Production Systems and Information Engineering
Volume 10 (3), pp. 98–104.
https://doi.org/10.32968/psaie.2022.3.8

AN APPROACH TO CLASSIFY ALGORITHMS BY COMPLEXITY

OLIVÉR HORNYÁK
University of Miskolc
Hungary Institute of Information Technology
oliver.hornyak@uni-miskolc.hu

Abstract. This paper investigates computer algorithms complexity, which takes an important role in software design. An algorithm is a finite set of instructions, those if followed, accomplishes a particular task. It is not language specific; any language and symbols can represent instructions. While complexity is usually in terms of time, sometimes complexity is also analyzed in terms of space, which translates to the algorithm's memory requirements. The paper gives an overview of the most widely used O notation. Experienced programmers can evaluate the time and memory complexity of a block of source code; however, this is not possible when the algorithm is available in the form of an executable. In this paper a method is proposed to evaluate algorithms without having the source code. The potential drawbacks of the proposal are also considered.

Keywords: algorithm, complexity

1. Introduction

Computers run programs that are representations of an algorithm. Computational complexity refers to measures to evaluate the efficiency of algorithms [6]. The need for writing efficient algorithms is very old. Euclid worked on finding the greatest common divisor of two integers in his Book VII [7]. This is said to be the oldest known algorithm which is still in use. As you can see the study of investigating the available resources is back to 300 BC.

The complexity theory provides the theoretical estimates for the resources needed by an algorithm to solve any computational task. Two resource types are investigated typically: time and space complexities [3]. Time complexity is defined as the amount of time requires executing each statement of the algorithm.

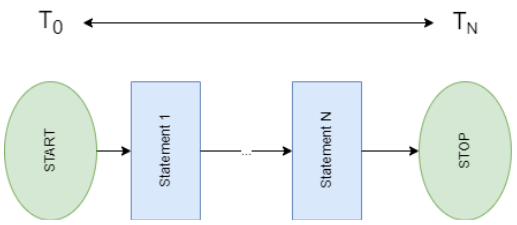

Figure 1. Time complexity

Space complexity refers to the memory space used by the algorithm. It consists of the space of the input parameters and the temporary space required by algorithm – the later one is celled auxiliary space.

Figure 2. Space complexity

1.1. Literature review

In this section the textbooks on mathematical complexity are reviewed. Turing introduced a concept called Turing machines [12]. It serves as an idealized computer model. In his paper he focused on computability theory in general rather than complexity theory. Alternating Turing machines were introduced independently by [13], they introduced time and space complexity. Balcázar et al. [9], [10] focuses on structural complexity. An overview is given on the complexity classes in [11]. The O-notation, which will be explained in the next chapter was introduced by [14]. This is the most widely used mathematical notation. Specific complexity measures such as time and space complexity were first analyzed systematically by [15]. Computational difficulty of functions was studied by [16] Quantitative aspects of the computation were studied by [17] to find an efficient algorithm for graph problems. The gap theorem was independently proved by [18] and [19] stating that there are large computable gaps in the hierarchy of complexity classes. The following complexity classes can be identified:

$$NL \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{EXSPACE}$$

where

- NL stands for Nondeterministic Logarithmic space (that can be solved by a nondeterministic Turing machine using a logarithmic amount of memory space),
- P contains problems that can be solved by a deterministic Turing machine using a polynomial time,
- NP stands for nondeterministic polynomial time that can be solved in polynomial time by a nondeterministic Turing machine,
- PSPACE is the set of all decision problems that can be solved by a Turing machine using a polynomial amount of space,
- EXPTIME is the set of problems that are solvable by a deterministic Turing machine in exponential time,

- EXPSPACE denotes the problems solvable by a deterministic Turing machine in exponential space,
- $\subseteq$ denotes the subset relation.

Relations between time and space complexity is investigated in [11]. Complexity results for multiprocessor is investigated in [20].

1.2. Notations for computer algorithm complexity

The notation introduced in [14] which is used to describe the complexity is as follows. A function f is said to be $O(g)$ if here is a constant $c > 0$ and a natural number n_0 such that

$$f(n) \leq c\, g(n) \text{ for all } n \geq n_0,\ c > 0\ .$$

Some properties of the complexity measures.

- Constant multiplication does not modify complexity
  If $f_1(n) = c\, f_2(n)$ then $O(f_1(n)) = O(f_2(n))$
- Independent consecutive statements ignore non-significant term
  Let t_1 be the cost of running f_1 and t_2 be the cost of running f_2.
  The total time $t = t_1 + t_2$ and $O(f_1 + f_2) = \max(O(f_1), O(f_2))$
- Let' have the following polynomial function:
  $f(n) = a_m n^m + a_{m-1} n^{m-1} + .. + a_1 n + a_0$ then
  $O\big(f(n)\big) = O(n^m)$
- If $f_1(n) = \log_a n$ and $f_2(n) = \log_b n$ then $O(f_1(n)) = O(f_2(n))$

$O(g)$ notation is a worst-case complexity number as it represents the upper bound of the running time of an algorithm.

The notation is sometimes referred a Big O notation. Its advantage that it abstracts away constant order differences in efficiency which can vary from platform, language, OS to focus on the inherent efficiency of the algorithm and how it varies according to the size of the input [21]. Its limitation is that There are numerous algorithms that are too difficult to analyze mathematically. There may not be sufficient information to calculate the behaviour of the algorithm in an average case.

Some typical complexity measures:

Table 1. Typical complexities

Complexity	**Usage**
O(N), O(log N), O(1)	greedy algorithms, typical mathematical algorithms
O(N log N)	Sorting, binary search
$O(N^2)$	Graph, tree
$O(2^N N)$	Bit manipulation
O(N!)	Recursive algorithms

2. Asymptotic complexity calculator

As you can see the calculation of complexity can be difficult as it may depend on run-time parameters. In practice, the exact value is not necessary. Furthermore, the resource usage is critical for large n values only [4].

In this section the algorithm of a runtime complexity calculator will be specified. The problem is formularized as follows. The calculated complexity of the algorithm ($C(n)$) is defined in the following form [5]:

$$C(n) = \ a\, O\left(\frac{n}{b}\right) + O(n^k(\log n)^i)$$

You can note, that in O notation [14] the parameter b is always one so the equation can be reduced as:

$$C(n) = \ a\, O(n) + O(n^k(\log n)^i)$$

So that the calculator needs to find the $(a, k, i) \in R$ values that describes the algorithm best, however in practice, we can assume that $(a, k, i) \in \{0,1,2,3.\}$. This assumption reduces the search space significantly, while still can indicate the complexity classification of the algorithm examined.

Let's have a runtime algorithm that works on n input values. Let's create N random samples, denoted as n_1, n_2, ...n_N. Measure time- and memory consumption: t_j and m_j while executing the algorithm on each sample n_j, where j=1..N. Also calculate the complexity using the forementioned equation. Use least square error method to find the best fitting curve as you can see in Figure 3.

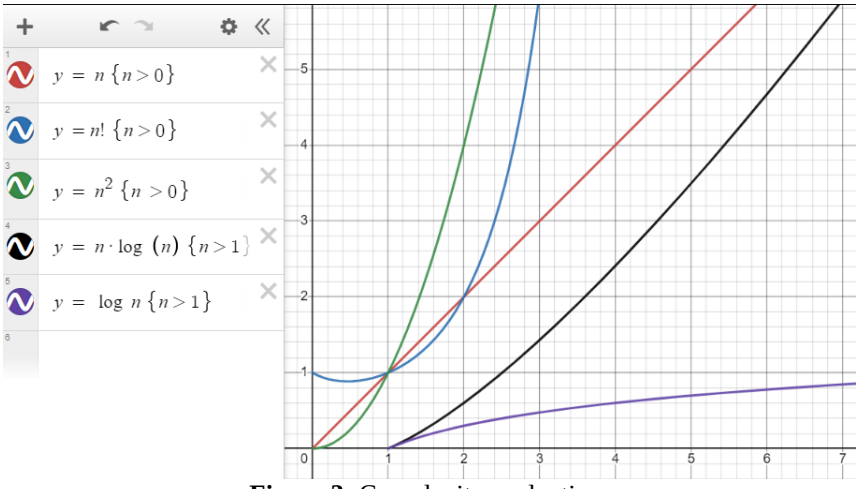

Figure 3. Complexity evaluation

Calculate the error i.e.: the square of the difference of the measured and calculated complexity value for each (a, k, i) indexes. The best fitting complexity curve at the $(a_{min}, k_{min}, i_{min})$ indexes.

The meta-algorithm of the calculator for the time complexity is as follows:

$error_{a,k,i} = 0$

for all (a, k, i) ∈ {1, 2, 3}

  for all j ∈ {1..N}

    calculate $C(n_j) = a\ O(n_j) + O(n_j^k (\log n_j)^i)$

    $error_{a,k,i} = (C(n_j) - t_j)^2$

find min $error_{a,k,i}$

display the (a,k,i) indexes of the minimum value

The same can be performed on memory consumption figures.

The algorithm can be improved bey introducing a weight factor for the error calculation, so that $(O(n_j) - t_j)^2$ to be a w_j where the larger n_j is the bigger w_j is. In other words, the fitting of larger n values is more important.

2.1. Limitations of the algorithm

The biggest limitation of the complexity calculation algorithm that it may become unresponsive: it can reach a large n value where the running time or memory consumptions exceeds the limitation of the host computer it uses.

Another limitation is that in practice almost all inputs have a limit: numerical variables have a variable type like int, long, float or double so you can not test against any arbitrary number of inputs.

It can be difficult to limit your computer to run only the algorithm to test. It is expedient to have a virtual running environment that allows a separated running environment to be defined. Figure 5 depicts the architecture of the runner.

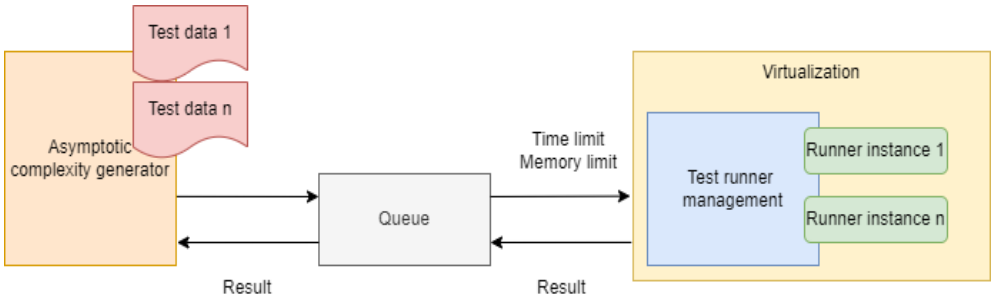

Figure 5. Architecture of the runner

Using this approach, the proposed algorithm remains responsible for large n values, although it won't return any value when the resource limits are exceeded. So, you will have test results for small n values where no code optimization is needed. The lack of results can be a bad smell that indicates that the algorithm is not optimal.

3. Summary

In this paper an overview was given on the calculation of time and memory complexity of software algorithms. A method was proposed that makes an estimation of the complexity of an executable by running the algorithm to be investigated. This approach is very useful when the source code is not available, it investigated the executable. Also, for unexperienced programmers it may indicate the need for optimization. The earlier it is detected, the easier to fix the performance problems.

References

[1] Kruskal, C. P., Rudolph, L., Snir, M.: A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 1990, 71, 1, pp. 95–132.

[2] Xu, M., Li, T., Wang, Z., Deng, X., Yang, R. Guan, Z.: Reducing Complexity of HEVC: A Deep Learning Approach. In *IEEE Transactions on Image Processing (TIP)*, vol. 27, no. 10, pp. 5044–5059, Oct. 2018. https://doi.org/10.1109/TIP.2018.2847035

[3] Li, T., Xu, M., Deng, X.: A deep convolutional neural network approach for complexity reduction on intra-mode HEVC. *2017 IEEE International Conference on Multimedia and Expo (ICME)*, Hong Kong, Hong Kong, 2017, pp. 1255–1260. https://doi.org/10.1109/ICME.2017.8019316

[4] JCT-VC: *HM Software*. https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags /HM-16.5/.

[5] Bentley, J. L., Haken, D., Saxe, J. B. (September 1980): A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12 (3), pp. 36–44. https://doi.org/10.1145/1008861.1008865

[6] Huang, S.: What is Big O Notation Explained: Space and Time Complexity. https:// www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-167 4 cfa8a23c/

[7] Euclid: *The Elements: Books I–XIII – Complete and Unabridged*. Translated by Sir Thomas Heath, Barnes & Noble, 2006.

[8] Kades, E.: The laws of complexity and the complexity of laws: the implications of computational complexity theory for the law. *Rutgers L. Rev*., 49 (1996), p. 403.

[9] Balcazar, J. L., Diaz, J., Gabarr, J.: Structural Complexity I, volume 11 of. EATCS Monographs on Theoretical Computer Science, 1988.

[10] Balcázar, J. L., Díaz, J., Gabarró, J.: *Structural complexity II*. Vol. 22. Springer Science & Business Media, 2012.

[11] Bovet, D. P., Crescenzi, P., Bovet, D.: Introduction to the Theory of Complexity. London, Prentice Hall, 1994.

[12] Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Society*, ser. 2, 42, 1936, pp. 230–265.

[13] Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. *Journal of ACM*, 28, 1981, pp. 114–133.

[14] Knuth, D.E.: The art of computer programming. Vol. 1. Fundamental Algorithms, Addison-Wesley, 1968.

[15] Hartmanis, J., Stearns, R.E.: On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117, 1965, pp. 285–306.

[16] Cobham, A.: The intrinsic computational difficulty of functions. *Proc. Congress for Logic, Mathematics, and Philosophy of Science*, 1964, pp. 24–30. https://doi.org/10.2307/2270887

[17] Edmonds, J.R.: Paths, trees and flowers. *Canadian Journal of Mathematics*, 17, 1965, pp. 449–467, https://doi.org/10.4153/CJM-1965-045-4.

[18] Trakhtenbrot, B. A. (1967). The Complexity of Algorithms and Computations (Lecture Notes). Novosibirsk University.

[19] Borodin, A.: Complexity Classes of Recursive Functions and the Existence of Complexity Gaps. *Proc. of the 1st Annual ACM Symposium on Theory of Computing*, 1969, pp. 67–78, https://doi.org/10.1109/SWAT.1969.4.

[20] Garey, M. R., Johnson, D. S.: Complexity results for multiprocessor scheduling under resource constraints. *SIAM journal on Computing*, 4.4 (1975), pp. 397–411. https://doi.org/10.1137/0204035

[21] *What is the big deal about Big-O notation in computer science? https://stackoverflow.com/questions/1996457/what-is-the-big-deal-about-big-o-notation-in-computer-science* (Last accessed 04.10. 2022)