# CS 2742 (Logic in Computer Science) – Winter 2014
# Lecture 24

Antonina Kolokolova

March 24, 2014

## 7.1 Grows of functions

What do we mean when we say that one algorithm. is faster than another? In computer science, the standard definition of one algorithm being faster than the other is that on *large enough* inputs the faster algorithm makes less operations *in the worst case*. What it means to say "in the worst case" is that out of all executions of algorithm for an input of a give size, we look at the longest execution time. For example, if we are searching for an element in the array, it is always possible that we get lucky and hit this element on the very first step. However, it is more realistic to consider the running time of this algorithm in the case when the element is the last one we look at, or not in the array at all: we know then that the algorithm can't do worse than that, this is a guarantee on its running time.

Here we will review the growth rate of some functions. Think of these functions as describing the worst-case running time of algorithms, so a function with a larger increase in its value will correspond to a slower algorithm.

As many of you know by now, $f \in O(g)$ if "f is at least as fast as $g$" is defined as follows: $\exists n_0, c \forall n > n_0 |f(n)| \leq c|g(n)|$

In particular, a $f(n) = \log n$ function grows slower (=better algorithm) than linear $f(n) = n$ (so $\log n \in O(n)$), which is in turn is better than quadratic $f(n) = n^2$, and all of them are much better than the exponential $f(n) = 2^n$. As an example, a binary search has logarithmic time complexity, searching in an array is linear, mergesort is $O(n \log n)$ which is better than the bubble sort with time $O(n^2)$.

**Example 1.** Here is an example of proving that one function is in $O()$ of another. We will show that $3n \in O(2^n)$. Set $n_o = 2$ and $c = 3$. Now $\forall n > 3$, $3n < 3 * 2^n$.

But there are functions, recursively defined, that grow so fast that we can't even describe their growth. An example of such a function is the Ackermann's function, recursively defined as follows:

$$\forall m, n > 0, A(0,n) = n+1, A(m,0) = A(m-1,1), A(m,n) = A(m-1, A(m,n-1))$$

This function is well-defined, but $A(x,x)$ grows very fast: $A(4,4)$ is already $2^{2^{2^{655536}}}$.

Talking about being well-defined, some seemingly sensible recursive definition may result in a not well-defined function (as in it is impossible to determine the value of a function, or there is some problem with determining it):

$$G(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + G(n/2) & \text{if } n \text{ is even} \\ G(3n-1) & \text{if } n > 1 \text{ and is odd} \end{cases}$$

Not well-defined because $G(5) = G(14) = 1 + G(7) = 1 + G(20) = 1 + (1 + G(10)) = 1 + 1 + 1 + G(5) = 3 + G(5)$. So $G(5) = 3 + G(5)$, subtracting $G(5)$ from both sides get $0 = 3$!

The "3n+1" problem can be represented as a question of whether the following recursive definition always gets to the base case.

$$T(n) = \begin{cases} T(n/2) & \text{if n is even} \\ T(3n+1) & \text{if n is odd} \\ 1 & \text{if } n = 1 \end{cases}$$

A more traditional way is to define it as a sequence of numbers starting with $n$, where every next number is either a half of previous (if it was even) or 3 times the previous plus 1 (if it was odd). So if we start with $n = 3$, we obtain a sequence $(3, 10, 5, 16, 8, 4, 2, 1)$. In this context, $T(n)$ defined as above is often called a *recurrence relation*.

But, interestingly, it is still open whether starting from any number $n$ it will eventually get to 1. Or alternatively, for our recursive definition of $T(n)$, if you were to write a program computing $T(n)$ according to that definition and stopping when 1 is reached, you would not be able to prove that your program terminates and returns an answer for any input without solving this problem, that is, without proving that any such sequence will always reach 1.

## 7.2 Running time of a recursive algorithm

There is one context in computer science where recursive definitions of this kind occur all the time: determining a running time of a recursive program. Consider, for example, the binary search algorithm: to find an element in a sorted array, check the middle element of the array, if it is equal to the one we are looking for then return, otherwise if it is less than ours then search in the right half of the array, and if it is greater than ours search in left half. Suppose that $b$ is the amount of time (number of steps, for example) needed to perform the instructions on one such check; and suppose that $c$ is the number of steps needed to return an answer if the array in which we are looking has size 0 or 1. Then, the worst-case running time of the algorithm $T(n)$ can be described as follows: it is $c$ if the size $n$ of the array is 0 or 1, and otherwise it is at most $T(\lceil n/2 \rceil) + b$:

$$T(n) = \begin{cases} c, & \text{if } n = 0 \text{ or } n = 1 \\ T(\lceil n/2 \rceil) + b & \text{if } n > 1 \end{cases}$$

For the lack of time, I will not go into detail on how to solve this recurrence relation. There is a method called Master Theorem that can be applied directly to most of the expressions of this form you will encounter in practice. In particular, for the binary search it gives $T(n) \in O(\log n)$.

To see another example, consider Mergesort algorithm. There, the algorithm recurses on both halfs of the array; moreover, it needs $O(n)$ time to combine the halfs of an array of size $n$, and has to do this at every level of the recursion. This gives us the following recurrence relation, again assuming time $c$ for the array of size 0 or 1, but now hiding the constants under $O$-notation for the amount of work done within one step:

$$T(n) = \begin{cases} c, & \text{if } n = 0 \text{ or } n = 1 \\ 2T(\lceil n/2 \rceil) + O(n) & \text{if } n > 1 \end{cases}$$

In this case, solving the recurrence relation will give us $T(n) \in O(n \log n)$.