Diseño y Análisis de Algoritmos
Tarea 4

Sergio Montoya Ramírez

Contents

Chapter 1

Pregunta 1

1.1 Código

```java
/* Dynamic Programming Java implementation of Coin
Change problem */
import java.util.Arrays;

class CoinChange {
  static long count(int coins[], int n, int sum)
  {
    // dp[i] will be storing the number of solutions for
    // value i. We need sum+1 rows as the dp is
    // constructed in bottom up manner using the base case
    // (sum = 0)
    int dp[] = new int[sum + 1];

    // Base case (If given value is 0)
    dp[0] = 1;

    // Pick all coins one by one and update the dp[]
    // values after the index greater than or equal to the
    // value of the picked coin
    for (int i = 0; i < n; i++)
    for (int j = coins[i]; j <= sum; j++)
    dp[j] += dp[j - coins[i]];

    return dp[sum];
    }

    // Driver Function to test above function
    public static void main(String args[])
    {
      int coins[] = { 1, 2, 3 };
      int n = coins.length;
      int sum = 5;
      System.out.println(count(coins, n, sum));
    }
    }
    // This code is contributed by Pankaj Kumar
```

1.2 Recurrencia

En este caso, la recurrencia que utiliza por cada numero menor a sum, sumar la cantidad de maneras en las que se puede llegar a sum desde este numero. Esto lo hace en cada paso sumando la cantidad de maneras que se puede llegar a la resta entre un sum y el numero en el que se esta.

1.3 Explicación

1. $coins$: Este es el array en donde están las monedas. Básicamente nos dice que números podemos acceder.
2. n : Este es el tamaño del Array en donde están las monedas. Básicamente, nos dice cuantos números distintos vamos a poder combinar.
3. sum: Este es el objetivo al que debemos llegar.
4. dp: Este es un array que va a guardar todas las maneras en las que se pueda llegar a el numero i. En particular $dp[i]$ es este numero de veces.
5. i : Este es un contador que nos permite revisar el valor por cada moneda. Es decir, vamos a tener que iniciar el proceso recursivo en alguna parte y esta variable nos permite iniciar desde el valor de cada una de las monedas que están en el array $coins$.
6. j: Esta es la variable que va a ir cambiando, básicamente cada ciclo lo que va a hacer es iniciar desde el valor que tenga $coins[i]$ y a partir de ahí recorrerá todos los valores menores a sum sumando las posibilidades que tiene la diferencia entre j y la moneda que se tiene.A

1.4 Diagrama Necesidades

Es difícil dibujar el diagrama de necesidades pues depende seriamente de las monedas que tengamos. Basicamente para cada punto se debe sumar de todos los puntos anteriores de diferencia con la moneda que se tenga. El diagrama que aqui se va a exponer va a mostrarse como si solo ubiera dos monedas $1, 2$ pero es sumamente dependiente de la cantidad y el valor de cada moneda.

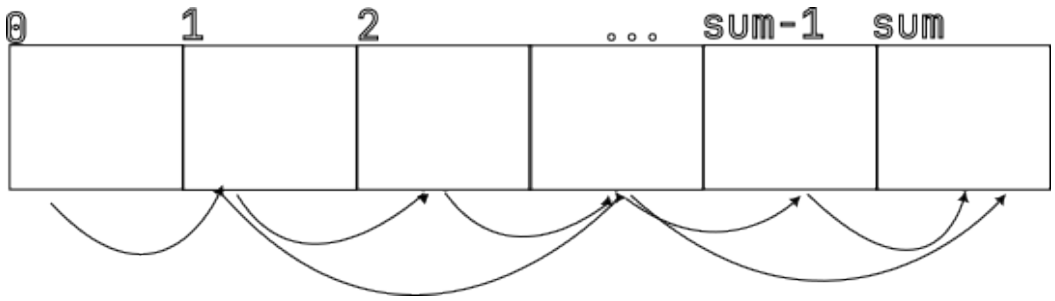

Figure 1.1: Diagrama de necesidades con dos monedas.

1.5 Complejidad Temporal

Revisemos linea a linea:

```
static long count(int coins[], int n, int sum) {
  int dp[] = new int[sum + 1]; // C1 | 1
  dp[0] = 1; // C2 | 1
  for (int i = 0; i < n; i++) // C3 | n
  for (int j = coins[i]; j <= sum; j++) // C4 | n*sum
  dp[j] += dp[j - coins[i]]; // C5 | n*sum
  return dp[sum]; // C6 | 1
  }
```

Con lo cual el total queda:

$$\begin{aligned} T(n, sum) &= C_1 + C_2 + C_3 \cdot n + C_4 \cdot n \cdot sum + C_5 \cdot n \cdot sum + C_6 \\ &= O(n \cdot sum). \end{aligned}$$

1.6 Complejidad Espacial

En este caso, vamos a utilizar no solo las variables normales que son constantes si no ademas, se hará uso de un array con tamaño $sum + 1$. Por lo tanto, la complejidad espacial es:

$$E(n, sum) = O(sum)$$

.

1.7 Ejemplos

```
@Test
public void primerTest() {
  int[] coins = {1, 2, 3, 4, 5};
  int n = coins.length;
  int sum = 20;
  long manerasSumar20 = CoinChange.count(coins, n, sum);
  assert manerasSumar20 == 192: String.format(
  "Algoritmo: %d | Respuesta: %d",
  manerasSumar20,
  192);
}

@Test
public void segundoTest() {
  int[] coins = {1, 2, 5, 10, 20, 50, 100};
  int n = coins.length;
  int sum = 200;
  long manerasSumar200 = CoinChange.count(coins, n, sum);
  assert manerasSumar200 == 73681: String.format(
  "Algoritmo: %d | Respuesta: %d",
  manerasSumar200,
  73681);
}
```

Chapter 2

Pregunta 2

2.1 Código

```java
// Java Solution

public class LongestPalinSubstring {

  // A utility function to print
  // a substring str[low..high]
  static void printSubStr(
  String str, int low, int high)
  {
    System.out.println(
    str.substring(
    low, high + 1));
  }

  // This function prints the longest
  // palindrome substring of str[].
  // It also returns the length of the
  // longest palindrome
  static int longestPalSubstr(String str)
  {
    // Get length of input string
    int n = str.length();

    // table[i][j] will be false if
    // substring str[i..j] is not palindrome.
    // Else table[i][j] will be true
    boolean table[][] = new boolean[n][n];

    // All substrings of length 1 are palindromes
    int maxLength = 1;
    for (int i = 0; i < n; ++i)
    table[i][i] = true;

    // Check for sub-string of length 2.
    int start = 0;
    for (int i = 0; i < n - 1; ++i) {
      if (str.charAt(i) == str.charAt(i + 1)) {
        table[i][i + 1] = true;
        start = i;
```

```
      maxLength = 2;
    }
  }

  // Check for lengths greater than 2.
  // k is length of substring
  for (int k = 3; k <= n; ++k) {

    // Fix the starting index
    for (int i = 0; i < n - k + 1; ++i) {

      // Get the ending index of substring from
      // starting index i and length k
      int j = i + k - 1;

      // Checking for sub-string from ith index to
      // jth index if str.charAt(i+1) to
      // str.charAt(j-1) is a palindrome
      if (table[i + 1][j - 1]
      && str.charAt(i) == str.charAt(j)) {
        table[i][j] = true;

        if (k > maxLength) {
          start = i;
          maxLength = k;
        }
      }
    }
  }
  System.out.print("Longest palindrome substring is: ");
  printSubStr(str, start,
  start + maxLength - 1);

  // Return length of LPS
  return maxLength;
}

// Driver code
public static void main(String[] args)
{
  String str = "forgeeksskeegfor";
  System.out.println("Length is: " + longestPalSubstr(str));
}
}

// This code is contributed by Sumit Ghosh
```

2.2 Recurrencia

Básicamente lo que hace es primero revisa todos los palíndromos que hay de tamaño 2 y los pone en la matriz. Luego, revisa cada uno de esos y va expandiendo con respecto al tamaño. Es decir que va a probando con substratos cada vez mas grandes.

2.3 Explicación

1. n: tamaño completo del string.
2. $table$: matriz $n \times n$ en la cual la posición $table[i][j]$ es verdadero si el substring $string[i..j]$ es palíndromo.
3. $maxLength$: Máximo tamaño del substring que se esta buscando. También pasa eventualmente a ser el valor mas grande de un substring que es palíndromo.
4. $start$ punto en el que inicia la búsqueda de palíndromos

Sin embargo, esto no deja muy claro el como funciona como tal. Por lo tanto miraremos con un poco mas de profundidad el siguiente loop.

```
for (int k = 3; k <= n; ++k) {
  for (int i = 0; i < n - k + 1; ++i) {
    int j = i + k - 1;
    if (table[i + 1][j - 1]
    && str.charAt(i) == str.charAt(j)) {
      table[i][j] = true;

      if (k > maxLength) {
        start = i;
        maxLength = k;
      }
    }
  }
```

Este loop básicamente va iterando desde 3 (que es el palíndromo mas pequeño que no se a revisado previamente) y mira si existe un palíndromo anterior que este yuxtapuesto a esta posición y repite este proceso hasta que reviso todos los posibles substrings.

2.4 Diagrama Necesidades

2.5 Complejidad Temporal

Revisemos linea a linea:

```
static int longestPalSubstr(String str)
{
  int n = str.length(); // C1 | 1
  boolean table[][] = new boolean[n][n]; // C2 | 1
  int maxLength = 1; // C3 | 1
  for (int i = 0; i < n; ++i) // C4 | n
    table[i][i] = true; // C5 | n
  int start = 0; // C6 | 1
  for (int i = 0; i < n - 1; ++i) { // C7 | n - 1
    if (str.charAt(i) == str.charAt(i + 1)) {  // C8 | n - 1
      table[i][i + 1] = true; // C9 | n - 1
      start = i; // C10 | n - 1
      maxLength = 2; // C11 | n - 1
    }
  }
  for (int k = 3; k <= n; ++k) { // C12 | n - 2
    for (int i = 0; i < n - k + 1; ++i) { // C13 | n^2
      int j = i + k - 1; // C14 | n^2
      if (table[i + 1][j - 1] // C15 | n^2
      && str.charAt(i) == str.charAt(j)) { // C16 | n^2
```

```
            table[i][j] = true; // C17 | n^2
            if (k > maxLength) { // C18 | n^2
              start = i; // C19 | n^2
              maxLength = k; // C20 | n^2
            }
          }
        }
      }
      System.out.print("Longest palindrome substring is: "); // C21 | 1
      printSubStr(str, start, // C22 | 1
      start + maxLength - 1); // C23 | 1
      return maxLength; // C24 | 1
    }
```

Por lo tanto la complejidad temporal seria:

$$\begin{aligned} T(n) &= C_1 + C_2 + C_3 + C_4 \cdot n + C_5 \cdot n + C_6 + \\ &\quad (C_7 + C_8 + C_9 + C_{10} + C_{11})(n-1) + C_{12} \cdot n + \\ &\quad (C_{13} + C_{14} + C_{15} + C_{16} + C_{17} + C_{18} + C_{19} + C_{20}) \cdot n^2 + C_{21} + C_{22} + C_{23} + C_{24} \\ T(n) &= O\left(n^2\right) \end{aligned}$$

.

2.6 Complejidad Espacial

En este caso todo es constante exceptuando la matriz *table* que de hecho tiene tamaño n^2 por lo tanto la complejidad espacial es:

$$E(n) = O\left(n^2\right).$$

2.7 Ejemplos

```
@Test
public void primerTest() {
  String str = "Esto es una prueba"
  int pal = LongestPalinSubstring.longestPalSubstr(str)
  assert pal == 1: String.format(
  "Algoritmo: %d | Respuesta: %d",
  pal,
  1);
}

@Test
public void segundoTest() {
  String str = "Para probarraborp tenemos hannah"
  int pal = LongestPalinSubstring.longestPalSubstr(str)
  assert pal == 14: String.format(
  "Algoritmo: %d | Respuesta: %d",
  pal,
  14);
}
```