puntos*

IV.- Dibuje *el segundo mejor árbol de recubrimiento mínimo* del grafo. Justifica tu respuesta

*Valor del ejercicio: 15 puntos.*

2

3

6

3

3

2

2

5

4

1

**[Ver adjunto Examen2\_Prob4.pdf]**

VI.- Se desean realizar búsquedas en tiempo constante sobre un catálogo de servicios médicos de un hospital privado, identificados por una clave formada por una combinación de cinco símbolos del siguiente conjunto de nueve elementos: {6, #, A, ☺, f, @, 3, ©, δ}. ¿Cuál sería el código *hash* asignado a la clave [f ©@☺δ] tomando en cuenta que no se esperan más de 500 servicios en el catálogo y el carácter de la extrema derecha se considera como el dato menos significativo? Incluya el desarrollo (la fórmula desglosada) paso a paso para obtener el código hash.

*Valor del ejercicio: 15 puntos.*

VII.- Considera la siguiente formula llamada “fastF”:

El resultado es en realidad un número de veces en que la súper-formula fue llamado con todos los parámetros mayores a 0.

If *x* = 3, *y* = 3, *z* = 4,

*F*(3, 3, 4) = 1 + F(0, 2, 3) + F(2, 0, 3) + (2, 2, 1) =

1 + 0 + 0 + 1 + F(-1, 1, 0) + F(1, -1, 0) + F(1, 1, -2)

1 + 0 + 0 + 1 + 0 + 0 + 0

2

Implementa **un método eficiente** que calcule el resultado de la súper-fórmula para los 3 valores enteros de entrada *x,y,z*. Como este número puede ser muy alto la salida será el módulo de 59143. No se aceptará la solución recursiva (la cual es muy ineficiente)

*Método a implementar***: static** **int** fastF(**int** x, **int** y, **int** z)

Ejecuta el método con los siguientes ejemplos para probar que el algoritmo es correcto y eficiente:

1. *F* (5, 6, 7) = 11
2. *F* (10, 9, 8) = 68
3. *F* (30, 29, 28) = 12822
4. *F* (95, 96, 97) = 7694
5. *F* (130, 130, 130) = 54204

public static int fastF(int x, int y, int z ) {

int result = 0;

if(x <= 0 || y <= 0 || z <= 0)

return result;

int[][] baseMatrix = {

{-3,-1,-1},

{-1,-3,-1},

{-1,-1,-3}

};

ArrayDeque<Integer[]> pendingToProcess = new ArrayDeque<Integer[]>();

pendingToProcess.add(new Integer[]{x,y,z});

while(!pendingToProcess.isEmpty()) {

result++;

Integer[] superFunc = pendingToProcess.pop();

for(int i = 0; i < 3; i++) {

x = superFunc[0] + baseMatrix[i][0];

y = superFunc[1] + baseMatrix[i][1];

z = superFunc[2] + baseMatrix[i][2];

if(x <= 0 || y <= 0 || z <= 0)

continue;

pendingToProcess.add(new Integer[]{x,y,z});

}

}

return result;

}

**[Ver adjunto Examen2\_Prob7.java para ver toda la solución, solo funciona con los primeros 2 casos]**

*Valor del ejercicio: 25 puntos*