Universidad
Rey Juan Carlos

Algoritmos Avanzados

Práctica 1: Algoritmos voraces

Alumno: Santiago Ramos Gómez

Grado: Ingeniería de Computadores 2022-2023

Introducción

En esta práctica se busca resolver el problema de planificación de tareas dado un array (ordenado o desordenado) y a partir de la técnica voraz obtener la mínima espera entre todas las tareas.

Para tener una idea más concreta, se expone el ejemplo propuesto en la práctica:

"Por ejemplo, sean cinco tareas de duraciones {1,2,3,4,5} y 2 procesadores. Se asignarán las tareas de duración 1, 3 y 5 al procesador 0 y las tareas de duración 2 y 4 al procesador 1.

El tiempo de espera de cada tarea será:
tarea de índice 0: 1 = 1
tarea de índice 1: 2 = 2
tarea de índice 2: 1+3 = 4
tarea de índice 3: 2+4 = 6
tarea de índice 4: 1+3+5 = 9

Entonces , el tiempo total de espera de todas las tareas será: 1+2+4+6+9 = 22"

El funcionamiento para un único procesador sería de la siguiente manera:

Dada una lista con valores {1,3,5} se realizan los siguientes operaciones:

Primera iteración: 1

Segunda iteración: 1 + (1+3)

Tercera iteración: 1 + (1+3) + (1+3+5)

Resultado final: 1 + (1+3) + (1+3+5) = 1 + 4 + 9 = 14

Si ahora se toma otra lista {2, 4}, se obtiene un valor final de 8.

Primera iteración: 2

Segunda iteración: 2 + (2+4)

Resultado final: 2 + (2+4) = 2 + 6 = 8

Una vez se ha obtenido un resultado final, se debe realizar la suma de todos **resultados finales** obtenidos.

En este caso: 14 + 8 = 22.

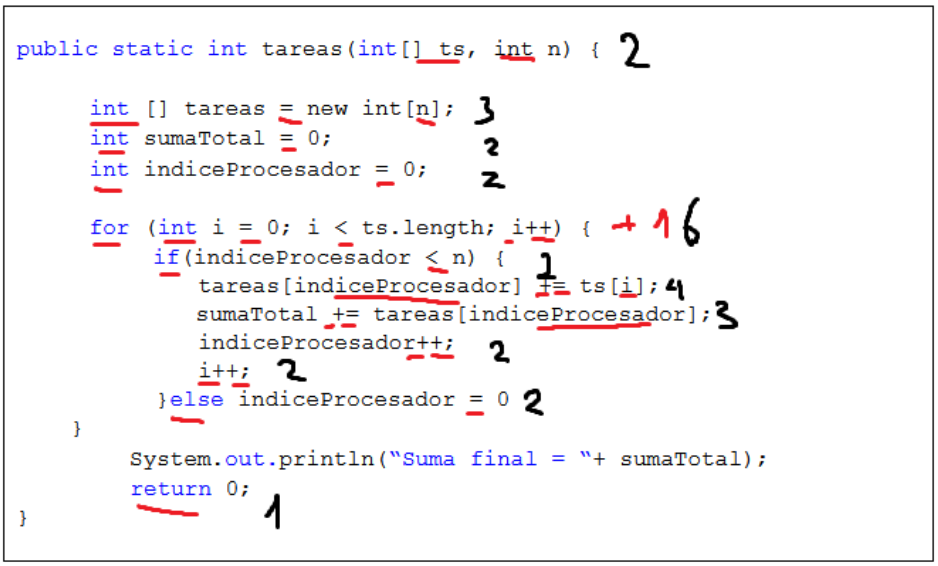
Algoritmo idealizado

```
public static int tareas(int[] ts, int n) {

      int [] tareas = new int[n];
int sumaTotal = 0;
int indiceProcesador = 0;

      for    (int    i    =    0;    i    <    ts.length;    i++)    {
           if(indiceProcesador < n) {
              tareas[indiceProcesador]           +=           ts[i];
              sumaTotal += tareas[indiceProcesador];
              indiceProcesador++;
              i++;
           }else indiceProcesador = 0
    }
         System.out.println("Suma final = "+ sumaTotal);
   return 0;
}
```

Fragmento de código 1: Algoritmo voraz idealizado

El algoritmo voraz idealizado es aquel que cuenta con una entrada de datos ordenados, en este caso el vector "ts". El algoritmo utiliza un vector adicional donde almacena la suma de datos previos con los nuevos de entrada. Una variable sumaTotal, que, como su propio nombre índice, se dedica a acumular la suma de los tiempos de espera, tomando los datos de la previa iteración del anterior vector. Por último, la variable indiceProcesador se trata de iterador para ir intercambiando entre procesadores, es por ello por lo que se reinicia a 0 cuando todos los procesadores han participado.

Análisis de la complejidad temporal del algoritmo:



```
public static int tareas(int[] ts, int n) {

      int [] tareas = new int[n];
      int sumaTotal = 0;
      int indiceProcesador = 0;

      for (int i = 0; i < ts.length; i++) {
          if(indiceProcesador < n) {
             tareas[indiceProcesador] += ts[i];
             sumaTotal += tareas[indiceProcesador];
             indiceProcesador++;
             i++;
          }else indiceProcesador = 0
    }
         System.out.println("Suma final = "+ sumaTotal);
         return 0;
}
```

Figura 1: Análisis de complejidad temporal del algoritmo voraz idealizado

(Los bucles for y while tienen un "+1" correspondiente a la comprobación de salida).

$$T(n) = 9 + \sum_{i=0}^{n} (21) + 1 + 1$$

$$T(n) = 21n + 11$$

Obteniéndose una complejidad temporal de notación O(n).

Análisis de la complejidad espacial del algoritmo:

Suponiendo que el IDE Eclipse utiliza 4 bytes para almacenar un dato de tipo "int", se obtiene:

int [N] tareas = 4 bytes * N;

int [M] ts = 4 bytes * M;

int n, sumaTotal, indiceProcesador = 12 bytes;

Obteniéndose una complejidad espacial = 4 bytes * N + 4 bytes * M + 12.

Siendo una complejidad espacial de notación O(n).

Algoritmo realista

```
public static int tareas(int[] ts, int n) {
     int   []   arrayIndicesOrdenados   =   ordenacionInsercion(ts);
     int [] tareas = new int[n];
int sumaTotal = 0;
int indiceProcesador = 0;

      for    (int    i    =    0;    i    <    ts.length;    i++)    {
           if(indiceProcesador < n) {

              tareas[indiceProcesador] += ts[arrayIndicesOrdenados[i]]

              sumaTotal += tareas[indiceProcesador];
              indiceProcesador++;
              i++;
           }else indiceProcesador = 0
    }
         System.out.println("Suma final = "+ sumaTotal);
   return 0;
}
```

Fragmento de código 2: Algoritmo voraz realista

El algoritmo voraz, a diferencia del idealista, no cuenta con una entrada de datos ordenados. Es por ello por lo que se utiliza un vector adicional en el cual se recoge un vector de índices ordenados (Del vector de datos). Esto se consigue llamando a la función ordenacionInsercion(int [ ] v), la cual devuelve este vector ordenado.

La diferencia respecto al anterior algoritmo viene subrayada en color amarillo, donde se aprecian tanto el nuevo vector como la nueva manera de recorrer el vector de datos, en la cual se debe usar el vector de índices ordenados, y utilizarlo a su vez como el propio índice del vector de datos.

Es decir, si se da un vector V = {5, 2, 3, 1, 4}, y se tiene un vector de índices ordenados del vector V tal que O = {3, 1, 2, 4, 0}, y el acceso conjunto, V [O [0]], se traduce en: V [3] = 1, ya que O [0] = 3.

De esta manera se puede acceder a los datos del array sin tener que alterarlo.

Análisis de la complejidad temporal del algoritmo:

```
public static int tareas(int[] ts, int n) {
    int [] arrayIndicesOrdenados = ordenacionInsercion(ts);
    int [] tareas = new int[n];
    int sumaTotal = 0;
    int indiceProcesador = 0;

    for (int i = 0; i < ts.length; i++) {
        if(indiceProcesador < n) {

            tareas[indiceProcesador] += ts[arrayIndicesOrdenados[i]]

            sumaTotal += tareas[indiceProcesador];
            indiceProcesador++;
            i++;
        }else indiceProcesador = 0
    }
    System.out.println("Suma final = "+ sumaTotal);
    return 0;
}
```

Figura 2: Análisis de complejidad temporal del algoritmo voraz realista

T(n) = 11 + $14n^2$-n-6 + $\sum_{i=0}^{n}$ (6 + 2 + 5 + 3 + 2 + 2 + 2) + 1 +1

T(n) = 11 + $14n^2$-n-6 + 22n +2

T(n) = $14n^2$ + 21n + 7

Siendo una complejidad de notación O(n^2).

Análisis de la complejidad espacial del algoritmo

int [N] tareas = 4 bytes * N;

int [M] ts = 4 bytes * M;

int [M] arrayIndicesOrdenados = 4 bytes * M;

int n, sumaTotal, indiceProcesador = 12 bytes;

Obteniéndose una complejidad espacial = 4 bytes * N + 8 bytes * 2M * + 12.

Siendo una complejidad espacial de notación O(n).

Algoritmo inserción

Añado este algoritmo para mostrar su complejidad temporal, necesaria en la complejidad temporal del algoritmo voraz realista.

```java
public static int[] ordenacionInsercion(int[] v) {
    int[] v2 = new int[v.length];
    v2[0] = 0;
    for (int i = 1; i < v.length; i++) {
        int aux = v[i];
        int j;
        for (j = i - 1; j >= 0 && v[v2[j]] > aux; j--)
            v2[j + 1] = v2[j];
        v2[j + 1] = i;
    }

    return v2;
}
```

Figura 3: Algoritmo de ordenación por inserción

Análisis de la complejidad espacial del algoritmo:

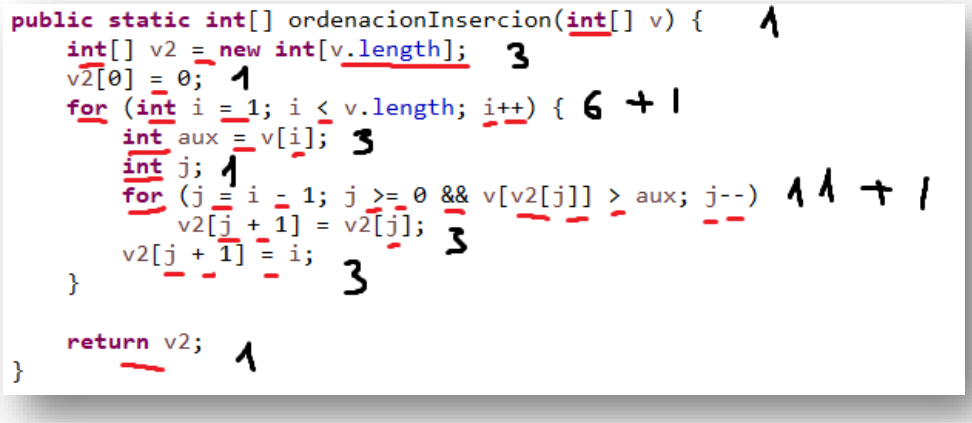


Figura 4: Análisis de complejidad temporal del algoritmo de ordenación por inserción

$$T(n) = 14n^2\text{-}n\text{-}6$$

Obteniéndose una complejidad temporal de notación O(n^2).

Conclusiones

Pienso que he obtenido complejidades muy altas y dándole más vueltas podría haber reducido estas. La complejidad temporal en esencia es la más crítica, ya que el almacenamiento no es tan limitado como el tiempo, uno se puede fabricar y otro no, por lo que a pesar de tener un gasto importante al generar un array bidimensional opino que tampoco es crítico y siempre es aceptable si ayuda a comprender y estructurar el código (siempre dentro de unos límites).