Laboratorio Nro. 1: Recursión

Luis Carlos Rodríguez Zúñiga
Universidad Eafit
Medellín, Colombia
lcrodriguz@eafit.edu.co

Laura Sánchez Córdoba
Universidad Eafit
Medellín, Colombia
lsanchezc@eafit.edu.co

2) Preguntas basadas en el Codigbat

3. El programa realiza la comparación de que la posición inicial no sea mayor al de la longitud del arreglo debido a que si lo hace el programa contara con un error y hará al objetivo 0, luego analiza si en el arreglo se encuentra un múltiplo de 5 y si consecutivo a él se encuentra el número 1, este no se tendrá en cuenta para la suma ej: ([3, 5, 1], 4)-→ false debido a que el uno no entra a las opciones de la suma, luego para realizar la suma se incrementa la variable start y al arreglo se le da la ubicación de start.

2.4.
Recursion 1

```
 public int factorial(int n) {
       if (n==0) //C1
       return 1; //C2
       return n*factorial(n-1); // n+T(n-1)

        }
      //T(n)= n+T(n-1)+C
```

Ecuación de recurrencia: T(n)=c1 +n/2(2C +n +1)
T(n)=O(c1 +n/2(2C +n +1))
O(n/2(2C +n +1)) Rs
O(2C +n +1)Rp
O(n) Rs

```
public int triangle(int r) {
  if (r == 0) return 0;   //C1 +C2
  else return r + triangle(r- 1); // n+T(n-1)
}
```

```
      //T(n)= n+T(n-1)+C
```

Ecuación de recurrencia: T(n)= c1 +n/2(2C +n +1)
T(n)=O(c1 +n/2(2C +n +1))
O(n/2(2C +n +1)) Rs
O(2C +n +1)Rp
O(n) Rs

BUNNYEARS

```
public int bunnyEars(int b) {
  if (b == 0) return 0; //C1+C2
  else return 2 + bunnyEars(b - 1); C3+ T(n-1)
}

//T(n)= T(n-1)+C
```

Ecuación de recurrencia: T(n) = c1 + Cn
T(n)=O(c1 + Cn)
O(Cn) Rs
O(n)Rp

BUNNYEARS 2

```
public int bunnyEars2(int b) {
    if (b == 0) return 0;  //C1
    if (b % 2 == 1) return 2 + bunnyEars2(b - 1);  //C2+ C3+T(n-1)
    else return 3 + bunnyEars2(b - 1); C4+T(n-1)
}
// T(n)= T(n-1)+C
```

Ecuación de recurrencia: T(n) = c1 + Cn
T(n)=O(c1 + Cn)
O(Cn) Rs
O(n)Rp

POWER N

```
public int powerN(int base, int n) {
  if (n == 0) return 1; //C1+ C2
  else return base * powerN(base, n - 1); //n+T(n-1)
```

```
}
     //T(n)= n+T(n-1)+C
```

Ecuación de recurrencia: T(n)= c1 +n/2(2C +n +1)
T(n)=O(c1 +n/2(2C +n +1))
O(n/2(2C +n +1)) Rs
O(2C +n +1)Rp
O(n) Rs

Recursión 2:
Tomados de:
Titulo: CodingBat: Java. Recursion-2
Autor:Gregor Ulm
Fecha:Marzo 2014
Disponible:http://gregorulm.com/codingbat-java-recursion-2/

GROUPSUM 5

```
public boolean groupSum5(int start, int[] nums, int target) {
   if (start >= nums.length) return target == 0;            //C1
   if (nums[start] % 5 == 0) {                              //C2
      if (start < nums.length - 1 && nums[start + 1] == 1) //C3
         return groupSum5(start + 2, nums, target - nums[start]); //T(n-1) + C4
      return groupSum5(start + 1, nums, target - nums[start]); //T(n-1) + C5
   }
   return groupSum5(start + 1, nums, target - nums[start])
         || groupSum5(start + 1, nums, target);

//T(n)=2T(n-1) +C
```

Ecuación de recurrencia:
T(n)= c1*2^(n-1)+C(2^n-1)

O(c1*2^(n-1)+C(2^n-1))
O(C(2^n-1)) RS
O(2^n-1) RP
O(2^n) RS

SPLIT ARRAY

```
public boolean splitArray(int[] nums) {
    return helper(0, nums, 0, 0);
}

private boolean helper(int start, int[] nums, int sum1, int sum2) {
    if (start >= nums.length) return sum1 == sum2;            //C1+C2
    return helper(start + 1, nums, sum1 + nums[start], sum2) //T(n-1) + C3
            || helper(start + 1, nums, sum1, sum2 + nums[start]); // T(n-1) + C4
}
//T(n)=2T(n-1) +C
```

Ecuación de recurrencia:
T(n)= c1*2^(n-1)+C(2^n-1)

O(c1*2^(n-1)+C(2^n-1))
O(C(2^n-1)) RS
O(2^n-1) RP
O(2^n) RS

GROUPNOADJ

```
public boolean groupNoAdj(int start, int[] nums, int target) {
    if (start >= nums.length) return target == 0; //C1 +C2
    return groupNoAdj(start + 2, nums, target - nums[start]) // T(n-1)+C3
            || groupNoAdj(start + 1, nums, target); T(n-1)+C4
}

//T(n)=2T(n-1) +C
```

Ecuación de recurrencia:
T(n)= c1*2^(n-1)+C(2^n-1)

O(c1*2^(n-1)+C(2^n-1))
O(C(2^n-1)) RS
O(2^n-1) RP
O(2^n) RS

SPLIT 53

```
public boolean split53(int[] nums) {
    return helper(0, nums, 0, 0);
```

```
}

private boolean helper(int start, int[] nums, int sum1, int sum2) {
    if (start >= nums.length) return sum1 == sum2;                //C1+C2
    if (nums[start] % 5 == 0)                                      //C3
        return helper(start + 1, nums, sum1 + nums[start], sum2); //T(n-1)+C4
    if (nums[start] % 3 == 0)                                      //C5
        return helper(start + 1, nums, sum1, sum2 + nums[start]); //T(n-1)+C6

    return helper(start + 1, nums, sum1 + nums[start], sum2) //T(n-1)+C7
            || helper(start + 1, nums, sum1, sum2 + nums[start]); //T(n-1)+C8  → peor de
los casos
}

//T(n)=2T(n-1) +C

Ecuación de recurrencia:
T(n)= c1*2^(n-1)+C(2^n-1)

O(c1*2^(n-1)+C(2^n-1))
O(C(2^n-1)) RS
O(2^n-1) RP
O(2^n) RS

     GROUPSUM 6

public boolean groupSum6(int start, int[] nums, int target) {
    if (start >= nums.length) return target == 0; //C1
    if (nums[start] == 6) //C2
        return groupSum6(start + 1, nums, target - nums[start]); //T(n-1)+C3
    return groupSum6(start + 1, nums, target - nums[start])  //T(n-1)+C4
            || groupSum6(start + 1, nums, target); //T(n-1)+C5
}

//T(n)=2T(n-1) +C

Ecuación de recurrencia:
T(n)= c1*2^(n-1)+C(2^n-1)

O(c1*2^(n-1)+C(2^n-1))
O(C(2^n-1)) RS
```

O(2^n-1) RP
O(2^n) RS

2.5. En los casos de análisis de complejidad de los ejercicios de recursión 1 se puede observar un tiempo de O(n) lo que significa que nuestro tiempo va acorde con la cantidad de datos a ser analizados dentro del algoritmo, donde se puede considerar a "n" como nuestra cantidad de datos a ser analizados.
En los ejercicios de recursión 2, se puede concluir que, "n" es como la longitud o la cantidad de datos que se encuentran dentro del arreglo y "m" el objetivo o el target, lo cual nos lleva a un doble recorrido a dicho arreglo analizando que datos("n") nos lleva a conseguir un "m" como resultado.

3) Simulacro de preguntas de sustentación de Proyectos

1. **Tablas: debido a que se tardaba más de un minuto para realizar las ejecuciones los valores usados se muestran en cada una de las siguientes tablas:**

Arraysum

Datos en el array	Tiempo en nanosegundo
10	3265
100	6998
1000	84905
10000	90503
100000	105897

GroupSum

Datos en el array	Tiempo en nanosegundo
10	114761
20	12318184
40	22163379563
50	44277312383
100	2.19605E+11

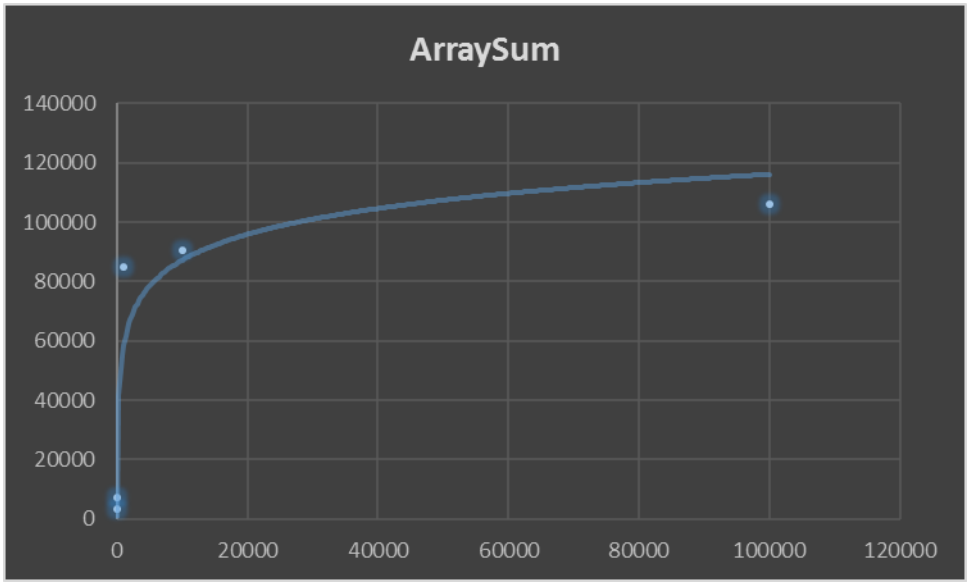
Fibonacci

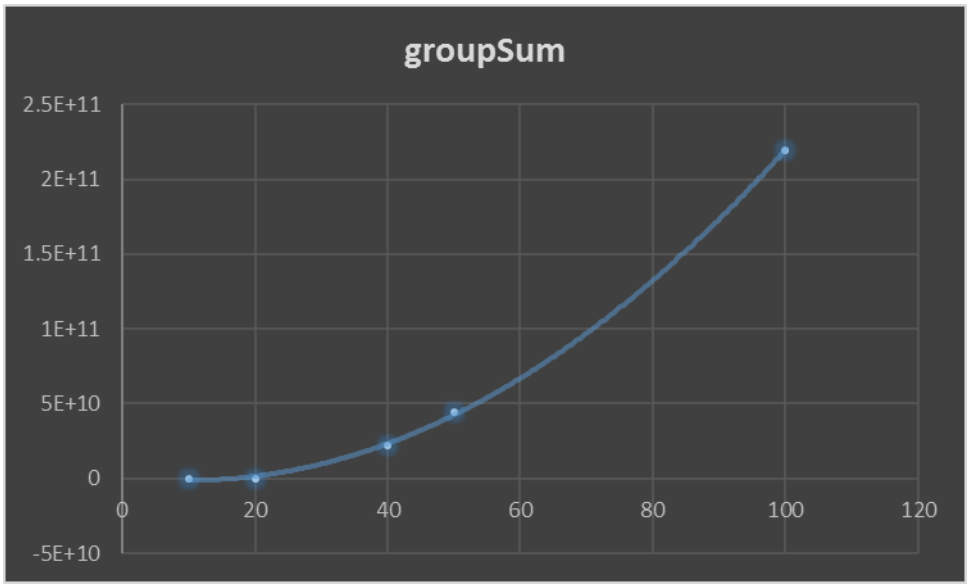
Datos a analizar	Tiempo en nanosegundos
10	9330
20	426857
30	7655421
40	1412754090

50	1.12083E+11

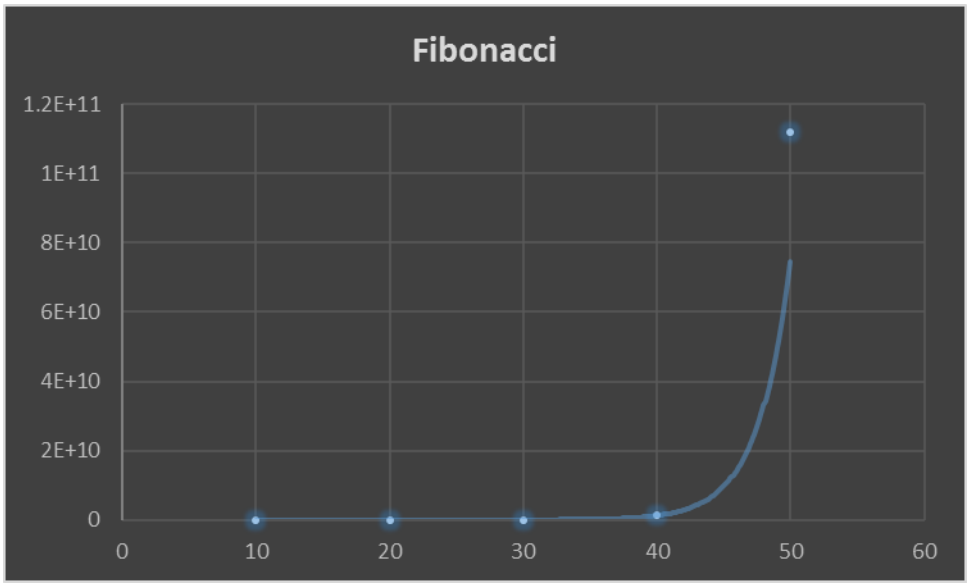
(se midió en nanosegundos para obtener una mayor exactitud sobre todo con los números pequeños).

2.





La mejor línea de tendencia para esta grafica fue la polinómica y no la logarítmica



En este caso la línea de tendencia más aproximada la exponencial y no la logarítmica

3. Pueden variar según el cálculo de las constantes las cuales dependerán de los procesos y servicios que esté desarrollando el procesador (independientes de programa) y la cantidad de datos que sea capaz de analizar el procesador de la máquina, por esto habrán maquinas que ejecuten el programa más rápido que otras que hasta el tiempo sea difícil de diferenciar en milisegundos los cuales es mejor tomar en nanosegundo y las gráficas (las cuales se esperan que sean en log) pueden variar de los factores anteriores.

4. El stackOverflow hace referencia a un problema que ocurre debido a un mal llamado recursivo, el stack almacena constantemente datos sin una condición de parada por lo que se llena la memoria que la máquina le tiene asignada puesto que su tamaño es limitado, para evitarlo se hace necesario definir bien la función recursiva que se detenga en un momento dado para dar una solución y evitar que la pila se desborde.

5. El valor más grande que se pudo calcular para Fibonacci fue 50, el cual se tardó 1,12083*10^11 nanosegundos puesto que se toma demasiado tiempo la ejecución con números grandes, pues su complejidad es 2^n pues usa dos llamados recursivos, por ende, si se planea ejecutar Fibonacci con 1 millón ser tendría una complejidad d 2^1000000, lo que toma demasiado tiempo de

ejecución dada la cantidad de llamados recursivos de los que se depende para hallar el resultado.

6. A través de lo que se llama programación dinámica que permite almacenar la solución de varias sub-rutinas llevadas a cabo mediante la recursión que se usó para resolver el problema, y los cuales pueden ser necesitados posteriormente para resolver el problema para índices más grandes. Este proceso de guardar las soluciones del problema para evitar resolverlo de nuevo se llama *memorización* y trabaja a la par con la recursión (tomado de: http://algorithms.tutorialhorizon.com/introduction-to-dynamic-programming-fibonacci-series/).

7. Podemos observar que mientras en recursión 1 se tiene una complejidad lineal O(n) por lo que el tiempo aumenta de la misma manera que los datos (por ejemplo 1 seg→1 dato), en recursión 2 se tiene una complejidad de 2^n (el tiempo de ejecución será dos elevado a la cantidad de datos) mayor, dados los dos llamados recursivos que se necesitan para resolver los problemas. De esta manera concluimos que: en primer lugar, dada la complejidad, para los ejercicios de recursión 2 no se podrían usar valores demasiado grandes, pues la ejecución tardaría demasiado y en segundo, que los algoritmos de recursión 2 comparando complejidad, son menos eficientes que los de recursión 1.

4) Simulacro de Parcial

1. *Start +1, nums, target*
2. *a*
3. *1. n-1, a.b,c*
   2. a,b
   3. res, c
4. *e*

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co