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

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

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

Por tanto, 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.

Ordenamiento por inserción

```
public static int[] ordenamientoInsercion(int[] array) {

      int index, aux;

      for    (int    i    =    1;    i    <    array.length;    i++)    {
           index = array[i];
        aux = i - 1;

           while    (aux    >=    0    &&    array[aux]    >    index)    {
                     array[aux      +      1]      =      array[aux];
                              aux        =        aux        -        1;
       }
       array[aux + 1] = index;
    }
           return array;
}
```

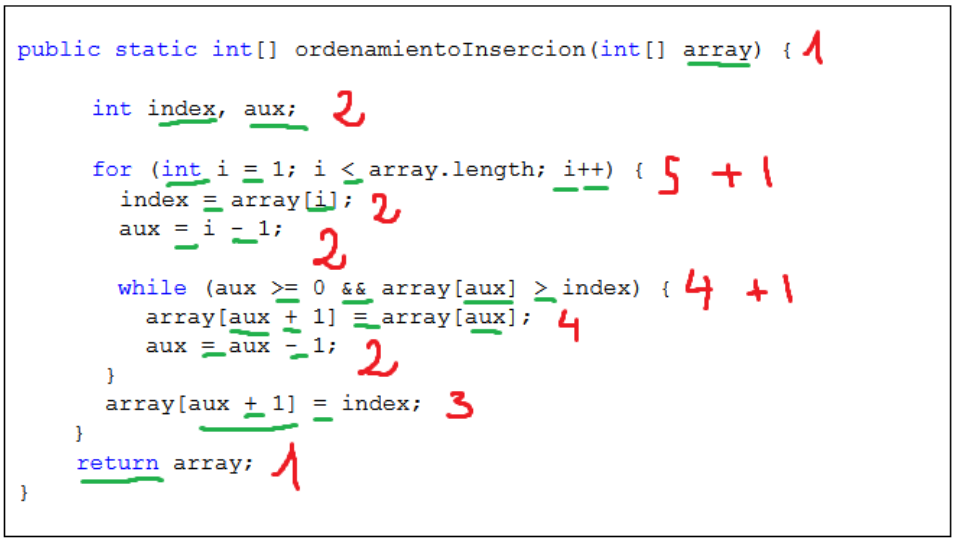
Fragmento de código 1: Ordenamiento por inserción

Se utiliza un el ordenamiento por inserción debido a que era obligatorio, pero se podrían haber utilizado otras variantes como Quicksort para obtener menor complejidad temporal.

Este algoritmo selecciona al inicio un punto del array para pivotar, en el caso de arriba se utiliza la posición número 1, representado como **index**, a continuación, se utiliza la variable **aux** como un puntero a la posición anterior a index.

Por último, llega la parte esencial de este ordenamiento, el bucle **while** donde se comprueba si **aux** está en uno de los límites del array (0) y si el valor que hay en la casilla apuntada por **aux** es mayor que el valor de la casilla apuntada por **index**, en caso de que se cumpla, el valor de la casilla de **aux** se mueve a la casilla donde se inició **index**, a continuación, se decremente en uno el valor de **aux** y se vuelve a repetir el proceso. Una vez no se cumpla, el valor inicial guardado en la variable **index** toma lugar en un hueco en el cual su valor a la izquierda es o bien el principio del array o un número menor, y a su derecha un número mayor, ya que han sido previamente desplazados en el bucle **while** (también podría ser el límite derecho del array).

Análisis de la complejidad temporal del algoritmo:



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

T(n) = 1 + 2 + $\sum_{i=1}^{n}$ (5 + 4 + T(while) + 3) + 1 + 1

T(while) = $\sum_{i=1}^{n-1}$ (4 + 4 +2) + 3 + 1 = 8 (n – 1) + 4 = 8n – 8 + 4 = 8n – 4

Por lo cual:

T(n) = 1 + 2 + $\sum_{i=1}^{n}$ (5 + 4 + 8n – 4 + 3) + 1 + 1 =

T(n) = 3 + n (8n + 8) + 2 = $8n^2$ + 8n + 5

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

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] array = 4 bytes * N;

Int index, aux = 8 bytes;

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

Siendo una complejidad de notación O(n).

Suma de tiempos

```
public static int sumaTiempos(int[] [] array, int n, int tareasPorProcesador) {

int suma = 0;
	int auxSuma, aux, j;
	for (int i = 0; i < n; i++) {
		aux = 0;
		auxSuma = 0;
		j = 0;
		while (j < tareasPorProcesador) {
			if (array[i][j] > 0) {
				auxSuma += array[i][j] + aux;
				aux = array[i][j] + aux;
			 }
			 j++;
		}
		suma += auxSuma;
	}
	return suma;
}
```

Fragmento de código 2: Suma de los tiempos totales de las tareas

En este apartado se cubre el algoritmo que realiza la suma de los tiempos para obtener el mejor resultado.

Se utiliza un array bidimensional que almacena cada procesador y cada uno de estos guarda los números (tiempos) que tiene que tratar.

Se recorre el array en orden ascendente, procesador por procesador. Se utilizan variables auxiliares y una final llamada suma, que almacena la suma de todas las sumas de tiempos de cada procesador.

Las variables **aux, auxSuma** y **j** se inicializan a 0 dentro del bucle for porque interesa que su valor se establezca de nuevo a 0 en cada nueva iteración de un procesador.

La variable **aux** sirve para almacenar el valor extra que se le tiene que sumar al siguiente número. Por ejemplo:

Se tiene una lista con valores {1,3, 5} y se realizan las iteraciones:

1º: 1

2º: 1 + (1 + 3)

3: 1 + (1 + 3) + (1 + 3 + 5)

A partir del segundo caso se puede apreciar como el valor teñido en azul es el mismo que se tiene que sumar de manera adicional al siguiente valor en la lista.

La variable **auxSuma** almacena la suma de los tiempos de manera fragmentada, tomando solo los nuevos valores, es decir, tomando el ejemplo anterior, primero suma 1, posteriormente (1+3) y finalmente (1+3+5).

Dando como resultado la suma total final de este procesador:

auxSuma = 1 + (1+3) + (1+3+5) = 14

Por último, la variable **j** se utiliza como centinela en el bucle.

Dentro del bucle **while** existe una condición if que comprueba si un valor es mayor que 0. Esto es debido a que en caso de obtener un valor impar en la relación tareas procesadores, se crea espacio extra para todos los procesadores, que cuenta con valor 0. Además, el valor de tiempo de una tarea siempre debe ser ti > 0, si no, quiere decir que o bien el tiempo que se tarda es negativo, o si es 0, la tarea ya está completada de base.

Análisis de la complejidad temporal del algoritmo:

```
public static int sumaTiempos(int[] [] array, int n, int tareasPorProcesador) {

int suma = 0;
      int auxSuma, aux, j;
      for (int i = 0; i < n; i++) {
            aux = 0;
            auxSuma = 0;
            j = 0;
            while (j < tareasPorProcesador) {
                  if (array[i][j] > 0) {
                        auxSuma += array[i][j] + aux;
                        aux = array[i][j] + aux;
                  }
                  j++;
            }
            suma += auxSuma;
      }
      return suma;
}
```

T(n) = 3 + 2 + 3 $\sum_0^n$ (5 + 3 + T(while) + 2) 1 + 1

T(while) = $\sum_0^n$ (1 + 2 + 4 + 3 + 2) + 1 = 12n + 1

Por lo cual:

T(n) = 8 + n (8 + (12n + 1) + 2) + 2

T(n) = 8 + 11 n + 12n^2 + 2

T(n) = 12n^2 + 11n + 10

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

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 suma, auxSuma, aux, j, n, tareasPorProcesador = 6 * 4 bytes = 24 bytes.

Int [N] [M] array = N * M * 4 bytes

Total = N * M + 28 bytes

(Al ser un array bidimensional se obtienen valores cuadráticos en la complejidad espacial).

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

Función Tareas

```
public static int tareas(int[] ts, int n) {
      int tareasPorProcesador = ts.length / n;
      if ((ts.length % n)! = 0)
            tareasPorProcesador++;
      int[] [] procesadores = new int[n][tareasPorProcesador];
      int indiceTareas = 0;
      int i = 0;
         while(i < ts.length){
           for (int j = 0; j < n && i < ts.length; j++){
                procesadores[j][indiceTareas] = ts[i];
                i++;
           }
           indiceTareas++;
      }
      int suma = sumaTiempos(procesadores, n, tareasPorProcesador);
       System.out.println("Suma final = " + suma);
     return 0;
 }
```

Fragmento de código 3: Se establecen los índices de las tareas a cada procesador

La función inicial del programa, encargada de generar el tamaño del array de los procesadores, el array bidimensional utilizado de manera auxiliar para facilitar el seguimiento del programa y la introducción de valores a este.

El motivo por el cual se comprueba que " i < ts.length " dos veces es porque a pesar de que todos los arrays de procesadores tienen la misma cantidad de celdas, no todos ellos deben estar completos.

Análisis de la complejidad temporal del algoritmo:

```
public static int tareas(int[] ts, int n) {
      int tareasPorProcesador = ts.length / n;
      if ((ts.length % n)! = 0)
            tareasPorProcesador++;
      int[] [] procesadores = new int[n][tareasPorProcesador];
      int indiceTareas = 0;
      int i = 0;
         while(i < ts.length)
            for (int j = 0; j < n && i < ts.length; j++){
                  procesadores[j][indiceTareas] = ts[i];
                  i++;
            }
            indiceTareas++;
      }
      int suma = sumaTiempos(procesadores, n, tareasPorProcesador);
      System.out.println("Suma final = " + suma);
     return 0;
 }
```

T(n) = 2 + 3 +2 + 2 + 4+ 2 + 2 + $\sum_{i=0}^{n}$ (T(for) + 2) + 1 + 2 + T(sumaTiempos) + 1

T(for) = $\sum_{i=0}^{n}$ (7 + 3 +2 +1) = 13n

T (n) = 17 + n(13n +2) + 3 + 12n^2 + 11n + 10 + 1

T(n) = 25n^2 + 13n + 31

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

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 [X] ts = X * 4 bytes

Int [N] [M] procesadores = N * M * 4 bytes

Int n, tareasPorProcesador, indiceTareas, i, j , suma = 5 * 4 bytes = 20 bytes

Total = X + N * M + 28 bytes

(Al ser un array bidimensional se obtienen valores cuadráticos en la complejidad espacial).

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

Main

```
public static void main(String[] args) {

    tareas(ordenacionInsercion(ts.clone()), 2);

}
```

Imagen de código 1: Main del programa

Adicionalmente se añade la función main, la cual invoca a tareas, la cual obtiene una copia del array que solicita el usuario, pero ya ordenado, junto con el número de procesadores.

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).