Taller 2 - Análisis de Algoritmos

Sergio Andrés Mejía Tovar

28 de septiembre de 2020

Resumen

En este documento se presenta la solución al segundo taller de la clase de Análisis de Algoritmos, desarrollando un análisis del problema de la mínima suma de la partición binaria de dos subconjuntos y buscándole solución por medio de un algoritmo de Programación Dinámica.

Parte I
Partición binaria de un conjunto cuya diferencia de sus sumas sea mínima

1. Análisis y diseño del problema

1.1. Análisis

El problema se puede definir de manera informal como la búsqueda de dos subconjuntos resultantes de la partición binaria de un conjunto C que abarquen completamente los elementos de C tal que la diferencia entre la suma de los elementos de los subconjuntos sea mínima. Así, Sea C una conjunto de tamaño n definido del siguiente modo:

$$C = \{x \mid x \in \mathbb{R}\}$$

Se buscan dos subconjuntos C_1 y C_2 cuya diferencia de sumas sea mínima, tal que:

$$C_1 \cup C_2 \equiv C \wedge \text{mín}\left(\left|\sum_{i=0}^{|C_1|} C_{1i} - \sum_{j=0}^{|C_2|} C_{2j}\right|\right)$$

1.2. Diseño

Con las observaciones realizadas en el análisis anterior, podemos escribir el diseño de un algoritmo que solucione el problema de la partición binaria de un conjunto cuya diferencia de sus sumas sea mínima.

- **Entradas:**
  1. Un conjunto $C = \{x \mid x \in \mathbb{R}\}$ de $n \in \mathbb{N}$ elementos.
- **Salidas:**
  1. Dos subconjuntos C_1, C_2 tal que $C_1 \cup C_2 \equiv C$ y que cumplan $\text{mín}\left(\left|\sum_{i=0}^{|C_1|} C_{1i} - \sum_{j=0}^{|C_2|} C_{2j}\right|\right)$

2. Algoritmos

2.1. Programación Dinámica

2.1.1. Algoritmo

Este algoritmo nace de la idea de utilizar la solución de problemas parciales para calcular la solución del problema total. Esto se realizará por medio de una Matriz de Memoización (que se denominará M) de dos dimensiones que indexará en cada uno de sus elementos la solución a un problema parcial. Se decidió trabajar como la definición de un problema parcial lo siguiente: Insertar un elemento a uno de los dos subconjuntos, los cuales pueden ya tener elementos. Para indexar en las filas se tomará el índice que define el elemento a insertar a continuación en uno de los dos subconjuntos (teniendo en cuenta también el momento en que ya no se agregarán más elementos, definido en la fila $i = 1$ y los elementos del conjunto C_i se indexan en las filas $i > 2$), y se indexará en la otra dimensión por la actual diferencia (en valor absoluto) entre las sumas de los valores de los dos subconjuntos.

Así, se tiene una matriz $M : \mathbb{R}^{n \times m}, n = |C| + 1, m = 1 + 2\sum_{i=0}^{|C|} |C_i|$. El valor de m es el definido puesto que en el peor de los casos parciales, todos los elementos pertenecen a uno de los subconjuntos, y se consideró el doble puesto que la diferencia puede ser positiva o negativa (aunque se considerará el valor absoluto para los resultados). Para calcular el valor $M_{ij}, i \neq 1$ se calcula del siguiente modo $M_{ij} = \text{mín}\left(M_{(i-1)(j+C_i)}, M_{(i-1)(j-C_i)}\right)$. Los casos bases son $i = 1 \Rightarrow M_{ij} = |S - j|$ debido a que cuando no se agrega ningún elemento, la mínima diferencia es la actual. La solución del problema total se encontrará en $M_{(|C|+1)S}$.

Además, se generará una tabla de backtracking T para poder, después de calcular la tabla, hallar la configuración de C_1 y C_2 tal que sean los conjuntos que generan la solución. En esta tabla se guardará a qué subconjunto se agregó el elemento para la solución del problema parcial. En términos de implementación se trabajó con booleanos True y False $\in \mathbb{B}$, True cuando se agrega al subconjunto C_1 y False para el subconjunto C_2.

Algorithm 1 Partición binaria de un conjunto tal que la diferencia de las sumas de los dos subconjuntos sea mínima

```
1: procedure MinDifferencePartition(C)
2:     n ← |C|
3:     S ← 1 + Σ_{i=0}^{|C|} |C_i|
4:     let M : ℝ^{(n+1)×(2S+1)}
5:     let T : 𝔹^{(n+1)×(2S+1)}
6:     for j ← 1 to 2S + 1 do
7:         M[1][j] ← |j − S|
8:     end for
9:     for i ← 2 to n + 1 do
10:        for j ← 2S + 1 downto 1 do
11:            n1 ← ∞ ∧ n2 ← ∞
12:            R ← (j + A[i − 1] + 1) ∧ L ← (j − A[i − 1] + 1)
13:            if L ∈ [1, 2S] then
14:                n1 ← M[i − 1][L]
15:            end if
16:            if R ∈ [1, 2S] then
17:                n2 ← M[i − 1][R]
18:            end if
19:            if n1 < n2 then
20:                M[i][j] ← n1
21:                T[i][j] ← False
22:            else
23:                M[i][j] ← n2
24:                T[i][j] ← True
25:            end if
26:        end for
27:    end for
28:    let C_1 ← {}
29:    let C_2 ← {}
30:    j ← S
31:    for i ← n + 1 downto 2 do
32:        if T[i][j] = True then
33:            C_1 ← C_1 ∪ {A[i − 1]}
34:            j ← j + A[i − 1] + 1
35:        else
36:            C_2 ← C_2 ∪ {A[i − 1]}
37:            j ← j − A[i − 1] + 1
38:        end if
39:    end for
40:    return (M[n + 1][S], C_1, C_2)
41: end procedure
```

2.1.2. Complejidad

El algoritmo MinDifferencePartition tiene órdenes de complejidad $O(nm)$ debido a que para hallar la solución al problema es necesario calcular todos los valores de la Matriz de Memoización M y para esto se recorren todos sus elementos y se hace la operación correspondiente. Se puede ver que $\Omega(1)$ cuando $|C| = 0$. Ambos órdenes de complejidad son calculados por inspección de código.

2.1.3. Invariante

La variable i, j que indexan la Matriz de Memoización M indican que para el estado M_{ij} ya se encuentran calculadas todas las soluciones previas (entiéndase previas como calculadas todas las posiciones de las filas $1 \leq n \leq i-1$ y calculadas las posiciones $M_{im} \mid j+1 \leq m \leq S+1$.

- **Inicio:** Los elementos $M_{1j} = j$, pues representan los casos base del problema.
- **Avance:** Para cada elemento M_{ij} se calcula como $M_{ij} = \text{mín}\left(M_{(i-1)(j+C_i)}, M_{(i-1)(j-C_i)}\right)$, estos valores ya se encuentran calculados.
- **Terminación:** Cuando $i > n+1$, se calcularon todos los elementos de M y el programa terminará.

2.2. Detalles de Implementación

El presente taller traer consigo un archivo Java con el nombre `MinDifferencePartition.java`. Para compilarlo y ejecutarlo se necesita mínimo Java JDK 1.8 y se puede compilar por consola con las siguientes líneas:

```
javac MinDifferencePartition.java
java MinDifferencePartition
```

El programa contiene una clase pública `MinDifferencePartition` que contiene los múltiples métodos y una clase privada `Result` para encapsular los resultados. La función `+minDifferencePartition(int[] A) : Result` recibe un arreglo que representa el conjunto de datos y retorna los resultados en una clase `Result` que contiene dos ArrayList representando los subconjuntos, y un número entero con el resultado de la mínima diferencia (esta clase contiene los getters necesarios para acceder a sus valores). Cabe aclarar que si bien la entrada y las salidas están representadas como arreglos, el orden de sus elementos no interesa puesto que conceptualmente representa un conjunto.

El programa viene cargado con múltiples casos de prueba, y para cada caso entregará una respuesta que mostrará la mínima diferencia y los dos subconjuntos resultantes.