

# Counting Independent Terms in Big-Oh Notation

Fabiano de S. Oliveira<sup>1\*</sup>  
Valmir C. Barbosa<sup>2</sup>

<sup>1</sup>Instituto de Matemática e Estatística  
Universidade do Estado do Rio de Janeiro  
Rua São Francisco Xavier, 524, sala 6019B  
20550-900 Rio de Janeiro - RJ, Brazil

<sup>2</sup>Programa de Engenharia de Sistemas e Computação, COPPE  
Universidade Federal do Rio de Janeiro  
Caixa Postal 68511  
21941-972 Rio de Janeiro - RJ, Brazil

## Abstract

The field of computational complexity is concerned both with the intrinsic hardness of computational problems and with the efficiency of algorithms to solve them. Given such a problem, normally one designs an algorithm to solve it and sets about establishing bounds on its performance as functions of the algorithm's variables, particularly upper bounds expressed via the big-oh notation. But if we were given some inscrutable code and were asked to figure out its big-oh profile from performance data on a given set of inputs, how hard would we have to grapple with the various possibilities before zooming in on a reasonably small set of candidates? Here we show that, even if we restricted our search to upper bounds given by polynomials, the number of possibilities could be arbitrarily large for two or more variables. This is unexpected, given the available body of examples on algorithmic efficiency, and serves to illustrate the many facets of the big-oh notation, as well as its counter-intuitive twists.

**Keywords:** Analysis of algorithms, Asymptotics, Big-oh notation, Computational complexity.

---

\*Corresponding author (fabiano@gmail.com).

# 1 Introduction

Computer algorithms require resources in order to run, most notably, time (the number of steps they must go through before termination) and space (the number of memory cells where their input and intermediate results are to be stored). In general, any given run of an algorithm may require a different amount of such resources even if the algorithm is fully deterministic, since both time and space usage depend heavily on the input to the algorithm.

This dependence on the input is quantified by means of what here we call an algorithm's *variables*, that is, numbers that explicitly or implicitly are part of any input to the algorithm and can affect its resource requirements for computing on that particular input. For instance, while the number of steps required by some algorithms for sorting an array of integers depends on the size of the array (this being a piece of information that probably is an explicit part of the input but in any case could easily be derived from it), the greatest integer in the array does not affect the number of steps. The latter holds under the assumption that each individual integer can be stored in a single processor register, which seems reasonable given that it is true of any modern processor for integers up to 4 billion.

The amount of time or space required by an algorithm is expressed as a function of its variables. In the sorting example, both time and space are non-decreasing functions of the single variable representing the number of integers in the input. As it happens, though, in most cases determining an algorithm's exact time or space function is a rather difficult task. To circumvent some of this difficulty while still allowing for meaningful statements about the algorithm's performance to be made, the commonly accepted practice has been to express such functions, through the so-called *big-oh* notation, in asymptotic terms.

**Definition (big oh, one variable).** *Let  $f(n)$  and  $g(n)$  be real functions. We say that  $f(n) = O(g(n))$  if there exist positive constants  $c$  and  $n_0$  such that  $f(n) \leq cg(n)$  for all  $n \geq n_0$ .*

The big-oh notation is in fact more than simply a notation, since the elements that go in its definition allow for a cleaner view of an algorithm's inner workings as far as its resource requirements are concerned. Thus, instead of seeking to determine, say, the time function  $f(n)$  of an algorithm, what one does is to look for some  $g(n)$  that, for sufficiently large  $n$ , is proportional to an upper bound on  $f(n)$ . If such an upper bound is "tight" (i.e., if it does reflect the time requirement of the worst-case inputs to the algorithm), then several conclusions can be drawn with the help of  $g(n)$ . For example, if algorithms  $A$  and  $B$  admit tight upper bounds on their time functions, respectively  $O(n^2)$  and  $O(n \log n)$ , then the worst-case performance of algorithm  $B$  is preferable to that of  $A$  for sufficiently large  $n$ .

It often happens for an algorithm's time and space functions to depend on more than one variable, each representing a different aspect of the inputs to the algorithm. This occurs routinely in the case of algorithms on graphs, since very commonly such algorithms' resource requirements are influenced by both

the graph's number of vertices and its number of edges, but it may also occur more indirectly. As an example, consider an algorithm  $A$  for sorting a set of  $n$  integers in  $O(n^2)$  time. Suppose further that an input to  $A$  comes in the form of two disjoint sets, say  $X$  and  $Y$ , respectively of sizes  $x$  and  $y$ , and that for reasons that have to do with differentiating elements in  $X$  from those in  $Y$  in some application, the natural way to express the running time of  $A$  is by making explicit use of  $x$  and  $y$ , as in  $f(x+y)$ , rather than coalescing them as a sum into the variable  $n$  and using  $f(n)$  instead. Proceeding in this way would lead to  $f(x+y) = O((x+y)^2) = O(x^2 + 2xy + y^2)$ , but clearly these expressions are clamoring for the big-oh notation to be extended to the two-variable case.

**Definition (big oh, two variables).** *Let  $f(n_1, n_2)$  and  $g(n_1, n_2)$  be real functions. We say that  $f(n_1, n_2) = O(g(n_1, n_2))$  if there exist positive constants  $c$ ,  $n_{1,0}$ , and  $n_{2,0}$  such that  $f(n_1, n_2) \leq cg(n_1, n_2)$  for all  $n_1, n_2 \geq 0$  satisfying  $n_1 \geq n_{1,0}$  or  $n_2 \geq n_{2,0}$ .*

With the extended definition in hand, we can express the algorithm's time function as  $f(x, y) = O(x^2 + 2xy + y^2)$  and finally see that, in reality,  $f(x, y) = O(x^2 + y^2)$ . This is so because  $2xy = O(x^2)$  for all valuations of  $x$  and  $y$  in which  $y \leq x$  and  $2xy = O(y^2)$  for those in which  $x \leq y$ .

This example is interesting also in that it highlights the advantage of being concise when using the big-oh notation: saying that  $f(x, y) = O(x^2 + y^2)$  is no less informative than saying that  $f(x, y) = O(x^2 + 2xy + y^2)$ , but is more useful to a potential user of the algorithm in question. Motivated by this observation, we equate conciseness with irreducibility, the latter defined as follows. First, let a *term* be the product of single-variable functions; e.g.,  $x^2$ ,  $2xy$ , and  $y^2$  are all terms. Given the  $k$  terms  $T_1, \dots, T_k$ , we say that term  $T_i$  is *independent* with respect to the sum  $S = T_1 + \dots + T_k$  if it is not asymptotically bounded from above in proportion to  $S - T_i$ , that is, if  $T_i = O(S - T_i)$  does not hold. We say that  $S$  is *irreducible* if all of  $T_1, \dots, T_k$  are independent. In the example,  $x^2 + y^2$  is irreducible but  $x^2 + 2xy + y^2$  is not.

In this article we concern ourselves with functions that, like  $S$ , can be written as a sum of terms. Our goal is to answer a specific question motivated by the following practical application. Suppose we are given the executable code for some program, along with the list of variables affecting its performance, but no further information (no source code, no time or space function, not even their big-oh forms). Suppose further that, before putting such code to use, we are tasked with estimating a bound on its time (or space) function as concisely as possible, via the big-oh notation, over a given range of the variables. If we were allowed to restrict our search to those time functions that can be expressed as a sum of terms, then knowing beforehand how many terms there can be in a concise representation thereof would be of great help. The question that interests us is then, how many terms can an irreducible sum-of-terms function have?

## 2 One variable, and beyond

Answering this question becomes easier if we make further assumptions, now regarding the nature of the single-variable factors that make up a term. If the sum-of-terms function in question is itself a function of a single variable, then the further assumption is quite reasonable and refers to restricting those factors to be one of the functions that commonly appear in algorithmic analysis: polynomial, polylogarithmic, logarithmic, and exponential. This given, the question is answered quite simply in the single-variable case, in which no sum of at least two terms constitutes an irreducible function.

We see this more clearly by following Hardy [1], who defined the class  $\mathcal{L}$  of *logarithmico-exponential functions* to be the one comprising the following functions:

- $f(n) = a$ , for any real constant  $a$ ;
- $f(n) = n$ ;
- $f(n) - g(n)$ , if  $f(n), g(n) \in \mathcal{L}$ ;
- $e^{f(n)}$ , if  $f(n) \in \mathcal{L}$ ;
- $\ln f(n)$ , if  $f(n) \in \mathcal{L}$  and, for some constant  $n_0$ ,  $f(n) > 0$  for all  $n \geq n_0$ .

As it turns out, any function arising naturally when analyzing an algorithm belongs to  $\mathcal{L}$  [2]. For example,  $4n^3 + (\log n)^5 + 2^{n^2}$  can be seen to be in  $\mathcal{L}$  by using the defining conditions for  $\mathcal{L}$  as follows:

- $f(n) + g(n) = f(n) - (0 - g(n)) \in \mathcal{L}$  if  $f(n), g(n) \in \mathcal{L}$ ;
- $f(n)g(n) = e^{\ln f(n) + \ln g(n)} \in \mathcal{L}$  if  $f(n), g(n) \in \mathcal{L}$ ;
- $f(n)^k = \prod_{i=1}^k f(n) \in \mathcal{L}$  if  $f(n) \in \mathcal{L}$ ;
- $kf(n) = \sum_{i=1}^k f(n) \in \mathcal{L}$  if  $f(n) \in \mathcal{L}$ ;
- $2^{f(n)} = e^{\ln 2^{f(n)}} \in \mathcal{L}$  if  $f(n) \in \mathcal{L}$ .

An important consequence of Hardy's work is that, given any two functions  $f(n)$  and  $g(n)$  in  $\mathcal{L}$ , we have  $f(n) = O(g(n))$  or  $g(n) = O(f(n))$ . Therefore, if  $f(n)$  is the sum of at least two terms, each one in  $\mathcal{L}$ , then  $f(n)$  is not irreducible.

And how about sum-of-terms functions having more than one variable? As noted above, multiple variables are a common occurrence in graph algorithms, whose time and space functions often depend on both  $n$  and  $m$ , the graph's numbers of vertices and edges, respectively. In fact, some of the best algorithms for numerous graph problems have time functions bounded by sums of two or three terms, often depending on more variables than simply  $n$  and  $m$ , as shown in Table 1. What we see in the table are counts of how many algorithms, as reported in a portion of [3], have time-function bounds with a certain number of terms on a certain number of variables. For example, 65 of the reported bounds

Table 1: Number of time-function bounds reported in the ‘‘Complexity survey’’ subsections of [3], considering the books’ Parts I and II only.

Number of variables	Number of terms		
	1	2	3
1	52	0	0
2	65	14	0
3	48	33	2
4	5	14	0
5	3	2	0
Total	173	63	2

on two variables have one single term and 14 have two, but none has three or more terms.

One might then wonder if two is the maximum number of terms whose sum is irreducible in the case of two variables. That, however, is not the case. To see this, consider the function  $f(x, y) = x^2 + y^2 + (xy)^{3/2}$ . The first term in this function is independent, since it does not hold that  $x^2 = O(y^2 + (xy)^{3/2})$ , as seen by simply fixing  $y = c$  for any positive constant  $c$ . The case of the second term is entirely analogous. As for the third term, set  $x = y$  to conclude that  $(xy)^{3/2} = O(x^2 + y^2)$  does not hold either. So  $f(x, y)$  is irreducible despite having more than two terms.

We may loosen the conjecture a little, and set about testing whether three, not two, is the maximum number of terms in an irreducible sum of terms. But once again, we are in no luck: the four-term sum  $f(x, y) = x^{485}y + x^{477}y^4 + x^{459}y^8 + x^{243}y^{32}$ , for example, is irreducible. This can be seen by setting, for each term in order,  $x = y^{0.7}$ ,  $x = y^{0.31}$ ,  $x = y^{0.21}$ , and  $x = y^{0.05}$ . So conjecturing further seems to have become a little too daunting and perhaps we should back off and consider the possibility that a function may in fact comprise an arbitrarily large number of terms and still be irreducible. Next we prove that this is the case when all the single-variable factors that go in a term are rising power laws, even if the exponents in these laws are all positive integers (i.e., the sum-of-terms function is a polynomial).

### 3 An arbitrarily large number of terms

Let  $f(x, y)$  be such that

$$f(x, y) = \sum_{i=1}^k x^{a_i} y^{b_i}.$$

If  $f(x, y)$  is to be an irreducible function, then clearly no two of the  $a_i$ ’s may equal each other, and similarly no two of the  $b_i$ ’s, since in either case one of the two terms involved would not be independent. Thus, it must be possible to arrange the  $a_i$ ’s into an increasing or decreasing sequence, and similarly the  $b_i$ ’s, but once again for the sake of irreducibility one of the sequences must be

increasing while the other is decreasing. We assume  $0 < a_1 < \dots < a_k$  and  $b_1 > \dots > b_k > 0$ .

For each  $i$ , we concentrate on valuations of  $x$  and  $y$  such that  $x = y^{z_i}$  for some  $z_i > 0$ , so the  $i$ th term of  $f(x, y)$  is independent if and only if  $a_i z_i + b_i > a_j z_i + b_j$  for all  $j \neq i$ . So arguing for the irreducibility of  $f(x, y)$  requires that we find  $a_1, \dots, a_k, b_1, \dots, b_k$ , and  $z_1, \dots, z_k$  such that

$$a_i z_i + b_i > a_j z_i + b_j$$

for all  $i$  and all  $j \neq i$ . Our approach will be to determine the  $a_i$ 's and the  $b_i$ 's in such a way as to automatically establish an interval within which to choose the value of each  $z_i$ . For  $1 \leq i, j \leq k$ , the constraints on this choice are

$$\begin{aligned} z_i &< (b_i - b_j)/(a_j - a_i) \text{ for } i < j, \\ z_i &> (b_i - b_j)/(a_j - a_i) \text{ for } j < i. \end{aligned}$$

A convenient, alternative way to view this condition is to define the ratio

$$r(i, j) = \frac{b_i - b_j}{a_j - a_i},$$

for which it holds that  $r(i, j) = r(j, i)$ . Using this equivalence whenever  $j < i$  allows the condition to be written as

$$\max_{1 \leq j < i} r(j, i) < z_i < \min_{i < j \leq k} r(i, j) \text{ for } 1 < i < k,$$

in addition to  $z_1 < \min_{1 < j \leq k} r(1, j)$  and  $z_k > \max_{1 \leq j < k} r(j, k)$ . So in order for  $f(x, y)$  to be irreducible, we must ensure that

$$\max_{1 \leq j < i} r(j, i) < \min_{i < j \leq k} r(i, j) \text{ for } 1 < i < k.$$

**Theorem 1.** *Given any positive integer  $k$ ,  $f(x, y)$  is irreducible with  $a_i = a_1(2 - \alpha^{i-1})$  and  $b_i = b_1\beta^{i-1}$ , where  $\alpha$  and  $\beta$  are constants such that  $0 < \alpha < \beta < 1 - \alpha < 1$ .*

*Proof.* We first write  $r(i, j)$  as

$$r(i, j) = \frac{b_1}{a_1} \left( \frac{\beta}{\alpha} \right)^{i-1} \frac{1 - \beta^{j-i}}{1 - \alpha^{j-i}}$$

and note that

$$r(i, j+1) = r(i, j) \frac{h_{i,j}(\beta)}{h_{i,j}(\alpha)},$$

where

$$h_{i,j}(t) = \frac{1 - t^{j-i+1}}{1 - t^{j-i}},$$

with first derivative given by

$$h'_{i,j}(t) = \frac{t^{j-i-1}((j-i)(1-t) - t(1-t^{j-i}))}{(1-t^{j-i})^2}.$$

For  $i < j < k$ , we have  $h'_{i,j}(t) > 0$  for  $t \in (0, 1)$ , since using

$$u_{i,j}(t) = (j-i)(1-t) - t(1-t^{j-i}),$$

we have  $u'_{i,j}(t) = (j-i+1)(t^{j-i} - 1) < 0$  and  $u_{i,j}(1) = 0$ , hence  $u_{i,j}(t) > 0$ . It follows that  $h_{i,j}(\beta)/h_{i,j}(\alpha) > 1$ , and therefore,  $r(i, j) < r(i, j+1)$ .

Moreover, we also have  $r(i-1, k) < r(i, i+1)$  for  $1 < i < k$ , which can be seen by noting that

$$r(i-1, k) < \lim_{j \rightarrow \infty} r(i-1, j) < r(i, i+1),$$

where

$$\lim_{j \rightarrow \infty} r(i-1, j) = \frac{b_1}{a_1} \left( \frac{\beta}{\alpha} \right)^{i-2}$$

and

$$r(i, i+1) = \frac{b_1}{a_1} \left( \frac{\beta}{\alpha} \right)^{i-1} \frac{1-\beta}{1-\alpha},$$

since  $\beta(1-\beta)/\alpha(1-\alpha) > 1$  for  $\alpha < \beta < 1-\alpha$ .

The desired inequality follows, since  $\max_{1 \leq j < i} r(j, i) = r(i-1, i) < r(i-1, k) < r(i, i+1) = \min_{i < j \leq k} r(i, j)$ .  $\square$

For the  $a_i$ 's and  $b_i$ 's of Theorem 1, following the proof reveals a clear recipe to determine the  $z_i$ 's: choose  $z_1 < r(1, 2)$ ,  $z_k > r(k-1, k)$ , and the remaining ones to satisfy

$$r(i-1, k) < z_i < r(i, i+1) \text{ for } 1 < i < k.$$

## 4 Integral exponents and constrained valuations

Our argument in the previous section for the irreducibility of the function  $f(x, y) = \sum_{i=1}^k x^{a_i} y^{b_i}$  relied on the particular valuation for  $x$  and  $y$  that sets  $x = y^{z_i}$ . All we have required of the exponents  $a_i$ ,  $b_i$ , and  $z_i$  is that they be positive, which leaves plenty of room for them to be nonintegers. This is not a problem in itself, and in fact there exist landmark algorithms that run in time bounded by the input size raised to an irrational power.<sup>1</sup> However, when it comes to the analysis of practical computer algorithms, in most cases we expect the exponents  $a_i$  and  $b_i$  to be positive integers.

Additionally, depending on the domain at hand, it is often the case that the valuation tying the  $x$  and  $y$  variables together should only employ values for

<sup>1</sup>One of the well-known examples of this in the single-variable case is Strassen's algorithm for multiplying two  $n \times n$  matrices, whose running time is  $O(n^{\log_2 7})$  [4].

Table 2: For  $f(x, y)$  as in Theorem 2, with  $k = 6$ ,  $p_\alpha = 1$ ,  $q_\alpha = 3$ ,  $p_\beta = 1$ , and  $q_\beta = 2$ , the cell at position  $j, i$  gives the value of  $a_j z_i + b_j$ . These values are highlighted in a bold typeface whenever  $j = i$ , indicating the maximum on each column (i.e., for fixed  $i$  and all  $j$ ).

$j$	$a_j$	$b_j$	$i: z_i$					
			1: 0.05	2: 0.14	3: 0.21	4: 0.31	5: 0.47	6: 0.70
1	243	32	<b>44.15</b>	66.02	83.03	107.33	146.21	202.10
2	405	16	36.25	<b>72.70</b>	101.05	141.55	206.35	299.50
3	459	8	30.95	72.26	<b>104.39</b>	150.29	223.73	329.30
4	477	4	27.85	70.78	104.17	<b>151.87</b>	228.19	337.90
5	483	2	26.15	69.62	103.43	151.73	<b>229.01</b>	340.10
6	485	1	25.25	68.90	102.85	151.35	228.95	<b>340.50</b>

$z_i$  that are bounded from above by a constant. This is the case of the already noted domain of graph algorithms, in which a graph's numbers  $n$  of vertices and  $m$  of edges are such that  $m = O(n^2)$ . In this section we show that there continue to exist exponents for which  $f(x, y)$  is irreducible even if we constrain  $a_i$  and  $b_i$  to be positive integers (hence  $f(x, y)$  to be a polynomial), and likewise if  $z_i$  is constrained to be no greater than a constant.

**Theorem 2.** *Given any positive integer  $k$ ,  $f(x, y)$  is an irreducible polynomial with  $a_i = a_1(2 - \alpha^{i-1})$  and  $b_i = b_1\beta^{i-1}$ , where  $\alpha = p_\alpha/q_\alpha$  and  $\beta = p_\beta/q_\beta$  are rational constants such that  $0 < \alpha < \beta < 1 - \alpha < 1$ ,  $a_1 = q_\alpha^{k-1}$ , and  $b_1 = q_\beta^{k-1}$ .*

*Proof.* Proceed exactly as in the proof of Theorem 1, then observe that all  $a_i$ 's and  $b_i$ 's are positive integers.  $\square$

A detailed example illustrating Theorem 2 is given in Table 2 and Figure 1 for  $k = 6$ . Table and figure provide different takes on the exact same setting, the former highlighting the integral nature of the exponents in  $f(x, y)$  as well as each  $a_j z_i + b_j$  as a maximum over all  $j$ , the latter highlighting each  $z_i$  and  $r(i, j)$  for  $j > i$ .

**Theorem 3.** *Given any positive integer  $k$  and a constant  $c > 0$ , there exist constants  $\alpha$  and  $\beta$  such that  $0 < \alpha < \beta < 1 - \alpha < 1$  for which  $f(x, y)$  is irreducible with  $z_i < c$ ,  $a_i = a_1(2 - \alpha^{i-1})$ , and  $b_i = b_1\beta^{i-1}$ , where  $b_1/a_1 < c$ .*

*Proof.* Proceed exactly as in the proof of Theorem 1, then impose  $r(k-1, k) < c$  to obtain an upper bound on the allowable values of  $k$ :

$$k < 2 + \log_{\beta/\alpha} \frac{a_1}{b_1} \left( \frac{1 - \alpha}{1 - \beta} \right) c.$$

The result follows from noting that, for  $b_1 < ca_1$ ,

$$\lim_{\beta/\alpha \rightarrow 1^+} \log_{\beta/\alpha} \frac{a_1}{b_1} \left( \frac{1 - \alpha}{1 - \beta} \right) c = \infty.$$

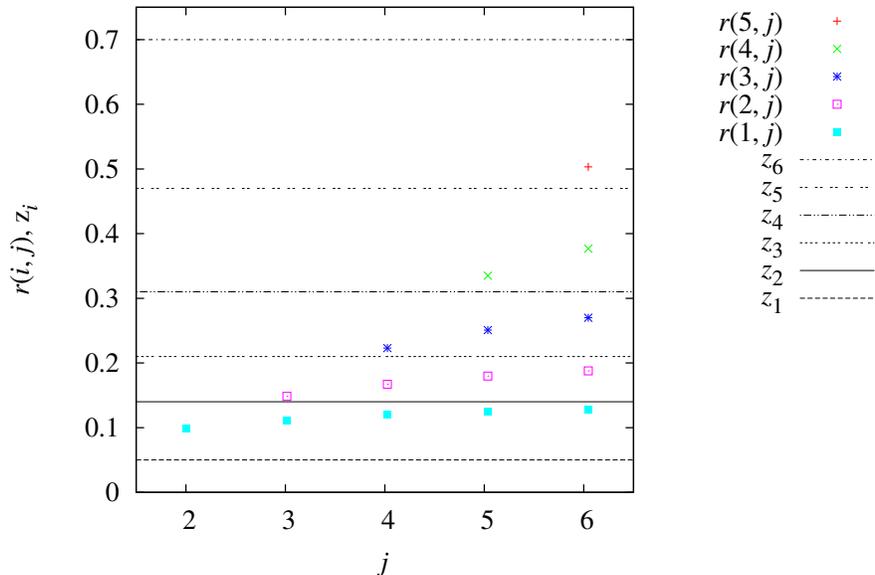


Figure 1:  $r(i, j)$  and  $z_i$  values for  $1 \leq i < j \leq k$ , in the same setting as in Table 2.

Therefore, choosing  $\alpha$  and  $\beta$  to be arbitrarily close to each other accommodates any desired  $k$ .  $\square$

## 5 Concluding remarks

The analysis of computer algorithms via the big-oh notation is an essential part of most activities within computer science, including both theoretical studies and the myriad of applications to which people working in the field devote themselves. In the great majority of situations the algorithm that is being considered is known at some level of detail, so that obtaining big-oh expressions for how much time or space it consumes, though far from being a simple task, is at least a well-defined one. In this article, by contrast, we started out with a “black-box” version of an algorithm, that is, a version that we can only analyze by running it on a given set of inputs to make measurements of how much of the necessary resources the algorithm spends.

Faced with the task of discovering big-oh expressions bounding such resource usage, and limiting our search to polynomial-like functions of the relevant variables, we found that, in principle, an automated procedure to carry out the task might have to consider functions comprising an unbounded number of terms. This is surprising, given all the accumulated knowledge on so many algorithms

to solve so many different problems, but we feel that it sheds additional light on the big-oh notation itself, especially when we consider the subtle pitfalls that sometimes motivate a deeper examination of its use [5].

We close with two final remarks. The first is that our conclusions can be easily extended to the case of more variables, recursively by simply fusing together all current variables through appropriate valuations whenever a new variable is added to the pool. The second remark is that, even though for this work we found motivation in the analysis of computer algorithms, the big-oh notation is in fact of much wider interest and applicability, providing a crucial tool whenever it is “asymptotics,” not exact figures, that matter. This occurs in several other fields within mathematics, as well as in science and engineering.

## Acknowledgments

The authors acknowledge partial support from CNPq, CAPES, FAPERJ, and a FAPERJ BBP grant.

## References

- [1] G. H. Hardy. *Orders of Infinity: The 'Infinitärrechnung' of Paul du Bois-Reymond*. Cambridge University Press, Cambridge, UK, second edition, 1924.
- [2] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, second edition, 1989.
- [3] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume A–C. Springer-Verlag, Berlin, Germany, 2003.
- [4] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
- [5] K. W. Regan. A polynomial growth puzzle, 2015. Gödel’s Lost Letter and P = NP, <https://rjlipton.wordpress.com/2015/09/12/a-polynomial-growth-puzzle/>.