Taller 4 (Estructuras de Datos y Algoritmos)

David Gómez y Simón Correa

1.1 Código y constantes

```
private static int arrayMaxAux(int[] arrya, int n, int target) {
    if(n==0) {
      return target;                                   //c_1= 3
    }else{
      target = Math.max(target, array[n]);              //c_2 = 6
      return arrayMaxAux(array, n-1, target);    //T(n) = c_2 + T(n-
1)
    }
 }
```

1.2 Tamaño del problema

El tamaño del problema, son los elementos que faltan por sumar del arreglo.

1.3 ecuación de recurrencia

T(n) = a. c_1 if n = 0

b. c_2+T(n-1) if n > 0

1.4 Ecuación con Wolfram Alpha

T(n) = c_2*n + c_1

1.5 Aplicar la notación O

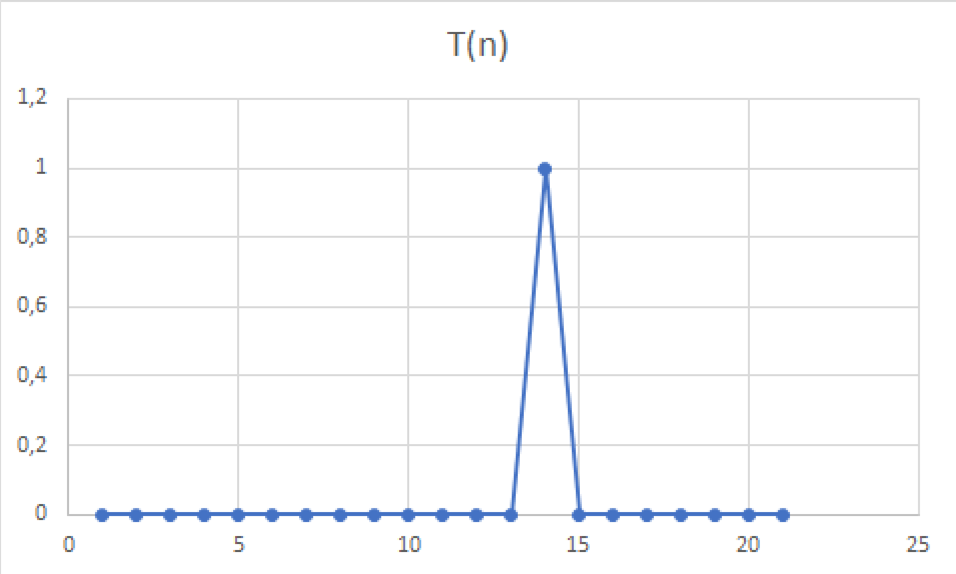
-T(n) es O (c_2*n + c_1) por definición de O

- T(n) es O (c_2* n) por regla de la Suma

- T(n) es O (n), por regla del producto

1.6 Complejidad

La complejidad que tiene el algoritmo trabajado para el peor de los casos, es decir para números muy grandes de n, o en otras palabras el arreglo tiene un mayor número de elementos, es de O(n)



Sin embargo, la complejidad calculada no coincide con los datos reales, tomando 20 tamaños diferentes del arreglo.

2.1 Código y Constantes

```
public static boolean groupSumAux(int start, int[] nums, int target)
{
        if (start == nums.length){ //
            return target==0;//c_1 = 4
        }else{
            return groupSumAux(start+1, nums, target-nums[start]) ||
groupSumAux(start+1, nums, target); //c_2 = 7
        } //T(n) = c_2 + T(n-1) + T(n-1)
    }
```

2.2 Tamaño del problema

El tamaño del problema es el tamaño del arreglo, es decir la cantidad de contenedores o cajas que hay que acomodar dentro del espacio

2.3 Ecuación de recurrencia

T(n) = a. c_1 if start = nums.length

c. c2 + T(n-1) + T(n-1) if n > 0

2.4 Ecuación con Wolfram Alpha

T(n) = c_2 (2^n - 1) + c_1 2^(n - 1)

2.5 Aplicar notación O

O (c_2 (2^n - 1) + c_1*2^(n - 1)) Por definición de O

O (c_2 *2^n – c_3 + c_1*2^(n - 1)) Por algebra

O (c_2 *2^n + c_1*2^(n - 1) – c_3)   Por algebra

O (c_2 *2^n + c_1*2^(n - 1))   Por regla de la suma

O (c_2 *2^n + c_1*2^n*2^(-1)) Por algebra

O (2^n +2^n) Por regla del producto

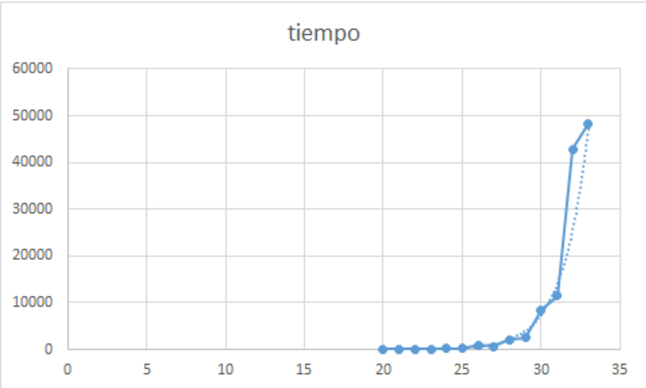
O(2*2^n) Por algebra

O(2^n) Por Regla del producto

2.6 Complejidad

La complejidad del algoritmo cuando n tomo los peores valores, es decir cuando hay que ingresar una cantidad de paquetes muy grande al contenedor, la ecuación adquiere una complejidad de

O (2 ^n).



Podemos observar que al realizar un testeo del tiempo de ejecución del algoritmo en función de n, los tiempos arrojados corresponden a la ecuación de complejidad O (2^n)

3.1 Código y Constantes

```
public static long fibonacci(int n) {
     if (n==1 || n==2)
          return 1;    //c_1 = 5
       else if (n==0)
          return 0;    //c_2 = 3
       else
          return fibonacci(n-1) + fibonacci(n-2);        //c_3 = 5
}
```

3.2 Tamaño del problema

El tamaño del problema depende del n-esimo termino que el usuario pida calcular de la secuencia de fibonacci

3.3 Ecuación de recurrencia

T(n) =   a. c_1 if n=1 && if n=2

b. c2   if n=0

c. c3 + T(n-1) + T(n-2)   if n >2

3.4 Ecuación con Wolfram Alpha

$T(n) = c_1 F_n + c_2 L_n$ (where c_1 and c_2 are arbitrary parameters)

(F_n is the n^th Fibonacci Number)

(L_n is the n^th Lucas Number)

3.5 Aplicar notación O

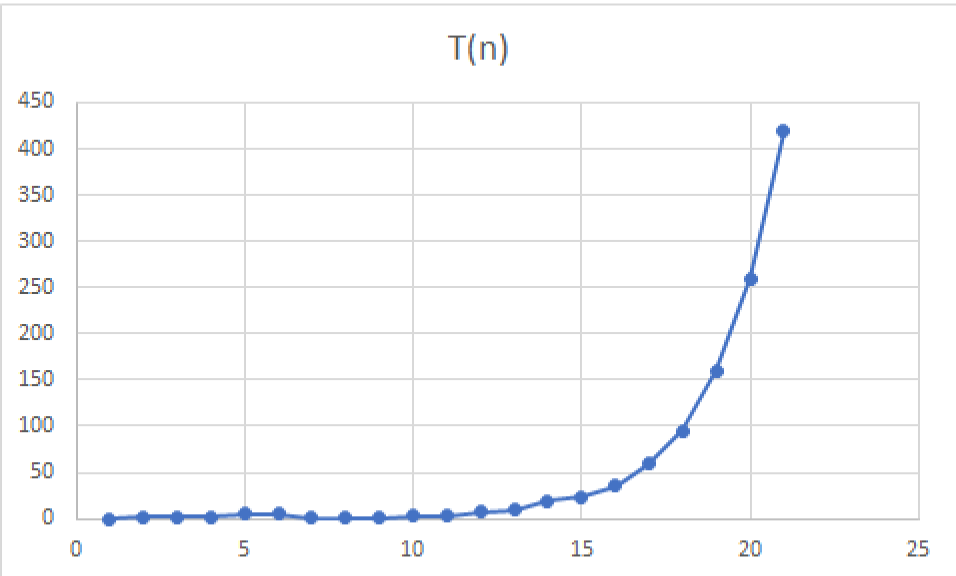
$O(c_1 F_n + c_2 L_n)$ Por definición de O

$O(F_n + L_n)$ Por regla del producto

3.6 Complejidad

La complejidad del algoritmo cuando n tomo los peores valores, es decir cuando se le pide al algoritmo calcular un termino n-esimo más grande de Fibonacci, la ecuación adquiere una complejidad de

$O(2^n)$.



Podemos observar que al realizar un testeo del tiempo de ejecución del algoritmo en función de n, los tiempos arrojados corresponden a la ecuación de complejidad $O(2^n)$