HITO 2

PRACTICA 3 STAR WARS

Algoritmo Voraz

ALUMNOS:
Esther Camacho Caro
Diego Dorado Galán

ESCUELA SUPERIOR DE INFORMÁTICA, UCLM
3/5/2021

Tareas a realizar:

1. Implementar un programa Java que determine cómo distribuir los contenedores entre los Eopies (se pueden proponer e implementar una o dos heurísticas voraces para resolver el problema), y mostrar lo siguiente:
    a. Número inicial de Eopies y contenedores (leídos por teclado)
    b. Información inicial sobre volumen de cada contenedor y volumen soportado por cada Eopie.
    c. Número total de litros transportados por los Eopies.
    d. También, en el caso de que existan, indicar los Eopies que no han podido transportan ningún contenedor.
2. ¿Cuántos litros totales de agua logran repartir los Jawas a las familias de Tatooine en 7 noches, si cada noche solamente pueden realizar un reparto y se utiliza el mismo número de Eopies por noche, pero soportando volúmenes diferentes?
Realice una simulación del problema en este caso.
3. Elaborar un documento en el que se detalle lo siguiente:
    a. Cuáles son los elementos principales del enfoque voraz seguido?
    b. Explicación breve de la(s) estrategia(s) seguida(s) para distribuir los contenedores entre los Eopies y transportar el agua a Tatooine.
    c. Determinar y explicar la complejidad teórica del algoritmo principal que habéis implementado (reparto de contenedores entre Eopies).
    d. ¿Es el algoritmo óptimo en cuanto al resultado obtenido? Justifica tu respuesta o proporciona contraejemplos si es necesario.

Cada clase y método del programa que se implemente debe estar debidamente documentado internamente para facilitar su comprensibilidad. Para ello se seguirá el estándar Javadoc para describir propósito, parámetros, tipo de retorno, etc.
El uso de un algoritmo Voraz es obligatorio.
Se debe realizar un buen diseño orientado a objetos del programa, creando las clases y la organización en métodos que se consideren necesarias.

Explicación del algoritmo voraz

Nuestro algoritmo voraz es bastante sencillo. Consiste en dos bucles for que recorrerán respectivamente el array de objetos Eopies, desde el índice equivalente a la longitud del array hasta 0 (de mayor a menor), y el array de objetos contenedores de la misma forma. Así pues, recorreremos el array de contenedores comparando si el valor de un objeto del array eopies tiene una capacidad mayor o igual a la del contenedor y si dicho contenedor no ha sido usado, asociándose esta al objeto de la clase Eopie en ese caso y cambiando la variable “usado” del objeto contenedor a verdadero a la vez que incrementamos el valor de una variable de tipo entero (un contador) para posteriormente contabilizar el número de Eopies que no han sido utilizados. A su vez, dentro del mismo método, incluimos una llamada a otro método definido como CalcularAguaTrans, que servirá para ayudarnos a calcular la cantidad total de agua transportada. Esto lo hemos realizado simplemente recorriendo el array contenedores, y acumulando la cantidad de agua de cada contenedor con la variable “usado” igual a true. Nuestro método además, devuelve un array de tipo double de dos elementos dónde el primero equivale a la cantidad total de agua transportada en una noche y el segundo al número de Eopies sin utilizar ese día.

Complejidad teórica del Algoritmo Voraz

```
public static double[] cargarEopies (Eopie [] eopies, Contenedor [] contenedor) {

    int contador_eopies =0;
    for (int i= (eopies.length-1); i>=0;i--) {
        for(int j=(contenedor.length-1); j>=0;j--) {
            if (eopies[i].getCapacidad()>= contenedor[j].getCapacidad() && contenedor[j].isUsado() == false) {
                contador_eopies ++;
                contenedor[j].setUsado(true);
                break;
            }
        }
    }
    double cantidad = calcularAguaTrans(contenedor);
    double [] solucion = {cantidad, (double)(eopies.length -contador_eopies)};
    return solucion;
}
```

La estructura básica de nuestro algoritmo voraz se basa en dos bucles for, de complejidades O(n) y O(m) y de un bucle if, cuya complejidad es O(1). Por tanto, la complejidad de nuestro algoritmo voraz, nombrado como “cargarEopies” es O (n · m).

Complejidad teórica de un algoritmo quicksort

```
public static void quicksortCont(Contenedor A[], int izq, int der) {

      double pivote=A[izq].getCapacidad(); // tomamos primer elemento como pivote
      int i=izq;          // i realiza la búsqueda de izquierda a derecha
      int j=der;          // j realiza la búsqueda de derecha a izquierda
      double aux;

      while(i < j){                          // mientras no se crucen las búsquedas
         while(A[i].getCapacidad() <= pivote && i < j) i++; // busca elemento mayor que pivote
         while(A[j].getCapacidad() > pivote) j--;           // busca elemento menor que pivote
         if (i < j) {                        // si no se han cruzado
             aux= A[i].getCapacidad();                    // los intercambia
             A[i].setCapacidad(A[j].getCapacidad());
             A[j].setCapacidad(aux);
         }
       }

       A[izq].setCapacidad(A[j].getCapacidad());      // se coloca el pivote en su lugar de forma que tendremos
       A[j].setCapacidad(pivote);      // los menores a su izquierda y los mayores a su derecha

       if(izq < j-1)
          quicksortCont(A,izq,j-1);          // ordenamos subarray izquierdo
       if(j+1 < der)
          quicksortCont(A,j+1,der);          // ordenamos subarray derecho
    }
```

Debemos mencionar también la importancia de nuestro algoritmo quicksort, por lo que analizaremos su complejidad teórica.
Básicamente, el orden de complejidad del algoritmo en el peor caso será de O (n^2), que ocurrirá cuando el array esté ordenado o casi ordenado. La complejidad teórica en el caso promedio será del orden O (n · log n).

Estrategia empleada

En esta práctica hemos empleado un algoritmo voraz que implementa una heurística voraz. Busca tomar la solución más optima en cada momento (de manera local), buscando una solución óptima a nivel global. Este tipo de algoritmo no siempre garantiza una solución óptima, pero resultan efectivos en los casos en los qué hallar una solución globalmente óptima es muy costoso y resulta efectivo reducirlo a buscar optimizar un problema a nivel local.
En lo que se refiere a nuestra práctica, si que resulta óptimo realizar un algoritmo voraz sobre nuestros datos ya que existe una ordenación previa de los arrays o datos. En caso contrario, no sería del todo eficiente.

▾ Complejidad

Implementación Directa

- El bucle requiere como máximo n (num. obj.) iteraciones: O(n) una para cada posible objeto en el caso de que todos quepan en la mochila
- La función de selección requiere buscar el objeto con mejor relación valor/peso: O(n)
- Coste del algoritmo: $O(n) \cdot O(n) = O(n^2)$

Implementación preordenando los objetos

- Ordena los objetos de mayor a menor relación valor/peso. existen algoritmos de ordenación $O(nlogn)$
- Con los objetos ordenados la función de selección es O(1)
- Coste del algoritmo: $O(max(O(nlogn), O(n) \cdot O(1))) = O(nlogn)$

Este análisis de la complejidad es de un ejercicio en el que se implementa también un algoritmo voraz y muestra la comparativa entre preordenar o no los objetos.

https://colab.research.google.com/drive/1AoxlaVrxVJdGs6THvJLB6zzUwNedoiO2?usp=sharing#scrollTo=XMNykKzoNs6F