4-2015

Why Big-O and Little-O in Algorithm Complexity: A Pedagogical Remark

Olga Kosheleva
University of Texas at El Paso, olgak@utep.edu

Vladik Kreinovich
University of Texas at El Paso, vladik@utep.edu

Comments:
Technical Report: UTEP-CS-15-26

Математические структуры и моделирование
2015, вып. XX, с. 1–??

УДК 000.000

WHY BIG-O AND LITTLE-O IN ALGORITHM COMPLEXITY: A PEDAGOGICAL REMARK

O. Kosheleva, V. Kreinovich

In the comparative analysis of different algorithm, O- and o-notions are frequently used. While their use is productive, most textbooks do not provide a convincing student-oriented explanation of why these particular notations are useful in algorithm analysis. In this note, we provide such an explanation.

1. Formulation of the Problem

O is ubiquitous in algorithm analysis. In algorithm analysis, to gauge the speed of an algorithm, usually, O-estimates are used: e.g., the algorithm requires time $O(n)$, or $O(n^2)$, etc.; see, e.g., [1]. Often, o-estimates are also used.

Need for O and o is not clearly explained. In many textbooks, the need to consider O- and o- estimates is not clearly explained.

What we do in this paper. The main objective of this short paper is to fill this gap by providing the students with a simple and – hopefully – convincing explanation of why O and o estimates are natural.

2. Analysis of the Problem and the Explanation of O

What we really want. What we want is to estimate how fast the algorithm is, i.e., how much time $T(x)$ it takes on different inputs x.

Worst-case and average computation time. In some situations, e.g., in automatic control, we need to make decisions "in real time", i.e., within a certain period of time. For example, if a automatic car sees a suddenly appearing obstacle, it needs to compute the needed change in trajectory so as to have time to avoid the collision. In general, the computation time $T(x)$ is different for different inputs x, so we want to be sure that for all these inputs, this computation time does not exceed a given threshold. In other words, we need to make sure that the worst-case time $\max\limits_x T(x)$ does not exceed this threshold.

Of course, the computation time depends on the size of the input: e.g., to answer a query, we sometimes need to look at all the records in the corresponding database.

University of Texas at El Paso, El Paso, TX 79968, USA
E-mail: olgak@utep.edu, vladik@utep.edu

The larger the database, the more time it takes, so if we simply take the maximum over all possible inputs x, we get a meaningless infinity. So get a meaningful description of the algorithm's speed, it is natural to limit ourselves to inputs of a given length (e.g., length in bits), i.e., to consider the worst-case time

$$T^w(n) \stackrel{\text{def}}{=} \max\{T(x) : \text{len}(x) = n\}.$$

Once we know that we are within the time limit, a natural next thing to estimate is how much time overall we spend on the corresponding computations. This overall time is, in general, proportional to the number of times when we call our algorithm, so, in effect, what we want to estimate is the average computation time

$$T^{av}(n) \stackrel{\text{def}}{=} \sum_x p(x) \cdot T(x),$$

where $p(x)$ is the frequency with which the input x happens among all inputs of the given length $n = \text{len}(x)$.

How to estimate worst-case and average computation time: the main challenge. The actual time of an algorithm depends on what computer they are implemented on, since different computers have different times for different elementary operations (such as addition, multiplication, etc.).

As a result, if we compare algorithms by simply using worst-case and average computation times as defined earlier, we may get different results depending on the computer on which the two compared algorithms are implemented. It is desirable to come up with a way of comparing algorithms themselves, a way that would not depend on the underlying computer.

How to compare algorithms: main idea. We want to make the comparison of algorithms independent on the difference in times needed to perform different elementary operations on different computers.

If we knew the number of elementary operations of each type, we could simply:

- count the number of operations $t_i(x)$ of each type i,
- multiply this number $t_i(x)$ of operation by the computation time w_i needed for a single operation of this type, and then
- add up the times needed for all the types:

$$T(x) = \sum_i w_i \cdot t_i(x).$$

Since we want a value that does not depend on the times w_i, let us fix arbitrary time for each – the simplest idea is to assume that each elementary operation of each type take exactly one unit of time $w_i = 1$. In this case, what we are doing is simply counting the number of elementary operations $t(x) = \sum_i t_i(x)$.

Comment. As one can easily check, the comparison results remain the same whether we use one unit of time for all types or we use different times for different types. Therefore, for simplicity, it is convenient to use the simplest way and take all the times equal to 1.

What is the relation between computation time and overall number of operations. The overall number $t(x)$ of elementary operations on an input x is equal to the sum

$$t(x) = \sum_i t_i(x),$$

where $t_i(x)$ is the total number of elementary operations of type i.

In these terms, the overall computation time is equal to

$$T(x) = \sum_i w_i \cdot t_i(x),$$

where $w_i > 0$ is the time needed for a single elementary operation of type i.

Let $w = \min(w_1, w_2, \ldots) > 0$ denote the smallest of the times w_i, and let $M = \max(w_1, w_2, \ldots)$ denote the largest of these times. Then, for every i, we have

$$m \le w_i \le M.$$

Multiplying all three sides of this double inequality by $t_i(x)$, we conclude that

$$m \cdot t_i(x) \le w_i \cdot t_i(x) \le M \cdot t_i(x).$$

Adding up the terms corresponding to all possible types of elementary operations, we conclude that

$$\sum_i m \cdot t_i(x) \le \sum_i w_i \cdot t_i(x) \le \sum_i M \cdot t_i(x),$$

i.e., that

$$m \cdot t(x) \le T(x) \le M \cdot t(x).$$

By taking the maximum over all inputs x of length n, we conclude that

$$m \cdot t^w(n) \le T^w(n) \le M \cdot t^w(n),$$

where we denoted

$$t^w(n) \stackrel{\text{def}}{=} \max\{t(x) : \text{len}(x) = n\}.$$

Similarly, by taking the average over all inputs x of length n, we conclude that

$$m \cdot t^{av}(n) \le T^{av}(n) \le M \cdot t^{av}(n),$$

where we denoted

$$t^{av}(n) \stackrel{\text{def}}{=} \sum_x p(x) \cdot t(x).$$

These quantities $t^w(n)$ and $t^{av}(n)$ are known as, correspondingly, worst-case and average-case computational complexity.

How can we compare the algorithms. A natural way to compare the two algorithms is as follows. We say that an algorithm A is *faster* than an algorithm B if

- whenever we fix a computer running the algorithm B,
- we can find another computer on which – in the worst case or in the average case – the algorithm A is always faster

(assuming that in principle, we can always find computers which run as fast as we want).

For the worst-case computation time, this means that:

- for each computer running the algorithm B,
- we can find a computer for running the algorithm A for which $T_A^w(n) < T_B^w(n)$ for all n.

For the average-case computation time, this means that:

- for each computer running the algorithm B,
- we can find a computer for running the algorithm A for which $T_A^{av}(n) < T_B^{av}(n)$ for all n.

How can we describe these properties in terms of the worst-case and average computational complexity? To answer this question, let us consider the property that $T_A^w(n) < T_B^w(n)$ for all n.

We know that $m_A \cdot t^w(n) \le T_A^w(n)$ and that $T^w(B) \le M_B \cdot t_B^w(n)$. Thus, the desired inequality $T_A^w(n) < T_B^w(n)$ implies that $m_A \cdot t_A^w(n) < M_B \cdot t_B^w(n)$ for all n, i.e., equivalently, that for all n, we have

$$t_A^w(n) < C \cdot t_B^w(n),$$

where we denoted $C \stackrel{\text{def}}{=} \dfrac{M_B}{m_A}$.

Similarly, for the average-case complexity, we conclude that for all n, we have

$$t_A^{av}(n) < C \cdot t_B^{av}(n),$$

for the same constant C.

In both cases, we have two functions $f(n) > 0$ and $g(n) > 0$ with the property that for some $C > 0$, we have $f(n) < C \cdot g(n)$ fr all n. This property is abbreviated as $f = O(g)$. In these terms, the above conditions can be described, correspondingly, as

$$t_B^w(n) = O(t_A^w(n))$$

and

$$t_A^{av}(n) = O(t_B^{av}(n)).$$

Let us show that, vice versa, if one of these two properties is satisfied, then the algorithm A is faster than the algorithm B in the above sense. Indeed, let us assume that for some C and for all n, we have

$$t_A^w(n) < C \cdot t_B^w(n).$$

We have assumed that we can select the A-computer which is arbitrarily fast, i.e., for which the maximal computation time M_A can be as small as possible. Let us use this assumption and select a computer with

$$M_A \le \frac{m_B}{C}.$$

For this choice, we have $C \cdot M_A \le m_B$.

Then, from $T_A^w(n) \le M_A \cdot t_A^w(n)$ and $t_A^w(n) < C \cdot t_B^w(n)$, we conclude that

$$T^w(n) < C \cdot M_A \cdot t_B^w(n).$$

Due to our choice of M_A, we have $C \cdot M_A \le m_B$ and thus, $T_A^w(n) < m_B \cdot t_B^w(n)$. We know that $m_B \cdot t_B^w(n) \le T_B^w(n)$, and therefore, we can conclude that $T_A^w(n) < T_B^w(n)$ for all n – exactly what we wanted.

Similarly, for the average-case complexity, $t_A^w(n) = O(t_B^w(n))$ implies that, for an appropriately selected A-computer, $T_A^{av}(n) < T_B^{av}(n)$ for all n. So, we arrive at the following conclusion.

Conclusions.

- The algorithm A is faster than the algorithm B in terms of the worst-case complexity if and only if $t_A^w(n) = O(t_B^w(n))$.
- Similarly, the algorithm A is faster than the algorithm B in terms of the average-case complexity if and only if $t_A^{av}(n) = O(t_B^{av}(n))$.

These two results explain why O-notations are used to compare algorithms.

3. Explanation of o

Idea. Instead of *selecting* a fast computer for the algorithm A, we can consider comparing the implementations of the algorithms A and B on *arbitrary* computers. Of course, if the computer for A is much slower than the computer for B, then we cannot expect the computation of A to be faster than B on *all* the inputs, but we can require it for all *sufficiently long* inputs.

Towards a more precise description of this idea. Let us say that an algorithm A is *much faster* than an algorithm B if:

- for every implementation of A and
- for every implementation of B,
- there exists a threshold n_0 such that for $n \ge n_0$, we have $T_A^w(n) < T_B^w(n)$ – or, correspondingly, $T_A^{av}(n) < T_B^{av}(n)$.

Let us reformulate this definition in terms of the computational complexities $t^w(n)$ and $t^{av}(n)$. Let us start with the algorithms A and B operating on the same computer. In this case, there exists an n_0 for which, starting with this n_0, we

have $T_A^w(n) < T_B^w(n)$. Due to $m_A \cdot t_A^w(n) \le T_A^w(n)$ and $T_B^w(n) \le M_B \cdot t_B^w(n)$, this implies that $m_A \cdot t_A^w(n) \le M_B \cdot t_B^w(n)$, i.e., that

$$t_A^w(n) < C \cdot t_B^w(n),$$

where $C \stackrel{\text{def}}{=} \dfrac{M_B}{m_A}$.

For any real number $\varepsilon > 0$, we can now consider a new B-computer which is $\dfrac{C}{\varepsilon}$ times faster than the previous one. For this new B-implementation B', the computation times of elementary operations are $\dfrac{C}{\varepsilon}$ times smaller, and thus,

$$M_{B'} = M_B \cdot \frac{\varepsilon}{C}.$$

Let us apply the assumption that the algorithm A is much faster than the algorithm B to the original implementation of the algorithm A and to the new implementation B' of the algorithm B. We then conclude that there exists an n_0 for which for all $n \ge n_0$, we have $T_A^w(n) < T_{B'}^w(n)$. Similarly to the above, we can thus deduce that

$$t_A^w(n) < C' \cdot t_B^w(n),$$

where

$$C' \stackrel{\text{def}}{=} \frac{M_{B'}}{m_A} = \frac{\varepsilon}{C} \cdot \frac{M_B}{m_A} = \frac{\varepsilon}{C} \cdot C = \varepsilon.$$

The resulting inequality $t_A^w(n) < \varepsilon \cdot t_B^w(n)$ implies that

$$\frac{t_A^w(n)}{t_B^w(n)} < \varepsilon.$$

Thus, for every $\varepsilon > 0$, there exists a natural number n_0 such that for all $n \ge n_0$, we have $0 < \dfrac{t_A^w(n)}{t_B^w(n)} < \varepsilon$. This is exactly the definition of a positive sequence having zero as the limit. So, we conclude that if the algorithm A is much faster than the algorithm B, then the ratio $\dfrac{t_A^w(n)}{t_B^w(n)}$ tends to 0. This is what is denoted by

$$t_A^w(n) = o(t_B^w(n)).$$

Similarly, the fact that A is much faster than B in terms of the average computation time implies that

$$t_A^{av}(n) = o(t_B^{av}(n)).$$

Let us show that, vice versa, if $t_A^w(n) = o(t_B^w(n))$, then the algorithm A is much faster than the algorithm B – in the sense of the above definition. Indeed, if

$$\frac{t_A^w(n)}{t_B^w(n)} \to 0,$$

this means that for every $\varepsilon > 0$, there exists an n_0 such that for all $n \geq n_0$, we have $\dfrac{t_A^w(n)}{t_B^w(n)} < \varepsilon$, i.e., equivalently, $t_A^w(n) < \varepsilon \cdot t_B^w(n)$.

Let us now consider that the algorithms A and B are implemented on some computers:

- the algorithm A is implemented on a computer with parameters m_A and M_A, and
- the algorithm B is implemented on a computer with parameters m_B and M_B.

Let us take $\varepsilon \stackrel{\text{def}}{=} \dfrac{m_B}{M_A}$. Then, $t_A^w(n) < \varepsilon \cdot t^w(B)$ means that $t_A^w(n) < \dfrac{m_B}{M_A} \cdot t^w(B)$, i.e., equivalently, that

$$M_A \cdot t_A^w(n) < m_B \cdot t_B^w(n).$$

We know that $T_A^w(n) \leq M_A \cdot t_A^w(n)$ and that $m_B \cdot t_B^w(n) \leq T_B^w(n)$. Thus, we conclude that for all $n \geq n_0$, we have $T_A^w(n) < T_B^w(n)$. This is exactly what it means for the algorithm A to be much faster than the algorithm B.

A similar conclusion can be made about average-case computations time, so we arrive at the following conclusions.

Conclusions.

- The algorithm A is much faster than the algorithm B in terms of the worst-case complexity if and only if $t_A^w(n) = o(t_B^w(n))$.
- Similarly, the algorithm A is much faster than the algorithm B in terms of the average-case complexity if and only if $t_A^{av}(n) = o(t_B^{av}(n))$.

These two results explain why o-notations are used to compare algorithms.

Acknowledgments.

This work was supported in part by the National Science Foundation grants HRD-0734825 and HRD-1242122 (Cyber-ShARE Center of Excellence), and DUE-0926721.

The authors are thankful to all the participants of the North American Annual Meeting of the Association of Symbolic Logic (Urbana-Champaign, Illinois, March 25–28, 2015), especially to Yuri Gurevich, for valuable discussions.

References

1. Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2009.