2.3 primero mira si todavía hay numero en el arreglo, para decir que si no hay más números termine el programa. Después mira en el arreglo si el numero en el que está en ese momento la función es un 5, si lo es, analiza el siguiente número, porque si es un 1 no lo tiene encuentra saltando a el siguiente numero o terminando el siclo (en caso de ser el último número), y después añade el 5 a la suma, y la última parte lo que se hace es seguir el ciclo normal del SumGroup.

2.4

```
public int triangle(int rows) {  // C0
  if(rows==0) return 0;   // C1
  return rows + triangle(rows-1); // C2 +t (n-1)
}
```

T(n)=

- (C0+C1) n=0
- t(n) n>0

T(n) =O(C2*n+K)= O(n)

Siendo n el entero rows

```
public String noX(String str) { //C0
  if(str.length() == 0) return ""; // C1
  if(str.charAt(0) == 'x') return noX(str.substring(1)); // C2+ t(n-1)
  return str.charAt(0) + noX(str.substring(1)); /C3+ t(n-1)
}
```

T(n)=

- (C0+C1) n=0
- C2+C3+t(n-1) n=x

T(n) =O((C2+C3)*n+K)= O(n)

Siendo n el número de caracteres del String

```
public int countPairs(String str) { //C0
  if(str.length() <=2) return 0; // C1
  if(str.charAt(0) == str.charAt(2)) return 1 + countPairs(str.substring(1)); // C3+t(n-1)
  return countPairs(str.substring(1)); // t(n-1)
```

}

T(n)=

- (C0+C1) n<=2
- C2+C3+t(n-1) n=n+2

T(n) =O(C3*n+K)= O(n)

Siendo n el numero de caracteres en el String

```
public int array11(int[] nums, int index) { //C0
  if(nums.length == index) return 0; //C1
  if (nums[index] == 11) return 1 + array11(nums , index + 1); //C2 + t(n,m-1)
  return array11(nums,index + 1); //t(n,m-1)
}
```

T(n)=

- (C0+C1) n=m
- C2+C3+t(n,m-1) n=11

T(m,n) =O(C2* t(n,m-1)+K)= O(m,n)

Siendo n el número de números en el arreglo y m siendo el número del índice

```
public String changePi(String str) { //C0
  if(str.length() < 2) return str; // C1
  if(str.substring(0,2).equals("pi")) return "3.14" + changePi(str.substring(2));// C2 + t(n-2)
  return str.charAt(0)+changePi(str.substring(1)); //C3+t(n-1)
}
```

T(n)=

- (C0+C1) n< 2
- C2+C3+t(n-1) n>=2

T(n) =O((C2+C3)*t(n)+K)= O(n)

Siendo n el número de caracteres en el String

```
public boolean groupSum6(int start, int[] nums, int target) { //C0
   if (start >= nums.length) return target == 0; // C1
   if (nums[start] == 6) // C2
      return groupSum6(start + 1, nums, target - nums[start]);//t(n-1)
   return groupSum6(start + 1, nums, target - nums[start])
         || groupSum6(start + 1, nums, target); //2t(n-1)
}
```

T(n)=

- (C0+C1) m>=n
- C2+ t(n-1) +2t(n-1) M=6

$T(n) = O(1/6(2k3^n+3C2(3^n-1))) = O(3^n)$

Siendo n el número de números en el arreglo m el número de start

```
public boolean groupNoAdj(int start, int[] nums, int target) { //C0
   if (start >= nums.length) return target == 0; //C1
   return groupNoAdj(start + 2, nums, target - nums[start])
         || groupNoAdj(start + 1, nums, target); // t(n-2) + t(n-1)
}
```

T(n)=

- (C0+C1) m>=n
- t(n-2) +t(n-1) M=6

$T(n) = O(k2^{n-1}) = O(2^n)$

Siendo n el número de números en el arreglo m el número de start

```
public boolean groupSum5(int start, int[] nums, int target) { //C0
   if (start >= nums.length) return target == 0; // C1
   if (nums[start] % 5 == 0) { //C2
      if (start < nums.length - 1 && nums[start + 1] == 1)// C3*n
         return groupSum5(start + 2, nums, target - nums[start]); //t(n-2)
      return groupSum5(start + 1, nums, target - nums[start]); //t(n-1)
   }
   return groupSum5(start + 1, nums, target - nums[start])
         || groupSum5(start + 1, nums, target); //2t(n-1)
}
```

T(n)=

- (C0+C1) m>=n
- t(n-2) +t(n-1) n/5=entero

$T(n) = O(k2^{n-1}) = O(2^n)$

Siendo n el número de números en el arreglo m el número de start

```
public boolean groupSumClump(int start, int[] nums, int target) {//C0
   if (start >= nums.length) return target == 0; //C3
   int sum = nums[start]; //C3
   int count = 1; //C4
   for (int i = start + 1; i < nums.length; i++)//C5*n
      if (nums[i] == nums[start]) { //C6
         sum += nums[i]; //C7
         count++; //C8
      }
   return groupSumClump(start + count, nums, target - sum)
         || groupSumClump(start + count, nums, target); //2t(n-1)
}
```

$O=((C1+C2+C3+C4+C5+C6+C7+C8)*n + (C9)*n^2) =O(n^2)$

Siendo n el número de números en el arreglo m el número de start

```
public boolean splitArray(int[] nums) { //C0
   return helper(0, nums, 0, 0); //t(C1)
}
public boolean helper(int start, int[] nums, int sum1, int sum2) {
   if (start >= nums.length) return sum1 == sum2; //t(C2*n)
   return helper(start + 1, nums, sum1 + nums[start], sum2) //t(C3*n^2)
         || helper(start + 1, nums, sum1, sum2 + nums[start]);
```

$O=(C1+(C2)*n + (C3)*n^2) =O(n^2)$

Siendo n el número de números en el arreglo

3)

3.1) el Stack overflow es cuando la memoria se llena y no puede seguir haciendo procesos, como tener un ciclo infinito, o en recursión cuando no se tiene una condición de parada o la condición de parada es imposible de llegar, lo cual sigue haciendo el programa incontables veces hasta que se agota la memoria, es evitable si se arreglan las condiciones de parada (aunque es posible que exista un código muy grande que lo agote la memoria, en este caso lo mejor es optimizarlo).

3.2) se pudo calcular hasta 55, pero no por el error del Stack overflow, sino por el tiempo que se necesita para llegar a calcularlo, el 56 se demora 37 minutos, por lo que el siguiente debe estar alrededor de 1 hora u ahí hasta que la memoria no pueda continuar haciendo los procesos y no puede calcular el Fibonacci de 1millon, ya que el Stack overflow impediría tal uso de la memoria y el numero sería muy grande como para que se pueda guardar en memoria

3.3) para poder calcular numero grandes se puede hacer que el Fibonacci guarde sus números en la memoria sólida, que no esté en la Ram y así poder calcularlo simplemente sumando el anterior y anterior a ese, pero se necesita tener la lista creada para números grandes. Si no se tiene la lista guardada, se puede crear un ciclo que guarde los datos del Fibonacci y al llegar a el siguiente lo único que necesita hacer es sumar el anterior y el anterior a este

3.4) los problemas de recursión 1 son de O(n) mientras que los de recursión 2 son de $O(x^n)$ es decir que mientras uno tiene complejidad lineal el otro la tiene exponencial, por lo que los de recursión 1 (no tan complejos) se pueden hacer más rápido y consumen menos memoria para el mismo número de datos