


Algorithmics	Student information	Date	Number of session
	UO: 270534	03/02/2021	5.1
	Surname: Iglesias Préstamo	 Escuela de Ingeniería Informática Universidad de Oviedo	
	Name: Ángel		



Activity 1. [Basic recursive models]

1. Division.

a. Task 1 – Division I.

As can be seen, the **Division1** method's complexity can be studied by applying the techniques relatives to the *Divide and Conquer* algorithms. This means, we can analyze the time complexities of the given piece of code by finding three values:

- i. $a \rightarrow$ number of recursive calls.
- j. $b \rightarrow$ number by which each call's size is reduced.
- k. $k \rightarrow$ complexity of the rest of the code, without considering the recursive calls.

Not only that but we must also study the relationship among those terms, this is:

- i. In the case $a < b^k$ the time complexity is: $O(n^k)$.
- j. If $a = b^k$ the time complexity is: $O(n^k \cdot \log(n))$.
- k. If $a > b^k$ the time complexity is: $O(n^{\log_b a})$.

Having all of this in mind, we can easily calculate the complexities of roughly any *Divide and Conquer* algorithm by **division**. As this is the first example, we will go step-by-step analyzing this case:

1. The value of a can be calculated in a rush by simply looking at the code: note that there's just 1 recursive call – this means to the method itself – and it's not inside of a loop, so $a = 1$.

2. The value of b is calculated by looking at the recursive call: notice that $rec1(n/3)$, means that the next call of the method's size will be the third of the actual size. This implies the value of $b = 3$.

3. In the case we want to find k , we can remove all the recursive calls from the equation, leaving us with just a for loop from 1 to n , with steps of 1. This is: the complexity of the code without considering the recursive calls: $O(n)$. And the value of $k = 1$.

Last, but the least, applying everything explained so far, let me compare a with b^k : as $a = 1, b = 3$ and $k = 1$; we can conclude: $1 < 3^1$ and so the **time complexity** is $O(n)$.

b. Task 2 – Division II.

It's clear I'm not going to explain as much as I did for the rest of the algorithms. However, everything will be properly defined.

1. As there are 2 recursive calls to the method: $a = 2$.
2. With each call, the size of the problem is divided by 2: $b = 2$.
3. There's a single and only for loop, this leads us to a time complexity of $O(n)$. And so $k = 1$.

Let me compare a with b^k : as $a = 2, b = 2$ and $k = 1$; we can conclude: $2 = 2^1$ and so the **time complexity** is $O(n \cdot \log(n))$.

c. Task 3 – Division III.

When solving this task, I – again – used the same process for me to calculate the time complexity of the given method.

1. As there are 2 recursive calls to the method: $a = 2$.
2. With each and every call, the size of the problem is divided by 2: $b = 2$.
3. The complexity of the rest of the code is $O(1) = O(n^0)$. And so, $k = 0$.

Let me compare a with b^k : as $a = 2, b = 2$ and $k = 0$; we can conclude: $2 > 2^0$ and so the **time complexity** is $O(n^{\log_2 2}) = O(n)$. As the logarithm with base 2 of 2 is 1.

d. Task 4 – Division IV.

For me to create this class, I performed some reverse engineering techniques, not even close; in order me to match the requested complexity: $O(n^2)$. What I understood is that the easiest formula I could follow for me to match the complexity is the first: when $a < b^k$ the time complexity is: $O(n^k)$. If we manage to restrict the values of a, b and k in order them to satisfy $a < b^k$, and we set $k = 2$: the complexity would be: $O(n^2)$. More in more, in the statement we are told to fix the value of $a = 4$: subdivide the problem into 4 sub-problems. This leads us to: $4 < b^2$, for example, with a value of $b = 4$, every condition will be met: $4 < 4^2 \Rightarrow 4 < 16$.

To sum up, we will need to perform **4 recursive calls**, in each call we will need to **split the size of the subproblems by 4**. And the **complexity of the rest of the code**, without taking into account the recursive calls must be $O(n^2)$.

2. Subtraction.

a. Task 1 – Subtraction I.

Let me perform a brief explanation for us to understand how to calculate the complexities of the *Divide and Conquer* algorithms by **subtraction**. Indeed, everything that changes are the formulas:

- i. If $a = 1$ the time complexity is: $O(n^{k+1})$.
- j. If $a > 1$ the time complexity is: $O\left(a^{\frac{n}{b}}\right)$.

Having that in mind, let's calculate the time complexity of the code:

- 1. As there is a recursive call: $a = 1$.
- 2. With each call, the size of the problem is reduced by 1: $b = 1$.
- 3. The complexity of the rest of the code is $O(1) = O(n^0)$. And so, $k = 0$.

Let me compare a with 1: as $a = 1, b = 1$ and $k = 0$; we can conclude: $a = 1$ and so the **time complexity** is $O(n^{k+1}) = O(n^{0+1}) = O(n)$.

b. Task 2 – Subtraction II.

Let's move on to the next class:

- 1. As there is a recursive call: $a = 1$.
- 2. With each call, the size of the problem is reduced by 1: $b = 1$.
- 3. The complexity of the rest of the code is $O(n)$. And so, $k = 1$. Remember everything explained so far about the complexities when there are for loops involved.

Let me compare a with 1: as $a = 1, b = 1$ and $k = 1$; we can conclude: $a = 1$ and so the **time complexity** is $O(n^{k+1}) = O(n^{1+1}) = O(n^2)$.

c. Task 3 – Subtraction III.

Finally, we are about to reach the end:

- 1. As there are two recursive calls: $a = 2$.

2. With each call, the size of the problem is reduced by 1: $b = 1$.

3. The complexity of the rest of the code is $O(1) = O(n^0)$. And so, $k = 0$.

Let me compare a with b^k : as $a = 2$, $b = 1$ and $k = 0$; we can conclude: $a > 1$ and so the **time complexity** is $O\left(a^{\frac{n}{b}}\right) = O\left(2^{\frac{n}{1}}\right) = O(2^n)$.

d. Task 4 – Subtraction IV.

As explained in Task 4 of the previous paragraphs. I started by having a look at the formulas that could fit the best with the requested complexity: $O(3^{\frac{n}{2}})$. By comparing this complexity with the second formula of *Divide and Conquer* by **subtraction**: $O\left(a^{\frac{n}{b}}\right)$. If we set the value of $a = 3$ and $b = 2$. Notice that the value of k will not affect the final value of the complexity, we will set it to 0.

With 3 recursive calls, and at each recursive call we reduce the size of the problem by 2, we will match the requested time complexity. The complexity of the rest of the code is constant.