Time Complexity Analysis and Big *O* Notation in CSE205

Why do I need to learn this stuff and what does the instructor really expect me to know?

Asymptotic time complexity analysis and Big O notation are fundamental topics within the field of computer science, primarily in regards to the study of algorithms. I know that many of the students in this course are not CS majors and they may wonder why we cover it at all. Well, algorithms have practical applications to programming, and being able to devise your own algorithm to solve a programming problem is fundamental to being a programmer (as well as being able to comprehend and use existing algorithms). It is not good, however, to design an algorithm which may be so inefficient that it is practically useless, so understanding algorithmic efficiency is a key part of your education. I teach you the formal definition, and the method we use to formally analyze an algorithm to determine its time complexity, because I want to make you a better programmer, one who stands out from the rest of the pack who do not know anything about time complexity or Big O.

I understand and know that many of you find the topic confusing. It *is* a difficult concept and when I was a student, I struggled with it as well. No one—short of evil geniuses—truly gets it the first time. It takes a lot of study to really comprehend this concept and to be able to apply it. Because of that, you will see that time complexity and Big O are typically covered in many different computer science and software engineering courses, e.g., at ASU in our Computer Science and Computer Systems Engineering programs, students study this topic in CSE205, Discrete Math (MAT243), Algorithms & Data Structures (CSE310), and again at the graduate level if a student pursues that option. In the Software Engineering program, it is also studied again in Design & Analysis of Data Structures and Algorithms (SER222), and I believe SwEng majors also take MAT243 so you will also see it there.

Now, the answer to your question of what does the instructor really expect me to know? First, I cover the material in a fair amount of detail starting with the formal definition of Big O, teaching you the proper way to analyze an algorithm, because I believe it is important when learning a new topic to always start with the fundamental concepts and the definition of Big O is as fundamental as it gets. So I discuss and show you how to use math to perform a semi-formal proof[1].

However, if one focuses just on the fundamentals and small details, then it is easy to lose sight of the big picture. And the big picture is what I want you to leave this course understanding. I am not going to put a question on the exam which asks you to use math to (in)formally prove the time complexity of an algorithm. I will put a few of these questions on the homework assignments and I apologize for doing so, but I have to. Do the best you can at solving them, but in the end, if you walk away from CSE205 still scratching your head wondering about time complexity and Big O notation, that is understandable. Remember that you came into CSE205 knowing *nothing* about time complexity or Big O notation (if not, pretend so) and you are leaving with *some* knowledge of the concept. Despite some possible confusion and less than complete understanding, by completing this course, you have expanded your knowledge, and that is what education is about[2].

Okay, here is the big picture list of what I want you to know about this subject when you finish CSE205:

- Understand what we mean when we talk about the *asymptotic time complexity* of an algorithm. Time complexity is a theoretical estimation of the time it will take an algorithm to solve a problem, given the problem size. We call it *asymptotic* time complexity because we allow the size of the problem to grow to infinity when we determine the time complexity.
- Know what we mean by the *worst case time complexity* – the algorithm will never perform worse than this, regardless of the size of the problem.
- Know what we mean by *best case time complexity* – the algorithm will never perform better than this, regardless of the problem size.

1 I would be kicked out of the math club with my sloppy proofs :) Proofs are why I went to grad school in CS rather than math.

2 It should not be about grades. If it were up to me, there would be no grades, but the administration makes the rules and they love grades.

- Know that $O(1)$, $O(n)$, $O(n\ lg\ n)$, and so on are *complexity classes*[3]. These classes characterize the common time complexities of different algorithms, i.e, some algorithms are in $O(1)$, some are in $O(n)$, and others are in $O(2^n)$.
- Understand the ranking of the most common complexity classes, e.g., $O(n\ lg\ n)$ is the complexity class consisting of all algorithms whose asymptotic time complexity is proportional to $n\ lg\ n$, and classifies algorithms which are faster than the members of the $O(n^2)$ complexity class, which itself classifies faster algorithms than the $O(2^n)$ complexity class.
- Know that we analyze algorithms using Big O notation rather than by measuring the running time on a computer because the time it takes to run the algorithm on a computer will vary from system to system. On the other hand, the fundamental time complexity of an algorithm is entirely independent of the speed of a system so this permits us to compare algorithms (and not computers) for speed. (This is not to say that comparing algorithms by physical timing measurements is not important nor useful, but that is just not something we discuss in CSE205.)
- Understand that the *key operation* of an algorithm is the one that is performed the most number of times *and* the key operation is often most fundamental to, or at the heart of, the algorithm. For example, in a sorting algorithm such as bubble sort which compares adjacent elements and swaps them when they are out of order, the operation that is performed the most number of times is the $>$ or $<$ comparison operation and note that the comparison operation is at the heart of any sorting algorithm which sorts by comparing elements.
- Know how to informally prove the time complexity of a simple algorithm—such as linear search—by identifying the key operation, being able to count or estimate how many times the key operation is performed as a function of the problem size, i.e., find $f(n)$, and how to choose the appropriate complexity class based on $f(n)$.
- Know that $O(1)$ is the constant complexity class and for any problem where the key operation is performed a constant number of times, the algorithm is $O(1)$ even if the constant number of times is $f(n) = 10$ million. That is, there is no such thing as $O(2)$, $O(20)$, or $O(123{,}456{,}789)$. In all cases, we would say the algorithm is $O(1)$.
- Know that the time complexity of linear search is $O(n)$ and binary search is $O(lg\ n)$
- Know that a $O(lg\ n)$ searching algorithm is considered very fast and very good, so binary search is a reasonably fast algorithm.
- [We discuss sorting algorithms in Module 5] Remember that bubble sort, insertion sort, selection sort are $O(n^2)$ algorithms and quick sort and merge sort are $O(n\ lg\ n)$.
- [We discuss sorting algorithms in Module 5] There is no in-memory sorting algorithm—i.e., where the entire list is stored in memory while sorting—which sorts by comparing list elements, e.g, is $list_i > list_{i+1}$, that is faster than $O(n\ lg\ n)$. Quick sort and merge sort, both being $O(n\ lg\ n)$, are two of the fastest sorting algorithms (among those sorting algorithms which sort based on comparisons). Just so you know, there *are* sorting algorithms which are faster than $O(n\ lg\ n)$ but these are not in-memory sorting algorithms which sort by comparing elements the way selection sort, insertion sort, bubble sort, merge sort, and quick sort do.
- [This subject is discussed in Module 6] Remember the best and worst case time complexities for: (1) adding an element to a list, stack, or queue; (2) for deleting an element from a list, stack, or queue; and (3) searching for an element in a list, stack, or queue.
- [We discuss this topic in Module 7] Know that the time to perform add, delete, and search operations on a well-balanced binary search tree is $O(lg\ n)$.

I do not expect you to be able to formally prove, from scratch, the time complexities of more complicated algorithms such a binary search, merge sort, or quick sort, and there will be no proofs on the exams.

3 Big O is *not* a function, i.e., $O(n^2)$ is not a mathematical function named O which has a parameter n^2. Repeat: $O(n^2)$ is a complexity class, i.e., the members of the class are all algorithms which have asymptotic time complexity which is proportional to n^2.

Time Complexity References for Beginners

Here are a few references to some beginner's guides to time complexity and Big O notation. The first six are brief with the last two being longer and covering the material in more depth. I did not search for video tutorials but if anyone has a link they wish to share, please send it to me and I will incorporate it in this list. I **strongly recommend** you read the discussion at the first six sites. This will help you better understand the subject.

1. A Beginners Guide to Big O Notation, freeCodeCamp
   http://medium.freecodecamp.org/my-first-foray-into-technology-c5b6e83fe8f1

2. A Beginner's Guide to Bog O Notation, Rob Bell
   http://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

3. Algorithms for dummies (Part 1): Big-O Notation and Sorting
   http://adrianmejia.com/blog/2014/02/13/algorithms-for-dummies-part-1-sorting

4. Big-O Notation for beginners, Changmin Lin
   http://medium.com/@changminlim/big-o-notation-for-beginners-ae17e8f70414

5. Learning Big O Notation With O(n) Complexity, DZone
   http://dzone.com/articles/learning-big-o-notation-with-on-complexity

6. Practical Java Examples of the Big O Notation, OrryMesser
   http://www.baeldung.com/java-algorithm-complexity

7. Asymptotic Notation, Khan Academy
   http://www.khanacademy.org/computing/computer-science/algorithms#asymptotic-notation.
   There are seven sections. Skip over Big Theta (Θ), Big Omega (Ω) notation, and the last practice question.

8. A Gentle Introduction to Algorithm Complexity Analysis
   http://discrete.gr/complexity.
   This article is more technical and in-depth, and covers some topics that we do not teach in CSE205 (ignore anything involving Big Θ and Big Ω). I suggest reading the sections: Motivation; Counting instructions; Worst-case analysis; Asymptotic behavior; Complexity; Big-O notation (skip over the exercises); Logarithms; Recursive complexity; and Logarithmic complexity.