3b. Asymptotics: Big O notation

Grzegorz Kosiorowski

Krakow University of Economics

1. Motivation and examples
2. Formal definitions
3. Hierarchy of typical sequences
4. Arithmetics of big O notation

Motivation

We are going to introduce a notation that allows us to compare the run times (and, thus, effectiveness) of various algorithms. To this aim, we will use our knowledge of sequences and their limits.
A disclaimer: we are not going to delve deeply into the topic of asymptotics and big O notations. In this presentation only the basics are provided (but it will suffice for our course).

Motivation

Assume that an algorithm deals with some data and the size of this data set is denoted by n (e.g. sorting lists of size n, solving systems of n linear equations...). The run time of this algorithm depends on the size of data. Thus for data of size n ($n = 1, 2, \ldots$) we denote the run time by $t(n)$, or simpler t_n. Naturally, the run time does not depend only on the quality and structure of the algorithm: for example, the efficiency of the hardware on which it is executed also matters. To eliminate any outside factors in comparison of run times, we focus not on their values for any particular n but on their growth rate expressed by so-called big O notation.

Example 1 - potatoes

Let the set of data for algorithms be a sack of n potatoes.
The first algorithm aims to check if there are no rotten potatoes in a sack in the simplest possible way: it just checks one by one if a potato is rotten or not. If conducting the operation "check if a potato is rotten" takes time of τ_1, then in the "worst" case (no rotten potatoes or only the last checked potato is rotten), the run time of this algorithm is $n \cdot \tau_1$. Namely, the run time of this algorithm increases linearly with the size of data: twice as many potatoes in the sack result in twice as long run time.

Example 1 - potatoes

Let the set of data for algorithms be a sack of n potatoes.
The second algorithm aims to check if there is a pair of potatoes of the same weight in the sack, again in the simplest possible way: it just weights all pairs of potatoes in the sack against each other, until it finds a pair of an equal weight (or goes through all the pairs). If conducting the operation: "check if two potatoes have the same weight" takes time of τ_2, then in the "worst" case (it needs to go through all pairs), the run time of this algorithm is $\frac{n(n-1)}{2} \cdot \tau_2$ (each potato needs to be weighted against $n-1$ others, and we need to divide the product $n(n-1)$ by 2 because we counted each step twice). We may bound the run time from above by $\frac{n^2}{2} \cdot \tau_2$. Namely, the run time of this algorithm increases with the square of the size of data: twice as many potatoes in the sack result in four times longer run time.

Example 1 - potatoes

What should be our focus when we compare run times of algorithms such as the two above? For any given n, it is difficult to say which algorithm is faster, because we do not know τ_1 i τ_2 (more precisely: they depend on our "hardware" and many other factors that fall outside our control). Therefore, it is more important to study which of the run times increases more when the size of data grows. From that point of view, when "sacks" are sufficiently large, the first algorithm will be faster. For example, even if for $n = 10$ the second algorithm's run time is 2 seconds and the first algorithm's run time is 5 seconds, then already for $n = 10$ the first algorithm will be executed in at most 50 seconds, while the second one might take even over 3 minutes to run.

Examples

Naturally, comparing the run times of two algorithms that solve different problems has only a theoretical value (we cannot say that one of them is "better") The "potato example" was constructed as just an illustration of the problem. The next two examples aim to reveal how we can benefit of a tool that measures the asymptotic growth of run time (or, so-called time computational complexity). We can compare the effectiveness of two algorithms solving the same problem.

Example 2 - sorting

You should already know some sorting algorithms.For example, the bubble sort of the list of size n requires that we go through the list $(n-1)$ times and each time we need to perform no more than $(n-1)$ comparisons of elements (and, potentially, swap their places in the list). To sum up, the bubble sort needs at most $(n-1)^2$ operations of "comparison and swapping". Thus, its run time sequence t_n increases proportionally to a square of the size of the list.

Example 2 - sorting

On the other hand, the merge sort algorithm has a run time proportional to $n \log n$ (where n is the size of the list and henceforth, the base of log is by default 2). To determine which of the sorting algorithms is more effective with respect to run time (or, to be precise, with respect to the growth of run time) we need only to know which of the sequences: $(n-1)^2$ or $n \log n$ grows faster.

Example 3 - systems of equations

There are multiple procedures for solving systems of linear equations: some of them you learned at school or on the algebra course. You will learn many other methods on the numerical methods course. Their time complexity differs widely. For example if n is the number of equations, then the run time of the "determinant method" based on Cramer's rule is proportional to $n!$, while solving systems of linear equations by elimination methods (e.g. Gauss-Jordan method) has the run time proportional to n^3.

Preliminary notions

Let t_n be the run time required by the algorithm to deal with the data of size n. What properties of $(t_n)_{n=1}^{\infty}$ should we analyze? Some changes in the size of t_n do not matter for time complexity:

- Re-scaling, namely multiplying all elements of t by a constant: such an effect may be obtained by the means of changing hardware, so it does not measure the effectiveness of the algorithm.
- Adding a constant changes even less.
- We need to focus not on the value of t_n for a given n but on the speed of its growth: does it increase as fast as n, n^2, 2^n, or maybe $\log n$ or any other function?

We need "only" to define what "increase as fast as" means...

Definition

Big O notation

Let $(t_n)_{n=1}^{\infty}$ and $(s_n)_{n=1}^{\infty}$ be two number sequences which elements are non-negative real numbers. Then, $t_n = O(s_n)$ (it is read "t_n is big O of s_n" or "t_n is of the order of s_n") if for sufficiently large numbers n the values of t are bounded from above by (potentially re-scaled) values of s. Formally:

$$\exists_{C>0} \exists_{n_0 \in \mathbb{N}} \forall_{n>n_0} |t_n| \leqslant C|s_n|.$$

The big O notation provides only the upper bound for the growth of t_n. We can informally read: „t_n grows no faster than s_n", but t_n can grow much slower (or not at all)!

About notation

The traditional notation $t_n = O(s_n)$ is somewhat unfortunate as the $=$ is misleading. Here, this sign does not denote equality (as it usually does) in any sense. For example, it lacks symmetry: writing $O(s_n) = t_n$ does not make sense and it is possible that $t_n = O(s_n)$ but $s_n \neq O(t_n)$.

Therefore, it would be more mathematically correct and less misleading to write $t_n \in O(s_n)$. However, the notation $t_n = O(s_n)$ became customary in computer science, thus we are going to use it.

Big O notation - example

Example

$t_n = 6n^4 + 20n^2 + 2000$ is $O(n^4)$.

To prove this, it suffices to notice that for $n \geqslant 1$ it holds that $6n^4 + 20n^2 + 2000 \leqslant 6n^4 + 20n^4 + 2000n^4 \leqslant 2026n^4$.
By the way, $6n^4 + 20n^2 + 2000 \leqslant 2026n^4 \leqslant 2026n^5$, thus t_n is also $O(n^5)$. On the other hand, t_n is not, for example, $O(n^3)$, because for every C, if $n > \frac{C}{6}$, then $6n^4 + 20n^2 + 2000 > 6n^4 = 6n \cdot n^3 > Cn^3$.

Theorem about limits and big O notation

Limits and big O

Let $(t_n)_{n=1}^{\infty}$ and $(s_n)_{n=1}^{\infty}$ be sequences of non-negative values such that $\lim\limits_{n\to\infty} \frac{t_n}{s_n}$ exists. Then: I. If $\lim\limits_{n\to\infty} \frac{t_n}{s_n} = 0$, then $t_n = O(s_n)$ but $s_n \neq O(t_n)$.
II. If $\lim\limits_{n\to\infty} \frac{t_n}{s_n} = \infty$, then $s_n = O(t_n)$, ale $t_n \neq O(s_n)$.
III. If $\lim\limits_{n\to\infty} \frac{t_n}{s_n} \in (0, \infty)$, then $t_n = O(s_n)$ and $s_n = O(t_n)$.

This theorem can be directly and immediately derived from the definition of big O notation and the definition of a limit of a sequence.

Big O notation - examples

Example

$\log n = O(n)$ but $n \neq O(\log n)$.

It is sufficient to notice that $\log n = \frac{\ln n}{\ln 2}$, use the fact calculated in the previous presentation: $\lim_{n\to\infty} \frac{\ln n}{n} = 0$, and use the theorem about Big O and limits.

Example

$t_n = 6n^4 + 20n^2 + 2000$ is $O(n^4)$.

$$\lim_{n\to\infty} \frac{6n^4 + 20n^2 + 2000}{n^4} = 6 \in (0, \infty).$$

Big O notation - example

Complexity of polynomials

If t_n is a polynomial of degree k, then t_n is $O(n^k)$ and is not $O(n^j)$ for $j < k$.

The proof follows the rules of calculating limits of type "polynomial divided by polynomial".
The observation above is the reason why we earlier treated algorithms of run times $\frac{n(n-1)}{2} \cdot t_2$ and $(n-1)^2$ as "algorithms of quadratic time complexity". Their run time is $O(n^2)$.
The following question arises: how the big O notation is applied to ordering the time complexity of algorithms?

Time computational complexity

Time computational complexity

Let a_n and b_n be run time sequences of algorithms A and B (respectively) and $a_n = O(b_n)$ but $b_n \neq O(a_n)$. Then, we say that the algorithm A has lower *time computational complexity* than the algorithm B.

For short, we will be using the notion of *time complexity* or *computational complexity*. We will not study other types of computational complexity during this lecture.
Typically, if A has lower time complexity than B, we prefer to use the algorithm A. However, this is not always the case: maybe we do not use large sets of data for this problem and then the algorithm B does not even have to be slower. It can also make up for faster run time growth with other qualities (simpler code, lower memory usage etc.)

Example - potatoes

We recall the two examples of "potato algorithms". For the sack of n potatoes, the algorithm of searching for a rotten potato has a run time of $n\tau_1$ ($\tau_1 \in \mathbb{R}_+$) and the algorithm of searching for two potatoes of the same weight has a run time of $\frac{n(n-1)}{2}\tau_2$ ($\tau_2 \in \mathbb{R}_+$). We calculate:

$$\lim_{n\to\infty} \frac{n\tau_1}{\frac{n(n-1)}{2}\tau_2} = \lim_{n\to\infty} \frac{2\tau_1}{(n-1)\tau_2} = 0.$$

Thus, $n\tau_1 = O(\frac{n(n-1)}{2}\tau_2)$ but $\frac{n(n-1)}{2}\tau_2 \neq O(n\tau_1)$. Therefore, the first algorithm has lower (time) computational complexity.

Example sorting

The bubble sort algorithm has a maximal run time of $t_1(n-1)^2$, while the merge sort algorithm has run time of $t_2 n \log n$, $t_1, t_2 \in \mathbb{R}_+$. The first one has higher computational complexity because:

$$\lim_{n\to\infty} \frac{t_1(n-1)^2}{t_2 n \log n} = \frac{t_1}{t_2} \lim_{x\to\infty} \frac{(x-1)^2}{x \log x} \overset{H}{=} \frac{t_1}{t_2} \lim_{x\to\infty} \frac{2x-2}{\log x + \frac{1}{\ln 2}} \overset{H}{=}$$

$$\overset{H}{=} \frac{t_1}{t_2} \lim_{x\to\infty} \frac{2}{\frac{1}{x \ln 2}} = \infty.$$

Therefore, for large sets of data, the merge sort will run faster than the bubble sort.

Example - systems of linear equations

Solving systems of linear equations by the Gauss-Jordan method has a run time of an^3, while solving them by the Cramer's rule has a run time of $bn!$ where n is a number of equations and $a, b \in \mathbb{R}$.
We calculate:

$$\lim_{n\to\infty} \frac{an^3}{bn!} = \lim_{n\to\infty} \frac{an^3}{bn(n-1)(n-2)(n-3)(n-4)!} = 0,$$

because $\lim_{n\to\infty} \frac{an^3}{bn(n-1)(n-2)(n-3)} = 0$. Therefore, the Gauss-Jordan method is much faster for large systems of equations.

Transitivity of computational complexity

Transitivity of computational complexity

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. Namely, the algorithm A has lower time complexity than the algorithm B, the algorithm B has lower time complexity than the algorithm B, then the algorithm A has lower time complexity than the algorithm B.

Thanks to transitivity, comparing algorithms by means of big O notation is reasonable and orders the time sequences (and thus the algorithms) into a certain hierarchy.

Theorem of hierarchy

Theorem of hierarchy of seqences

The ordering of most typical sequences by their big O notation (in such a way that each of them is O of all sequences placed after it) is as follows:

$$1, \log n, n^{\alpha_1}, n^{\alpha_2}, n, n \log n, n^{\beta_1}, n^{\beta_2}, a^n, b^n, n!, n^n,$$

for any positive $\alpha_1 < \alpha_2 < 1 < \beta_1 < \beta_2$ i $1 < a < b$.

Here, we denote by 1 a sequence such that each of its elements is 1, namely a constant sequence. The complexity $O(1)$ is achieved only for algorithms running in constant time independent of the size of data that we deal with.

Stirling's formula

Most of the ordering from the hierarchy theorem can be easily justified by already presented methods. However, for sequences of high computational complexity (placed close to the right end of the hierarchy), we can use the famous Stirling's formula for approximating factorials.

Stirling's formula

For large numbers n we may approximate:

$$n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

Formally:

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\ \left(\frac{n}{e}\right)^n} = 1.$$

Hierarchy - example

Stirling's formula

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1.$$

For example, from the Stirling's formula we can infer that:

$$\lim_{n\to\infty} \frac{n!}{n^n} = \lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} \cdot \lim_{n\to\infty} \frac{\sqrt{2\pi n}}{e^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}}{e^n} \stackrel{H}{=} 0.$$

Thus $n! = O(n^n)$ but $O(n^n) \neq n!$.

Hierarchy - remarks

Big O notation provides only the upper bound of the computational complexity. However, we are usually interested in an upper bound that is as small as possible. Recalling the example of $t_n = 6n^4 + 20n^2 + 2000$, we know that $t_n = O(n^4)$. By the hierarchy theorem, $t_n = O(n^5)$ and generally $t_n = O(n^k)$ for any $k > 4$, let alone $t_n = O(2^n)$ and $t_n = O(n!)$. However, all these bounds are just a consequence of the smallest bound, namely $O(n^4)$. Therefore, $t_n = O(n^4)$ is the most interesting answer to the question about the big O notation of t_n.

Generally, the earlier in our hierarchy im bardziej a run time sequence of a given algorithm is, the better (namely, the smaller its time complexity).

Hierarchy - polynomial time

Polynomial complexity

A sequence t_n is *of polynomial complexity* if $t_n = O(n^m)$ for some $m \in \mathbb{N}$.

There is a popular "rule of thumb" that only algorithms of polynomial time (namely, algorithms with run time sequences of polynomial complexity) are practically efficient when we potentially deal with large sets of data (this is known as the Cobham-Edmonds thesis). In practice, the most desirable (but not always achieveable) time complexities are $O(n)$ or $O(n \log n)$.

Polynomial and non-polynomial complexities

Example

Assume that a person solves a system of 3 linear equations in 5 minutes using the Cramer's rule and in 10 minutes using the Gauss-Jordan method. Assuming that the speed of conducting operations by this person is constant, estimate the time needed for this person to solve a system of 6 linear equations with both methods?

The run time of the Gauss-Jordan method is $O(n^3)$, thus solving the system of 6 equations with this method should take about $\frac{6^3}{3^3} = 8$ times longer than solving 3 equations, namely 80 minutes.On the other hand, the same ratio for the Cramer's rule ($O(n!)$) is $\frac{6!}{3!} = 120$, thus this person would need about 10 hours to solve a system of 6 equations with this method.

Arithmetics of big O notation

Generally, for each interesting algorithm from the lecture I am going to provide its time complexity but without any proofs. Computing the time complexity of given algorithms falls beyond the scope of this course.
However, it might be interesting to know how to perform arithmetics on big O notation.

Arithmetics of big O notation - theorem

Arithmetics of big O notation

The following claims hold:

a) If $b_n = O(a_n)$ and k is a constant, then $kb_n = O(a_n)$.

b) If $b_n = O(a_n)$ and $c_n = O(a_n)$, then $b_n + c_n = O(a_n)$.

c) If $b_n = O(a_n)$ and $d_n = O(c_n)$, then $b_n + d_n = O(\max\{a_n, c_n\})$.

d) If $b_n = O(a_n)$ and $d_n = O(c_n)$, then $b_n \cdot d_n = O(a_n \cdot c_n)$.

Arithmetics of big O notation - theorem

Arithmetics of big O notation

a) If $b_n = O(a_n)$ and k is a constant, then $kb_n = O(a_n)$.
b) If $b_n = O(a_n)$ and $c_n = O(a_n)$, then $b_n + c_n = O(a_n)$.

From a) we infer that if a procedure runs by repeating a certain algorithm a finite number of times (k times), then the time complexity of this procedure is the same as the time complexity of the underlying algorithm.
From b) we infer that if two algorithms have the same time complexity and a procedure consists of performing these two algorithms, then this procedure has the same time complexity as these algorithms.

Arithmetics of big O notation - theorem

Arithmetics of big O notation

c) If $b_n = O(a_n)$ and $d_n = O(c_n)$, then $b_n + d_n = O(\max\{a_n, c_n\})$.
d) If $b_n = O(a_n)$ and $d_n = O(c_n)$, then $b_n \cdot d_n = O(a_n \cdot c_n)$.

From c) we infer that generally if a procedure consists of performing these two algorithms, then this procedure has the same time complexity as the algorithm that has higher time complexity of the two.

From d) we infer if a procedure consists of repeating an algorithm a number of times determined by a second algorithm (for example, the first algorithm consists of a loop and each repetition of a loop requires performing the second algorithm), then the time complexity of this procedure is a product of time complexities of these two algorithms.

Last remark

As we have already repeated multiple times, we cannot obtain a lower bound on the run time from the notation big O. In particular, if an algorithm is $O(n^2)$, it can also be $O(n)$. In fact, any program that is $O(n)$ is also $O(n^2)$. There exist a notation similar to big O that represents the lower bound on the run time of a given algorithm which uses the symbol of big Ω. Again, this is beyond the scope of this lecture.