Algorithmics	Student information	Date	Number of session
	UO: 283928	03/14	Session 3
	Surname: Suárez Losada		
	Name: Gonzalo		

Escuela de Ingeniería Informática
Universidad de Oviedo

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Activity 1. Basic recursive models

Complexities of every class

Division1.java

This method uses the Divide an Conquer strategy by division as every class called so. Then, the idea for getting the complexities is similar among them all so it will not be repeated.

There are three variables in the complexity of such a class:

- a: amount of recursive calls
- b: denominator in the recursive call
- k: overall complexity of the method without recursive calls

Then we must check the proportion of a with respect to b^k:

- If $a < b^k$: $O(n^k)$
- If $a = b^k$: $O(n^k * \log n)$
- If $a > b^k$: $O(n^{\log_b a})$

For this example:

```
{
 long cont = 0;
 if (n<=0) cont++;
 else
  { for (int i=1;i<n;i++) cont++ ;   //O(n)
    rec1(n/3);
  }
 return cont;
```

- a = 1 recursive call
- b = 3
- k = 1 (linear complexity)

Then, as $1 < 3^1$, the complexity will be linear $O(n^1)$

Division2.java

```
public static long rec2 (int n)
{
 long cont = 0;
 if (n<=0) cont++;
 else
  { for (int i=1;i<n;i++) cont++ ;
    rec2(n/2);
    rec2(n/2);
  }
 return cont;
}
```

- a = 2 recursive call
- b = 2
- k = 1 (linear complexity)
- As $2 = 2^1$, complexity will be $O(n * \log n)$

Algorithmics	Student information	Date	Number of session
	UO: 283928	03/14	Session 3
	Surname: Suárez Losada		
	Name: Gonzalo		

Division3.java

```
public static long rec3 (int n)
{
 long cont = 0;
 if (n<=0) cont++;
 else
  { cont++ ; // O(1)
    rec3(n/2);
    rec3(n/2);
  }
 return cont;
}
```

- a = 2 recursive call
- b = 2
- k = 0 (constant complexity)
- As 2 > 2^0, complexity will be O($n^{\log 2 (2)}$)

Division4.java

```
public static long rec4 (int n)
{
 long cont = 0;
 if (n<=0) cont++;
 else
  { for (int i=1;i<n;i++) cont++ ;  //O(n)
    for ( int j = 1; j < n ; j++ ) cont++;
    rec4(n/2);
    rec4(n/2);
    rec4(n/2);
    rec4(n/2);

  }
 return cont;
}
```

- a = 4 recursive calls
- b = 2
- k = 1 (linear complexity)
- As 4 < 2^1 complexity will be O($n^{\log 2 (4)}$) = O(n^2)

fib1()

```
/* Iterative solution with temporal complexity O(n) */
public static int fib1(int n) {
  int n1 = 0;
  int n2 = 1;
  for (int i = 1; i <= n; i++) {
      int s= n1+n2;
      n1 = n2;
      n2 = s;
  }
  return n1;
}
```

In this case, no recursion is used. In fact, as stated, an iterative linear complexity approach is used.

Algorithmics	Student information	Date	Number of session
	UO: 283928	03/14	Session 3
	Surname: Suárez Losada		
	Name: Gonzalo		

fib2()

```
/* Second iterative solution with time complexity O(n)
 * using a vector. This algorithm will be viewed later
 * as a very simple case dynamic programming */
public static int fib2(int n, int[]v) {
  v[0] = 0;
  v[1] = 1;
  for (int i=2; i <= n; i++)
      v[i]=v[i-1]+v[i-2];
  return v[n];
}
```

Once again, this method uses an iterative linear complexity approach

From now on, the remaining methods will be implemented by a Divide and Conquer strategy based on subtraction. The concept is the same but modifying the complexity formulas:

- If a < 1: O(n^k)
- If a = 1: O(n^{k+1})
- If a > 1: O ($a^{n/b}$)

fib3()

```
/* First recursive solution: T(n)=T(n-1)+O(1)
 * => O(n) */
public static int fib3(int n) {
    return aux(0, 1, n);
}

private static int aux(int n1, int n2, int n) {
  if (n < 1) return n1;
    return aux(n2, n1+n2, n-1);
}
```

In this case, we are told a linear complexity is archieved. Let us check so:

- a = 1 recursive call
- b = 1
- k = 0

Then, as 1 == 1: complexity will be O(n^{0+1}) = O(n^1). Then, it was correct.

Algorithmics	Student information	Date	Number of session
	UO: 283928	03/14	Session 3
	Surname: Suárez Losada		
	Name: Gonzalo		

fib4()

```
/* Second recursive solution: T(n)=T(n-1)+T(n-2)+O(1),
 * that is an exponential solution O(1.6^n), that is,
 * each time n grows from n to n+1, the time is multiplied
 * by 1.6. Or what is the same, the amount of time it takes
 * for n is the sum of times for the preceding sizes
 * (n-1) and (n-2) */
public static int fib4(int n) {
  if (n<=1)
     return n;
  return fib4(n-1) + fib4(n-2);
}
```

By applying the D&C strategy by subtraction, we get:

- a = 2
- b = 1 or 2
- k = 0

So the final complexity will be somewhere in between O(2^n) and O($2^{n/2}$).

Subtraction1.java

```
public static long rec1(int n){
 long cont = 0;
 if (n<=0) cont++;
 else
   { cont++;  // O(1)=O(n^0)
     rec1(n-1);
   }
 return cont;
}
```

- a = 1 recursive call
- b = 1
- k = 0

Then, as 1 == 1: complexity will be O(n^{0+1}) = O(n^1).

Subtraction2.java

```
public static long rec2(int n)
{
 long cont = 0;
 if (n<=0) cont++;
 else
   { for (int i=0;i<n;i++) cont++; // O(n)
     rec2(n-1);
   }
 return cont;
}
```

- a = 1 recursive call
- b = 1
- k = 1

Then, complexity is O(n^{1+1}) = O(n^2)

Subtraction3.java

```java
public static long rec3(int n)
{
 long cont = 0;
 if (n<=0) cont++;
 else
  {
   cont++;  //O(1)
   rec3(n-1);
   rec3(n-1);
  }
 return cont;
}
```

- a = 2 recursive calls
- b = 1
- k = 0

Then, the complexity will be O($a^{n/1}$) = O(2^n), which is exponential

Subtraction4.java

```java
public static long rec4(int n){
     long cont = 0;
     if (n<=0) cont++;
     else
       { cont++;  // O(1)=O(n^0)
         rec4(n-2);
         rec4(n-2);
         rec4(n-2);
       }
     return cont;
   }
```

- a = 3 recursive calls
- b = 2
- k = 0

Then, the complexity will be O($a^{n/b}$) = O($3^{n/2}$), which is exponential and the one we are asked for

VectorSum1

```java
/* This method iteratively calculates the sum of
 * the vector with linear complexity O(n) */
public static int sum1(int[]a) {
 int n= a.length;
 int s=0;
 for(int i=0;i<n;i++)
    s=s+a[i];
 return s;
}
```

As stated, the method has an iterative linear complexity approach

Algorithmics	Student information	Date	Number of session
	UO: 283928	03/14	Session 3
	Surname: Suárez Losada		
	Name: Gonzalo		

VectorSum2

```
/* This method recursively calculates the sum of
 * the vector with a complexity O(n) */
public static int sum2(int[]a) {
 return recSust(0,a);
}

private static int recSust(int i, int[]a) {
 if (i==a.length)
     return 0;
 else
     return a[i] + recSust(i+1,a);
}
```

- a = 1 recursive call
- b = 1
- k = 0

Then, as 1 == 1: complexity will be O(n^{0+1}) = O(n^1).

VectorSum3

```
/*  This method recursively calculates the sum of
 * the vector with a complexity O(n) */
public static int sum3(int[]a) {
  return recDiv(0,a.length-1,a);
}

private static int recDiv(int iz,int de,int[]a) {
 if (iz==de)
     return a[iz];
 else {
     int m= (iz+de)/2;
     return recDiv(iz,m,a)+ recDiv(m+1,de,a);
 }
}
```

- a = 2 recursive calls
- b = 2
- k = 0

Then, as 2 > 2^0: complexity will be O ($n^{\log_2 2}$) = O(n)