

TOPIC PLAN		
Partner organization	Belgrade Metropolitan University	
Topic	Analysis of Complexity of algorithms	
Lesson title	Big-O notation	
Learning objectives	<p>Student can interpret Big-O notation.</p> <p>Student can calculate complexity of an algorithm.</p> <p>Student can compare two or more algorithms by their complexity.</p>	<p>Strategies/Activities</p> <p><input type="checkbox"/> Graphic Organizer</p> <p><input type="checkbox"/> Think/Pair/Share</p> <p><input checked="" type="checkbox"/> Modeling</p> <p><input checked="" type="checkbox"/> Collaborative learning</p> <p><input checked="" type="checkbox"/> Discussion questions</p> <p><input type="checkbox"/> Project based learning</p> <p><input checked="" type="checkbox"/> Problem based learning</p>
Aim of the lecture / Description of the practical problem	<p>The aim of this lecture is to learn how to calculate the complexity of an algorithm. Using the notion of functional limit, the notion of Big-O will be introduced.</p> <p>Practical problem: When studying the complexity of an algorithm, we are concerned with the growth in the number of operations required by the algorithm as the size of the problem increases. In order to get a handle on its complexity, we first look for a function that gives the number of operations in terms of the size of the problem, usually measured by a positive integer n, to which the algorithm is applied. We then try to compare values of this function, for large n, to the values of some known function, such as a power function, exponential function, or logarithm function. Thus, the growth of functions refers to the relative size of the values of two functions for large values of the independent variable.</p>	<p>Assessment for learning</p> <p><input checked="" type="checkbox"/> Observations</p> <p><input checked="" type="checkbox"/> Conversations</p> <p><input checked="" type="checkbox"/> Work sample</p> <p><input type="checkbox"/> Conference</p> <p><input type="checkbox"/> Check list</p> <p><input type="checkbox"/> Diagnostics</p>

Previous knowledge assumed:	functional limits	Assessment as learning <input checked="" type="checkbox"/> Self-assessment <input type="checkbox"/> Peer-assessment <input type="checkbox"/> Presentation <input type="checkbox"/> Graphic Organizer <input type="checkbox"/> Homework Assessment of learning <input checked="" type="checkbox"/> Test <input checked="" type="checkbox"/> Quiz <input type="checkbox"/> Presentation <input type="checkbox"/> Project <input type="checkbox"/> Published work
Introduction / Theoretical basics	<p style="text-align: center;">Functional limits</p> <p>Definition 1. The limit of function $f: [b, c] \rightarrow \mathbb{R}$ as x goes to $a \in [b, c]$ equals L if and only if for every $\varepsilon > 0$ there exists some $\delta > 0$ such that whenever $x (\neq a)$ is within δ of a, then $f(x)$ is within ε of L. If a is a limit point of the domain of f, then, intuitively, the statement</p> $\lim_{x \rightarrow a} f(x) = L$ <p>is intended to convey that values of $f(x)$ get arbitrarily close to L as x is chosen closer and closer to a. The issue of what happens when $x = a$ is irrelevant from the point of view of functional limits. In fact, a need not even be in the domain of f. The above definition is so called $\delta - \varepsilon$ definition. Actually, if we choose output tolerance $\varepsilon > 0$, than must be some input tolerance $\delta > 0$ so that any input within δ of a has an output within ε of L (see Figure 1).</p>	

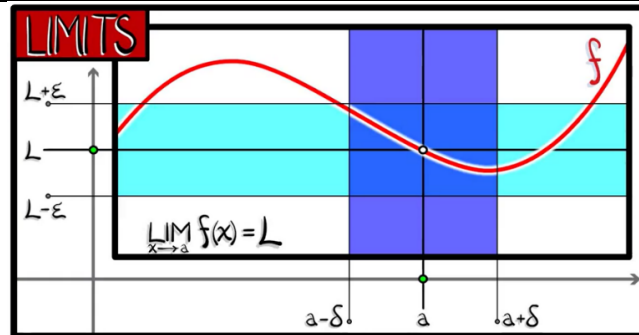


Figure 1.

The “critical” part of above definition is that as you change ε you need to be update δ (see Figure 2). If you make ε smaller still and decrease your level of acceptable error of the output, you need to find some amount of acceptable error on the input. And this has to continue for every possible non-zero value of ε .

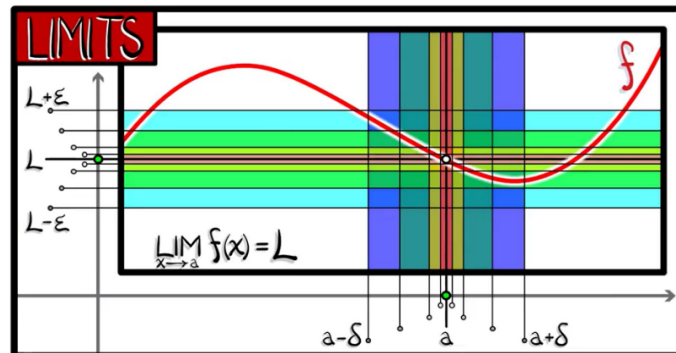
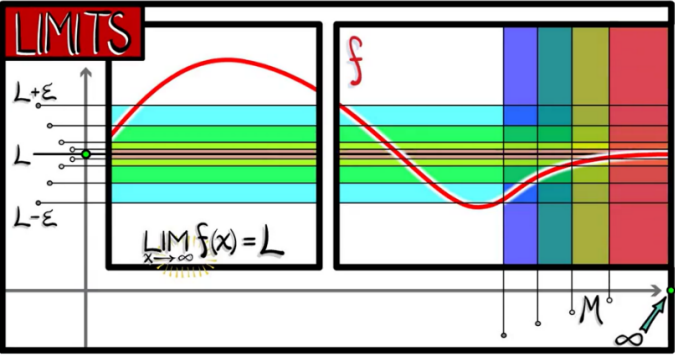


Figure 2.

This view of the definition is extendable in other context. Consider the limit as x goes to infinity of $f(x)$. What does it mean if that limit is equal to L ? For given output tolerance ε , there must be some tolerance on the input that guarantees striking within ε of L . In this case there is some bound $M \in \mathbb{R}$, so that whenever your input is greater than M , then your output is within ε of L . As before, this must be true no matter what ε you choose. Furthermore, if

	<p>you ε smaller and smaller, then M will be larger and larger.</p> <p>If the limits as x goes to infinity of $f(x)$ exists (see Figure 3), we use next denotation</p> $\lim_{x \rightarrow \infty} f(x) = L.$  <p>Figure 3.</p> <p>Analogous holds for limit as x goes to $-\infty$ of $f(x)$.</p>	
<p>Action</p>	<p>Discussion with students about orders of growth. Actually, we will talk about what happens when function gets very, very large, i.e. how quickly goes to infinity.</p> <p>Definition 1. Let functions f and g defined in the neighborhood of the point a, satisfy that</p> $\lim_{x \rightarrow a} f(x) = \lim_{x \rightarrow a} g(x) = \infty.$ <ol style="list-style-type: none"> 1) Functions f and g are of the same order, when x goes to a, if and only if holds $\lim_{x \rightarrow a} \frac{ f(x) }{ g(x) } = k, (k \neq \infty, k \neq 0).$ 2) Function f has a higher order than function g, when x goes to a, if and only if holds $\lim_{x \rightarrow a} \frac{ f(x) }{ g(x) } = \infty.$ 3) Function f has a lower order than function g, when x goes to a, if and only if holds 	

$$\lim_{x \rightarrow a} \frac{|f(x)|}{|g(x)|} = 0.$$

Remark. In terms of introducing a definition for a Big- O notation for using it for analysis of complexity of algorithms we will be particularly interested in the case when $a = \infty$ in Definition 4.

In this way, we can obtain the next scale of growth

$$\dots < \ln x < \dots < \sqrt[3]{x} < \sqrt{x} < x < x^2 < x^3 < \dots < 2^x < e^x < 3^x < 4^x < \dots$$

when x goes to ∞ . This scale are infinite and dense.

Quantifying this rates of changes, will give us a new language - Big- O notation (or asymptotic notation).

Definition 2. Given a function $g: \mathbb{R} \rightarrow \mathbb{R}$

$$O(g) = \left\{ f(x) : 0 \lim_{x \rightarrow \infty} \frac{|f(x)|}{|g(x)|} < \infty \right\}.$$

$f \in O(g)$ means that f is equal or less order than g .

According to Definition 1, if $\lim_{x \rightarrow \infty} \frac{|f(x)|}{|g(x)|} = \infty$ is satisfied, then $f \notin O(g)$.

Alternately, we can define big- O as follows.

Definition 3. Let $f, g: \mathbb{R} \rightarrow \mathbb{R}$. We say that f is $O(g)$ if and only if there are constants $c, N \in \mathbb{R}^+$ such that $|f(x)| \leq c \cdot |g(x)|$, for $x > N$ holds.

Some important properties of Big- O are

1. $f \in O(f)$ (reflexivity of big- O),
2. If $f \in O(g)$ and $c \in \mathbb{R} \setminus \{0\}$, then $c \cdot f \in O(g)$,
3. If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$ (transitivity of big- O),
4. If $f \in O(h)$ and $g \in O(h)$, then $f + g \in O(h)$,
5. $O(f + g) = \max\{O(f), O(g)\}$,
6. $O(f \cdot g) = O(f) \cdot O(g)$.

Complexity of an algorithm

When you are analyzing an algorithm or code for its computational complexity using Big- O notation, you can ignore the primitive operations that would contribute less-important factors to the run-time. Also, you always take the *worst case* behavior for Big- O .

Here is a list of classes of functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first.

Notation	Name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	$n \log n$
$O(n^2)$	quadratic
$O(n^r)$	Polynomial ($r > 1$, other than n^2)
$O(r^n)$	Exponential ($r > 1$)

The running time of iterative algorithms is straightforward to compute. Let $f_1(n)$ be the time it takes one iteration of the algorithm to run, and let $f_2(n)$ be the number of iterations. Then the running time of the algorithm is $O(f_1(n) \cdot f_2(n))$.

Example 1.

1.

```
int increment(int n) {
    return n + 1;
}
```

This function has one subexpression that takes constant time to execute and executes only once. So

"The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

	<p>it runs in $O(1)$.</p> <p>2.</p> <pre>int factorial(int n) { for (int i = n; i > 0; i++) { n *= i; } return n; }</pre> <p>This function has a subexpression $n *= i$ that takes constant time to execute, and this subexpression is executed n times. So $f_1(n) = 1$, $f_2(n) = n$, and the function runs in $O(n)$ time.</p> <p>3.</p> <pre>int foo(int n) { int x; for (int i = 0; i < n; i++) { x += i; } for (int i = 1; i < n/2; i++) { x *= i; } return x; }</pre> <p>In this case, there are two loops. The first runs in $O(n)$, and the second in $O\left(\frac{n}{2}\right) = O(n)$, so the total running time is in $O(n)$.</p> <p>4.</p> <pre>int bar(int n) { int x; for (int i = 0; i < n; i++) { for (int j = i; j < n; j++) { x += 1; } } }</pre>	
--	---	--

	<pre> return x; } </pre> <p>This function has one subexpression, the inner loop, that executes n times. What is the running time of the subexpression? The subexpression has one subexpression of its own that executes $n - i$ times and is constant. So the running time of the inner loop is in $O(n - i)$. Now the problem with determining the running time of the function is that i varies. But we can make estimates, as long as the estimates are greater than the actual value, so let's assume that the running time of the inner loop is n. Now the inner loop executes n times, so the total running time is in $O(n^2)$.</p> <p>5.</p> <pre> int baz(int k, int n) { int res = 0; for (int i = 0; i < k; i++) { res += (k - i) * k; } for (int i = 0; i < n; i++) { res -= (n - i) * i; } return res; } </pre> <p>This functions has two loops, the first of which is in $O(k)$ and the second of which is in $O(n)$. We don't know which of the two loops is faster, since it depends on the relative sizes of k and n, so we can only say that the function runs in $O(k + n)$. It is also possible to say that the function runs in $O(\max(k, n))$ since we can give an upper bound on the faster loop by assuming it runs in the same amount of time as the slower loop. Note that in general, it is not possible to give the running time of a multiple input function in terms of only one of its inputs.</p>	
--	---	--

"The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

6.

```
int foobar(int k, int n) {  
    int res = 0;  
    for (int i = 0; i < k; i++) {  
        for (int j = 0; j < n; j++) {  
            res += (k - i) * j;  
        }  
    }  
    return res;  
}
```

The inner loop runs in $O(n)$ time, and the outer loop iterates k times, so the running time of this function is in $O(k \cdot n)$.

Recursive algorithms are somewhat harder to analyze than iterative algorithms. They usually require inductive analysis. We start at the base case and work our way up higher inputs until we see a pattern. One way that helps is to draw a tree of the recursive calls, with each call as a node and an edge between the caller and the callee. We then count how many nodes are in the tree as a function of the input. Then the running time of the algorithm is the number of nodes in the tree times the amount of time each call takes (not including the recursive calls each call makes).

Example 2.

1.

```
int factorial2(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial2(n - 1);  
    }  
}
```

We draw a tree of the recursive calls in Figure 5. About n recursive calls are made, and each call takes constant time, so the running time of **factorial()** is in $O(n)$.

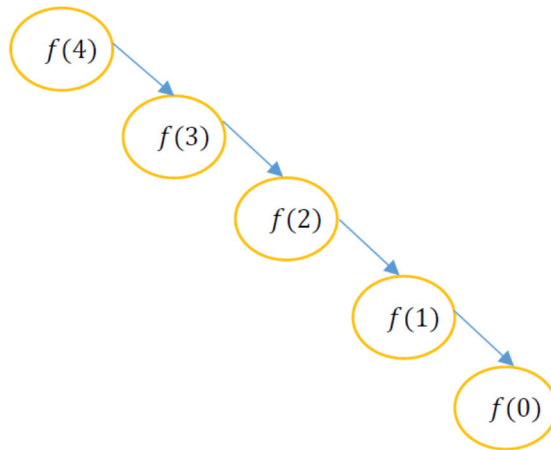


Figure 5. Tree of recursive calls for **factorial(4)**.

2.

```

int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
  
```

Again we draw a tree of the recursive calls in Figure 6. The tree is a nearly complete binary tree, so it has about $2n$ nodes in it. So the running time of this function is in $O(2^n)$.

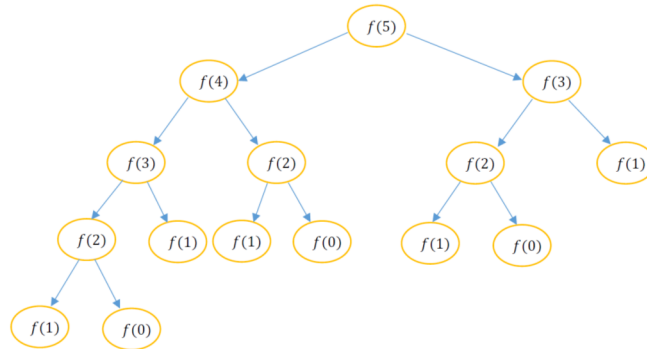


Figure 6. Tree of recursive calls for fibonacci(5).

Finally, at the end of the lesson we will mention other types of notations that are used in computer science to describe the performance or complexity of an algorithm. We will define the Big- Ω and the Big- Θ notations.

Materials / equipment / digital tools / software

The materials for learning are given as a part of references of the end from this topic plan;
Equipment: classroom, board, chalk;
Digital tools: laptop, projector;

Consolidation

- The teacher's discussion with the students through appropriate questions;
- Independent solving of simple tasks by the students under the supervision of the teacher;
- Given of examples by the teacher for introducing a new concept in a cooperation and a discussion with the students;
- Assignment of homework by the teacher with a time limit until the next class.

Reflections and next steps

Activities that worked

Parts to be revisited

<p>After the class, the teacher according to his personal perceptions regarding the success of the class fills in this part.</p>	<p>Through the success of the homework done by the students, questions and discussion at the beginning of the next class, the teacher comes to the conclusion which parts of this class should be revised.</p>
References	
<ol style="list-style-type: none"> 1. Dr Rale Nikolić, Elektronski materijali predavanja za učenje, Metropolitan Univerzitet, 2020. godina, Beograd 2. https://web.mit.edu/16.070/www/lecture/big_o.pdf 3. http://faculty.salisbury.edu/~ealu/COSC320/Lectures/complexity.pdf 4. https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172/lectures/11-BigO/11-BigO.pdf 	