Actividad 3

Materia: Estructura de Datos

Maestro: Adalberto Emmanuel Rojas Perea

Alumna: Isabel Kyra Romero Borunda

TECMILENIO

Índice

Introducción

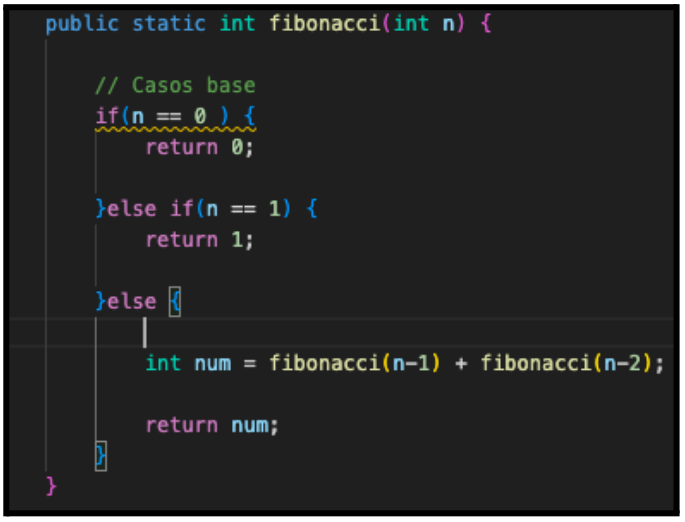
En esta actividad se usarán los conceptos de recursividad y backtracking para realizar diferentes algoritmos que se nos pidió. Siendo estos una serie de Fibonacci, uno para determinar si hay un subconjunto dentro de un conjunto que sumados den un objetivo, y por último un sudoku.

Implementación

Fibonacci

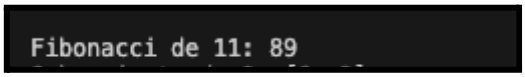
En esta parte se tiene que sacar el número 11 de la serie de Fibonacci, dicha serie empieza en 0 donde el siguiente número es la suma de los dos anteriores(0, 1, 1, 2, etc…).

Para que no se nos cicle la función tenemos que tener en cuenta los casos base, en estos casos son si n = 0 y n = 1. En caso de no ser estos dos números llegamos a donde se hace la recursividad donde num = fibonacci(n-1) + fibonacci(n-2).



Resultados

Se probó con diferentes números: 11, 5 y 3. Podemos ver en las imágenes el resultado de cada uno:



SubsetSum

Aquí se hizo una clase que regresa una lista con los subconjuntos que sumados nos dan el objetivo de uno conjunto.

Tenemos dos casos base, donde si el “objetivo == 0” nos regresa una lista vacía, en este caso solo se pensó en la suma de números enteros. Luego el segundo caso es **“i**(index de la lista) **== conjunto.length**(tamaño de la lista) **“** nos retorna null, porque sería el final de la lista.

```
public static List<Integer> SubsetSum(int[] conjunto, int objetivo,
int i) {

    if (objetivo == 0) {

        return new ArrayList<>();

    }else if(i == conjunto.length){
        return null;
    }else {
```

Luego si no suceden estos dos casos de arriba, usamos la recursividad para buscar los números que se pueden sumar. Esta parte funciona incluyendo un número a la lista que se le pueda restar al objetivo sin que llegue a 0, si cuando resta un número llega a ser menos de cero el número se excluye.

```
        // Intenta incluir el número actual
        List<Integer> incluido = SubsetSum(conjunto, objetivo -
        conjunto[i], i + 1);
        if (incluido != null) {
            incluido.add(conjunto[i]);
            return incluido;
        }

        // Intenta excluir el número
        return SubsetSum(conjunto, objetivo, i + 1);
    }
```

Resultados

```
// Subconjunto de conjunto
int[] o = {2,6,7,3,4};
System.out.println("Subconjunto de 5: " + SubsetSum( o,
objetivo:5, i:0));
```

Aquí se creó un array con varios números: 2, 6, 7, 3,4 y luego se probó con diferentes objetivos: 8, 5 y 15. Podemos ver en las imágenes de abajo que nos arroja listas con los subconjuntos que sumados nos dan el objetivo.

```
Subconjunto de 15: [7, 6, 2]
```

Sudoku

Se crearon 3 clases para el sudoku:

- **vacio**
  Nos dice si el espacio está vacío para agregar un número. Nos regresa las coordenadas -1, -1 cuando no hay un espacio vacío.

```java
public static int[] vacio (int[][] array) {
    for(int i = 0; i < array.length; i++) {
        for (int j = 0; j < array[i].length; j++) {
            if (array[i][j] == 0) {
                return new int[]{i, j};
            }
        }
    }
    return new int[]{-1,-1};
}
```

- **seguro**
  Nos dice si el lugar donde ponemos un número cumple con las condiciones. Se asegura de que no haya números repetidos en las filas y columnas del lugar, y que en los alrededores (cuadros de 3x3) no se encuentre el mismo número.

```java
public static boolean seguro(int[][] array, int fila, int col, int num) {
    // Checa la fila
    for(int i = 0; i< 9; i++) {
        if(array[fila][i] == num) {
            return false;
        }
    }
    // Checa columna
    for(int i = 0; i< 9; i++) {
        if(array[i][col] == num) {
            return false;
        }
    }

    // Checa si esta en el recuadro de 3x3
    int startRow = fila - (fila % 3), startCol = col - (col % 3);

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (array[i + startRow][j + startCol] == num) {
                return false;
            }
        }
    }

    return true;
```

- **sudoku**
  Es la que se encarga de llenar el sudoku utilizando las demás clases.
  Primero busca coordenadas de un lugar vacío, luego guarda esas coordenadas por separado. Luego coloca un número del 1-9 en ese lugar, si se cumplen las condiciones se usa la recursividad para volver a hacer todo. Si no se cumplen las condiciones al poner el número hace backtracking e intenta con otro número.

```java
public static boolean sudoku(int[][] sudoku) {
    // Buscar lugar vacio
    int[] coor = vacio(sudoku);  // coordenadas del lugar vacio
    if(coor[0] == -1)
        return true;
    ;
    int fila = coor[0];
    int col = coor[1];

    for(int num = 1; num<=9; num++) {
        // Cambia el numero si se cumplen las condiciones
        if (seguro(sudoku, fila, col, num)) {
            sudoku[fila][col] = num;

            if (sudoku(sudoku)) {
                return true; // solucionado
            }

            // retroceso
            sudoku[fila][col] = 0;
        }
    }

    return false;
}
```

Resultados

Primero creamos nuestro tablero de sudoku, y luego utilizamos la función de **sudoku.** Podemos ver en la segunda imagen como lo relleno.

```java
// Sudoku
int[][] cuadro = {
    {3, 0, 6, 5, 0, 8, 4, 0, 0},
    {5, 2, 0, 0, 0, 0, 0, 0, 0},
    {0, 8, 7, 0, 0, 0, 0, 3, 1},
    {0, 0, 3, 0, 1, 0, 0, 8, 0},
    {9, 0, 0, 8, 6, 3, 0, 0, 5},
    {0, 5, 0, 0, 9, 0, 6, 0, 0},
    {1, 3, 0, 0, 0, 0, 2, 5, 0},
    {0, 0, 0, 0, 0, 0, 0, 7, 4},
    {0, 0, 5, 2, 0, 6, 3, 0, 0}
};

sudoku(cuadro);
```

```
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
dfoh@MacBook-Air-de
```