Instituto Politécnico Nacional

Escuela Superior de Cómputo

Sesión 7: Secuencia de intervalos de peso máximo

Integrantes:

Escutia López Arturo

López Santiago Daniel

30/OCT/2016

Programación dinámica (Dynamic Programming)

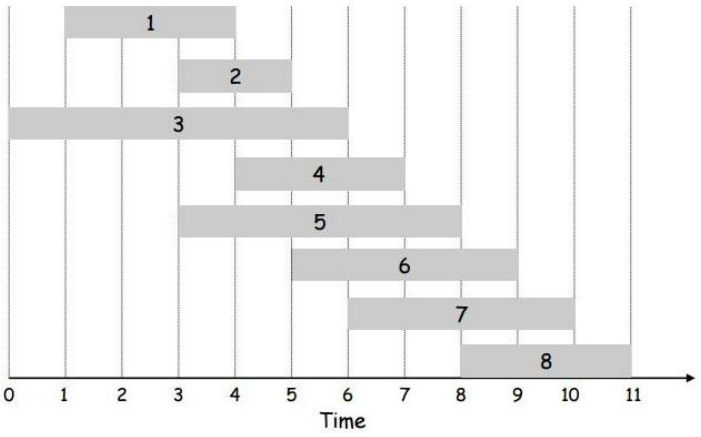
[1]Es una técnica utilizada para problemas que si los resolviéramos de una manera intuitiva con un algoritmo, su complejidad seria de forma exponencial, por lo cual con programación dinámica estos algoritmos reducen su complejidad en O(n^2) o de O(n^3). La idea tras la programación dinámica tiene cierta similitud tras la de divide and conquer, consiste en dividir un problema grande en un número determinado de subproblemas, resolvemos los subproblemas y unimos cada resultado para dar una solución total del problema original.[2] La diferencia que radica es que en DP las soluciones a los subproblemas forzosamente deben de ser las más óptima evitando realizar el cálculo de problemas resueltos con anterioridad en otras llamadas recursiva.

[3]En esta práctica se aplicó éste método en la resolución de problema conocido como Secuencia de intervalos de peso máximo (Weighted Interval Scheduling).Éste problema consiste en encontrar el conjunto con el peso máximo que no tenga traslapes en los trabajos,los cuales poseen un tiempo de inicio y un tiempo final. Para esto se realizaron los siguientes pasos.

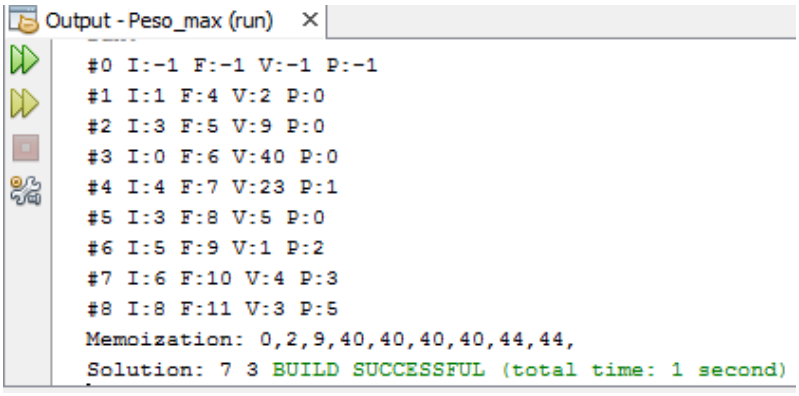
1. Implementar una función que tome los tiempos finales de cada intervalo y los ordene, esto debe hacerse con un algoritmo de complejidad O(n log n)
2. Implementar una función que calcule p(j), esto debe hacerse con un algoritmo de complejidad O(n log n)
3. Implementar una función que calcule la solución óptima para cada intervalo y la almacene en un arreglo.
4. Considerando todo lo anterior intégralo para obtener el conjunto de intervalos de peso máximo.

Tomando en cuenta el siguiente ejemplo y programando los pasos anteriores se obtiene la siguiente salida

Ex: p(8) = 5, p(7) = 3, p(2) = 0.



j	p(j)	
0	-	
1	0	2
2	0	9
3	0	40
4	1	23
5	0	5
6	2	1
7	3	4
8	5	3



Se imprime en el programa el conjunto de intervalos con su respectiva duración y Pj ordenados de manera ascendente con respecto a los tiempos finales, el resultado de los valores óptimos del problema con respecto a las peticiones de trabajo y el conjunto de intervalos que forman parte de la solución.

```
private static void FindS(int j){
    if(j==0) return;
    else if(sorted_end.get(j).getV()+M[sorted_end.get(j).getP()] > M[j-1]){
        System.out.print(j+" ");
        FindS(sorted_end.get(j).getP());
    }
    else
        FindS(j-1);
}

private static int OPT(int j){
    if(j==0) return 0;
    if (M[j]==-1)       //calculate optimal of j if it hasn´t calculated
       M[j]=max(sorted_end.get(j).getV()+OPT(sorted_end.get(j).getP()) , OPT(j-1));   //calculate Vj+ Opt[Pj]   and Opt[j-1]
    return M[j];
}

private static void CalculateP(int low,int high, Interval i){
    if(i.getV()<1)  return;
    int middle;
    //finding lower bound with binary search
while(low<high){
        middle=low+(high-low+1)/2;
        if(sorted_end.get(middle).getF()<=i.getS())    //if middle finishes before i starts,search  to the right
            low=middle;                                //for a nearest not overlapping interval for  i
        else
            high=middle-1;              //search to the left for not overlapping interval
    }
    if (sorted_end.get(low).getF()>i.getS())   //there isn't any not overlapping interval for i
        i.setP(0);
    else
        i.setP(low);            //nearest interval that  doesn't overlap i
}
```

Ésta es la parte del código donde se lleva a acabo los últimos 3 pasos, el primero es realizar un ordenamiento, para el cual fue utilizado mergesort que como sabemos es de complejidad n log n.

Posteriormente se manda a llamar la función CalculateP, ésta función lo que va a hacer es que por cada intervalo va dividiendo a la mitad el arreglo ordenado hasta que ya no se pueda seguir dividiendo. Si el intervalo middle termina antes que empiece el intervalo i,es un intervalo que no se traslapa pero no sabemos si es el intervalo más próximo al que recibe la función, es por eso que seguimos buscando desde middle-high, en caso de que se traslape procedemos a buscar desde low-middle . Esto se le conoce como un lower bound, el cual es una variación de la búsqueda

binaria, como sabemos la búsqueda binaria es de orden log n pero como lo hacemos para todo conjunto de intervalos la complejidad de la función es de nlogn.

En la función OPT pasamos a calcular los valores óptimos, para evitar calcular varias veces un mismo subproblema solamente en la llamada recursiva verificamos si el resultado de ese subproblema ya fue almacenado en el arreglo, en dado caso se retorna el valor,sino se calcula. Gracias a esto obtenemos una complejidad de O(n) ya que por cada llamada a la función solo toma O(1), y como sabemos solo se calcula 1 vez por cada elemento en el conjunto de intervalos que es de tamaño n .

Y para imprimir el resultado solo requerimos de <=n llamadas recursivas para recorrer el arreglo por lo cual también posee una complejidad de O(n)

Retormando :

1.-MergeSort O(nlogn)

2.-Pj O(nlogn)

3.-OPT O(n)

4.-Find Solution O(n)

Por lo tanto MergeSort O(nlogn)+ Pj O(nlogn)+ OPT O(n)+ Find Solution O(n)= O(n log n)

Referencias

- [1] Algorithm Design by Éva Tardos and Jon Kleinberg, 2005 https://www.cs.princeton.edu/courses/archive/spr05/cos423/lectures/06dynamic-programming.pdf
- [2]Advanced Algorithms Lecture 3: Dynamic Programming, http://faculty.cse.tamu.edu/nikolova/Teaching/Algorithms_Graduate_fall2012/lecture_notes/lec3.pdf
- [3]Dynamic Programming(Weighted Interval Scheduling), http://www.sfu.ca/~arashr/lecture16.pdf