OO Big O: A Sensitive Notation for Software Engineering

Joan Krone, William F. Ogden, and Murali Sitaraman

Technical Report RSRG-03-06
Department of Computer Science
451 Edwards Hall
Clemson University
Clemson, SC 29634-0974 USA

September 2003

OO Big O: A Sensitive Notation for Software Engineering

Joan Krone
Mathematics and Computer Science
Denison University
Granville, OH 43023, USA
+1 740 587 6484
krone@denison.edu

William F. Ogden
Computer and Information Science
The Ohio State University
Columbus, OH 43210, USA
+1 614 292 6004
ogden@cis.ohio-state.edu

Murali Sitaraman
Computer Science
Clemson University
Clemson, SC 29634, USA
+1 864 656 3444
murali@cs.clemson.edu

ABSTRACT

Traditionally the performance of algorithms is expressed in terms of sizes of objects that are parameters to the algorithm. Big-O notation, defined to compare functions over natural numbers, is used for this purpose. However, the running times of interesting classical algorithms vary widely – even in order of magnitude estimation –depending on the "values" of the objects, even if their sizes are the same. For example, it takes different times to search and sort depending on the permutations of the input. When sizes are used as the metric, it becomes impossible to document such variances. To address the problem, we present a new notion of Big O. The new notation permits more sensitivity because it allows the domains of the functions used to document performance to be arbitrary mathematical spaces instead of natural numbers. Where size-oriented metrics are adequate, performance expressions based on the new notation are compatible with classical Big-O expressions. We motivate the new notation use using an insertion sorting algorithm, and explain that is more suitable for teaching object-oriented computing.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *performance measures.*

General Terms – Algorithms, Measurement, Documentation, Performance.

Keywords – Components, efficiency, metrics, objects, reuse, specification.

1. INTRODUCTION

Big O notation is introduced in undergraduate data structures and algorithms courses to document the performance of algorithms, and to distinguish the efficiency of one algorithm from another. The notation is used to express the running times of algorithms in terms of input sizes. When an algorithm is documented to take time O(f(x)) on input x, the definition of Big O requires that the domain of the function is the set of natural numbers.

If the input to an algorithm is a number, then the Big O notation is both appropriate and sufficient. However, in object-oriented computing, objects may be complex container types whose entries are themselves complex types, so that counting the number of entries in the object may give some idea of complexity, but counting hides critical information necessary to provide a realistic performance estimate.

In current object-oriented curricula, to apply the big-O notation, a mapping is established between the abstract set of values an object can take to the domain of natural numbers. Typically, this mapping results in the length or size of an object. For example, to document the running time of a sorting algorithm with a list P as its input, first a mapping function to natural numbers such as, Length: List $\rightarrow$ **N**, is defined. Following the mapping, a function with natural numbers as its domain, such as n^2, is used to document the performance in big-O notation as $O(n^2)$.

This approach has shortcomings. Non-trivial objects are explained typically using a rich set of mathematical models. The classical definition of Big O requires that these arbitrary models be projected down to the natural numbers. This projection entails a significant loss of precision beyond that intrinsic to order of magnitude estimation. In other words, big O becomes lot less expressive than it has the potential to be. Moreover, given that larger objects are composed of smaller objects, the lack of a general method of formulating an appropriate natural number projection for a larger object from the projections for its constituent objects constitutes a barrier to compositional performance analysis.

To illustrate our new approach to algorithm analysis, we show the analysis of an insertion sorting algorithm and present an object-friendly Big O notation, referred to as OO Big O, hereafter in this paper. OO Big O allows use of functions over arbitrary mathematical spaces, and requires no intermediate mapping functions to the natural numbers. It permits documentation of performance estimates that are more precise and more informative, yet it remains compatible with classical Big O notation when size-oriented metrics are sufficient for a problem.

Given the page constraints, this paper serves as a motivating introduction to OO Big O for educators. Several important issues are addressed in other published and pending papers. It is important to know that OO Big O has compositional properties similar to the classical Big O notation. The properties and their proofs are established elsewhere [6]. There, we have defined corresponding notations for OO Ω, OO θ, and OO o, and established their properties as well. OO Big O permits a gradual transition to more exact analysis that is necessary ultimately in predictable, real-time software engineering. While performance specification of reusable components, including both time and space is the focus of our work in [10], establishing exact bounds is the topic of [2].

2. EXAMPLE PERFORMANCE ANALYSIS

We consider an algorithm for sorting a "list" as an illustrative example in this paper, though most other algorithms discussed in data structures courses raise similar issues. The running time complexity of the insertion sort procedure is given usually as O (n^2) where n is the number of elements in the list. Since the expression is based merely on the length of the list, it cannot make any distinction between the sorting times for different lists of the same length. The time for insertion sort is linear on a sorted list, and it increases to quadratic time complexity gradually depending on how unsorted the list is. To address this issue partially, a best case estimate, such as $\Omega(n)$ is used in analyzing the algorithm. While it is useful to know that insertion sorting takes linear time in some cases and quadratic time in some cases, the bounds do not show any correspondence between the particular permutation of entries in a list and the running time of insertion sort. What we need is the ability to express the performance in a way that is sensitive to the particulars of the list, not merely its length.

2.1 Insertion Sort Algorithm

To formalize the analysis of performance and illustrate the issues, we begin with a discussion of a List component. A complete specification is given in the Appendix, and it is discussed in detail in [11]. The component provides a List type and operations to manipulate objects of List type. It is parameterized by the type of entries in a list.

We can view a List abstractly as a pair of strings over the type of entries in the list. The first string contains the list entries preceding the current position and it is named *Prec*; the second string is the remainder of the list, *Rem*. Initially, a List is a pair of empty strings, i.e., (Λ, Λ).

Conceptualizing a List object as a pair of strings makes it easy to explain insertion and removal from the "middle". For example, suppose that (<3, 5>, <4, 1, 9>) is the abstract value of an Integer list. When the Insert operation is called to a add 7, the new element is inserted at the beginning of the *Rem* string of the list and the resulting list becomes (<3, 5>, <7, 4, 1, 9>). A call to the Remove operation reverts the list to (<3, 5>, <4, 1, 9>). Neither of these operations changes the insertion position. To change the position, Advance or Reset needs to be called. Advance moves the list position so that the list (<3, 5>, <4, 1, 9>) becomes (<3, 5, 4>, <1, 9>). Reset sets the insertion point to the beginning by making the list (Λ, <3, 5, 4, 1, 9>). Neither of these operations changes the overall contents of the list.

Since the operations on list position (Insert, Advance, Reset, Remove, etc.) all have easy specifications in terms of the mathematical model, and since the underlying linking pointers in the implementation are cleanly hidden, reasoning about client code is much simplified with this abstract model [8]. Undergraduate students find the model particularly easy to understand and comprehend [11].

Given this explanation of lists, Figure 1 contains the specification of an operation to sort a list. While an informal understanding of the operation is sufficient for the purposes of this paper, we give the formal specification because it helps connect the principles taught in object-oriented and software engineering courses with the principles of data structures and algorithms. The benefits of introducing an abstract client view to undergraduate students before discussing implementation details are documented in [5][11].

In the specification, the **requires** clause, when present, specifies an obligation for the caller. The **ensures** clause is a guarantee from a correct implementation, and it describes how the list *P* is updated. Here, *#P* denotes the incoming value of *P* and *P* denotes the outgoing value. The operation is specified to reset the List, and guarantee that the remaining part of the list is sorted and is a permutation of the entire incoming list which is *#P.Prec* concatenated with *#P.Rem*. The definition of *Is_Ascending_Order*

(or more precisely *Is_Non_Descending_Order*) should be given in terms of the definition used for sorting entries and it is omitted here due to space considerations.

```
Operation Sort_List( updates P: List );
    ensures P.Prec = Λ and In_Ascending_Order( P.Rem ) and
        P.Rem Is_Permutation #P.Prec o #P.Rem;
```

Figure 1: Specification of a List Sort Operation

Figure 2 contains a procedure to implement the list sorting operation. It uses a local list *Sorted* for the purpose.

```
Procedure Sort_List( updates P: List );
    Var P_Entry, S_Entry: Entry;
    Var Sorted: List;
  Reset ( P );
 While Length_of_Rem( P ) ≠ 0 do
        Remove( P_Entry, P );
        Iterate
                When Length_of_Rem( Sorted ) = 0
                        do exit;
                Remove( S_Entry, Sorted );
                When Lss_or_Comp( P_Entry, S_Entry )
                        do Insert( S_Entry, Sorted ) exit;
                Insert( S_Entry, Sorted );
                Advance( Sorted );
                repeat;
        Insert( P_Entry, Sorted );
        Reset( Sorted );
   end;
   P :=: Sorted;
end Sort_List;
```

Figure 2: Insertion Sort Procedure

Both the inner and outer loops maintain the invariant of keeping the elements in the *Sorted* list sorted. We have omitted a formal statement of the invariant in the figure because it is not necessary for the present discussion, though it is useful to present it at least at an informal level [3]. The strategy of the outer loop is to place successive values *P_Entry* from the list *P* into their proper place in the list *Sorted*. The **Iterate**…**repeat** construct is a "for ever" loop that is terminated when one of the exit conditions is satisfied. The task of the inner loop is to position the *Sorted* list appropriately for the insertion of *P_Entry*. In the inner loop, *Lss_or_Comp* is a generic procedure to compare two entries.

In the last statement of the procedure, a swap statement, denoted by :=:, is used to transfer the result from the *Sorted* list back to the parameter *P*, the list that is to be sorted. Even when pointers are avoided in a presentation of lists, assignment of references re-introduces reasoning complexity. For this reason, we do not define or use list assignment, using instead the swap operator. Swap operator can be implemented efficiently by exchanging references, without deep copying and without introducing aliasing [2].

2.2 Analysis

To analyze the efficiency of the insertion sorting algorithm, we examine first the inner loop and then the outer loop to get a duration estimate for each. Then we put those estimates together to get the duration clause to associate with the procedure. As is the case in the classical performance analysis, we assume that all List operations and the *Lss_Or_Cmp* operation to compare entries take a constant time, though this assumption is not necessary when using the revised Big O notation. Based on the assumption, in Big O analysis we note that the outer loop may execute at most n times (where n is the length of the input list). The inner loop might execute a maximum of n times if the next entry that is to be inserted into the sorted list needs to be compared with all the previously inserted entries into that list. This leads us to an overall worst-case complexity of $O(n^2)$ for the insertion sort procedure.

For a more precise analysis, note that whenever the inner loop is entered, the preceding string of the Sorted list is empty because it is reset in the outer loop. The inner loop compares each entry *S_Entry* in the remaining string of the sorted list with the entry *P_Entry*. After comparison, the entry is inserted back into the list and the list is advanced. The time for the inner loop depends upon the number of entries to be compared with the next item before finding the correct place for insertion, in particular, the number of entries in Sorted list that are less than or comparable to *P_Entry*. To get the increased precision, we need to define a function on strings of entries α that counts how many entries in α are less than an entry *E* and hence would be "advanced" over when positioning *E* after α has been sorted[1]. For this reason, we define a *Rank(E, α)* function and states some of its properties.

In the definition "$\circ$" denotes concatenation and $\prec$ denotes a generic notion of "less than" on type Entry. Given this definition, it is easy to see that the duration for the inner loop is proportional to *Rank(P_Entry, Sorted.Rem)* at the beginning of the loop (or *Rank(P_Entry, Sorted.Prec)* at the end of the loop). Rank counts the number of calls to Advance that are necessary to position the list properly.

Inductive definition on α: Str(Entry) **of**

Rank(E: Entry, α): $\mathbb{N}$ **is**

(i) Rank(E, Λ) = 0;

(ii) Rank(E, $\alpha \circ$ <D>) = $\begin{cases} \text{Rank}(E, \alpha) + 1 & \text{if } D \prec E \\ \text{Rank}(E, \alpha) & \text{otherwise} \end{cases}$;

The outer loop inserts the next entry into the *Sorted* list. Since the time of the outer loop depends upon the cumulative effect of positioning successive entries from the list *P* into the *Sorted* list, we define a "preceding rank" function *P_Rank(α)*. For an example, suppose that we are sorting a list with the

[1] We have chosen to count the number of calls to Advance operation at the end of the inner loop rather than the number of calls to the comparison operation. We could have defined it to count the comparisons. In order of magnitude analysis, however, this does not make any difference.

abstract value (Λ, <3, 5, 4, 1, 9>) in the ascending order. The entry 3 is inserted into the empty Sorted list followed by the entry 5 in the Sorted list containing entry 3. The time to insert the next entry 4 depends upon the "rank of 4" in the string preceding "4" which is the string <3, 5>. To compute the cumulative time, we find out for each entry the time it takes to insert it into the preceding list.

Inductive def. **on** α: Str(Entry) **of** P_Rank(α): ℕ **is**

(i) P_Rank(Λ) = 0;

(ii) P_Rank(α ∘ <E>) = P_Rank(α) + Rank(E, α);

Using the formula *P_Rank*, we can see that, it takes a lot less time to sort the list (Λ, <9, 5, 4, 3, 1>) in ascending order than the list (Λ, <1, 3, 4, 5, 9>), though the two lists have the same 5 entries. In the first case, no advances are necessary. In the second case, 10 advances are necessary.

3. OO BIG O DEFINITION

The traditional Big O is a relation between natural number based functions defined in the following way [1]: Given f, g: $\mathbb{N}\rightarrow\mathbb{R}$, f(n) is O(g(n)) iff ∃ positive constants c and n_0 such that $f(n) \le c \cdot g(n)$ whenever $n \ge n_0$.

A program whose running time is f(n) is said to have growth rate O(g(n)).. As we have explained, when the object classes central to modern programming are formally modeled, they present to clients a view of objects that could come from essentially arbitrary mathematical domains, since the point of introducing objects is to simplify reasoning about the functionality of components by providing a minimalist explanation that hides the details of their implementation. But the functionally simplest models may have little to do with natural numbers. So the natural expression of the duration f(x) of an operation on object x is as a function directly from the input domain used to model x to the real numbers. Clearly, any gauging function g that we might want to use as an estimate for f should have this same domain. Accordingly, the relation between functions f(x) and g(x)) is defined by:

Definition: Given f, g: Dom $\rightarrow\mathbb{R}$, f(x) is O(g(x)) iff (∃ A: $\mathbb{R}^{>0}$, ∃ H: ℝ ∍ ∀ x: Dom, $f(x) \le A \cdot g(x) + H$).

In other words, for two timing functions f and g mapping a computational domain Dom to the real numbers, to say that f(x) is O(g(x)) is to say that there is some positive acceleration factor A and some handicap H such that for every domain value x, $f(x) \le A \cdot g(x) + H$. If we think of f and g as representing competing processes, f being big O of g means that f is not essentially faster than g. If g is run on a processor A times faster than f's processor and also given a head start H, then g will beat f on all input data x.

Using the new notation, we can give a more precise estimate of the Sort_List procedure as:

O(Max(|#P.Prec ∘ #P.Rem|, P_Rank(#P.Prec∘#P.Rem)).

The maximum is necessary is this expression, because even when no advances are necessary, the insertion sort procedure given in Figure 2 takes at least linear time because every entry is moved from the initial list to the sorted list in the process. Using the new estimate, it is clear that insertion sorting algorithm takes different times on lists with abstract values (Λ, <9, 5, 4, 3, 1>) and (Λ, <1, 3, 4, 5, 9>).

An important theorem about *P_Rank* is that $P\_Rank(\alpha) \le |\alpha| \cdot (|\alpha| - 1)/2$, so it follows that the duration of the *Sort_List* procedure is O($|\#P.Prec \circ \#P.Rem|^2$) too. Thus we can get the much less exacting estimate produced by traditional Big O analysis if we wish. We are just not forced to when we need a sharper estimate. Another point to note is that besides being compatible with correctness proofs for components,

the direct style of performance specification is much more natural than using an intermediary natural number.

Of course, in order to use this definition, it is necessary to have mathematical support in the form of theorems about the revised definition of Is_O. For example, we need an additive property so we can apply our analysis to a succession of operation invocations:

Theorem OM1: If $f_1(x)$ is (O $g_1(x)$) and $f_2(x)$ is O($g_2(x)$), then

$f_1(x) + f_2(x)$ is O(Max($g_1(x)$, $g_2(x)$)).

A development of appropriate theorems and definitions appears elsewhere [6].

4. DISCUSSION

Traditional Big O order of magnitude estimates are inadequate for software engineering because they deal only with the domain of natural numbers. To use them for non-trivial objects, it becomes necessary to artificial mapping functions from the value space of the objects to the set of natural numbers, and this mapping in turn leads to a loss of information necessary for precise estimation. To address these shortcomings, we have introduced a new notation that scales up for doing order of magnitude analysis to the context of generic, abstract data types.

There are key benefits for introducing the new notation in undergraduate Computer Science education. The new notation helps connect principles taught in object-based software engineering courses with techniques for efficiency analysis in data structures courses. The students will see that the notation scales up to performance estimation on arbitrary objects, including generic ones. They will realize that Big O notation is a useful software engineering tool and that it can be used for expression of performance estimates at various levels of precision depending on the needs of the situation.

The generality of the new notation is especially useful in the context of reusable components, where the context of use is not known in advance, and hence, performance needs to be expressed making few assumptions about component usage. Performance specifications are necessary for components to reason about the performance of component-based systems in a modular fashion. To avoid the rapid compounding of imprecision that otherwise happens in such systems, it is also essential to use high precision performance specification mechanisms, such as the one presented here.

Probably the most significant benefit from an educational perspective is that the new notation permits a better understanding of the underlying algorithms. For example, understanding the performance estimate for insertion sorting given in this paper clarifies for a student how it is intrinsically different from a bubble sorting algorithm, unlike more classical analysis where both algorithms have $O(n^2)$ worst case complexity. From an educator's perspective, the compatibility of the new notation with the traditional one makes it possible to introduce it as a natural extension, though we believe that it is just as easy to start with the new notation. The few undergraduate students who have used the new notation did so with ease and found it to be more meaningful in understanding algorithms. However, this limited experience is not sufficient to allow us reach any formal conclusions.

5. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grants CCR-0081596 and CCR-0113181. We would like to thank members of the *Reusable Software Research Groups* at Clemson University and The Ohio State University for their inputs on the topics discussed in this paper.

6. REFERENCES

[1] Aho, A., Hopcroft, J., Ullman, J., *Data Structures and Algorithms*, Addison-Wesley, 1983.

[2] Harms, D.E. and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components", *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, 1991, 424-435.

[3] Henderson, P. B., "Mathematical Reasoning in Software Engineering Education," *Communications of the ACM*, Vol. 46, No. 9, 2003, 45-50.

[4] Krone, J., Ogden, W. F., and Sitaraman, M., *Modular Verification of Performance Constraints*, Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003, 25 pages.

[5] Long, T.J., Weide, B. W., Bucci, P., and Sitaraman, M., "Client-View First: An Exodus from Implementation-Biased Teaching", *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, ACM , 1999, 136-140.

[6] Ogden, W. F., *CIS680 Course notes*, Spring 2002.

[7] "Special Feature: Component-Based Software Using RESOLVE," *ACM SIGSOFT Software Engineering Notes 19*, No. 4, Eds. M. Sitaraman and B. W. Weide, October 1994, 21-67.

[8] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software-Component Behavior," in Frakes, W.B., ed., *Software Reuse: Advances in Software Reusability (Proceedings Sixth International Conference on Software Reuse),* Springer-Verlag LNCS 1844, 2000, 266-283.

[9] Sitaraman, M., Weide, B. W., Long, T.J., Ogden, W. F., "A Data Abstraction Alternative to Data Structure/Algorithm Modularization," *LNCS 1766 Volume on Generic Programming*, Eds. D. Musser and M. Jazayeri, Springer-Verlag, 2000, 102-113.

[10] Sitaraman, M., Krone, J., Kulczycki, G., Ogden, W., and Reddy, A. L. N., "Performance Specification of Software Components," *ACM SIGSOFT Symposium on Software Reuse*, May 2001.

[11] Sitaraman, M., Long, T.J., Weide, B. W., Harner, E. J., and Wang, L., "Teaching Component-Based Software Engineering: A Formal Approach and Its Evaluation," *Computer Science Education*, Vol. 12, Nos. 1 – 2, Swets & Zeitlinger, March 2002, 11-36.

7. APPENDIX: LIST COMPONENT SPECIFICATION

The appendix shows the specification of a *List* component in RESOLVE notation [7]. *List_Template* is a generic **concept** (specification template) and it is parameterized by the type of entries to be contained in lists. To provide abstract mathematical explanations of the operations, an object of type *List* **is modeled by** an *ordered pair of mathematical strings* of entries.

We have used specifications such as these routinely in freshmen undergraduate courses on data structures [5][11]. Given this specification, students act as clients and use lists in problem solving within the first few weeks of their second quarter/second semester course. They use a specification-based "natural" or forward reasoning method to reason about correctness [8]. Only later they learn how to implement lists using pointer structures. In addition to classical examples such as Lists, students also see data abstractions that result from recasting classical algorithms as objects [11].

RESOLVE specifications use a combination of standard mathematical models such as integers, sets, functions, and relations, in addition to tuples and strings. The explicit introduction of mathematical models allows use of standard notations associated with those models in explaining the operations. Our experience is that this notation—which is precise and formal—is nonetheless fairly easy to learn to

understand even for beginning computer science students, because they have seen most of it before in high school and earlier.

```
Concept List_Template (type Entry)
   Type List is modeled by
        (Prec: string of Entry,
         Rem: string of Entry)
     exemplar P
     initialization ensures
      |P.Prec| = 0 and |P.Rem| = 0;

    Operation Insert (
        alters E: Entry
        updates P: List
      );
      ensures P.Prec = #P.Prec and
           P.Rem = <#E> • #P.Rem;

    Operation Remove (
        replaces R: Entry; updates  P: List
      );
     requires |P.Rem| > 0;
      ensures P.Prec = #P.Prec and
           #P.Rem = <R> • P.Rem;

    Operation Advance (
        updates P: List
      );
     requires |P.Rem| > 0;
     ensures P.Prec • P.Rem =
              #P.Prec • #P.Rem and
           |P.Prec| = |#P.Prec| + 1;

    Operation Reset (
        updates P: List
      );
     ensures |P.Prec| = 0 and
         P.Rem = #P.Prec • #P.Rem;

    Operation Advance_To_End (
        updates P: List
      );
     ensures |P.Rem| = 0 and
           P.Prec = #P.Prec • #P.Rem;
```

```
   Operation Prec_Length (
       restores P: List
     ): Integer;
    ensures Prec_Length = |P.Prec|;

   Operation Rem_Length (
       restores P: List
     ): Integer;
    ensures Rem_Length = |P.Rem|;
end List_Template;
```