

CS50 Computational Complexity

Overview

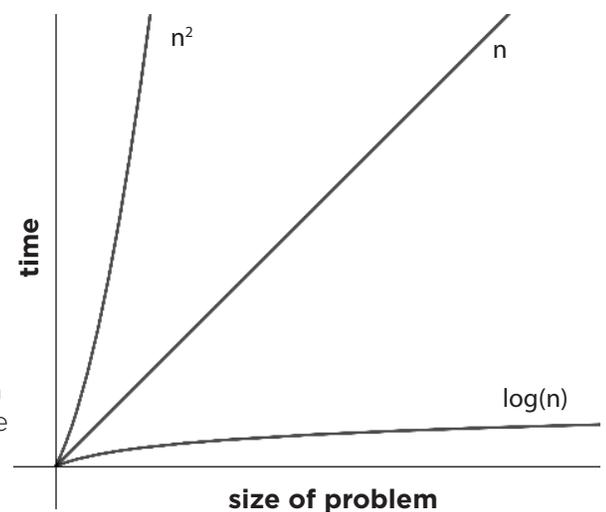
The subject of computational complexity (also known as time complexity and/or space complexity) is one of the most math-heavy topics, but also perhaps one of the most fundamentally important in the real-world. As we begin to write programs that process larger and larger sets of data, analyzing those data sets systematically, it becomes increasingly important to understand exactly what effect those algorithms have in terms of taxing our computers. How much time do they take to process? How much RAM do they consume? One aspect of **computational complexity** is the amount of time an algorithm takes to run, in particular considering the theoretical worst case and best case scenarios when running programs.

Key Terms

- computational complexity
- Big O
- Big Ω

Computational Complexity Notation

Big O notation, shorthand for "on the order of", is used to denote the worst case efficiency of algorithms. Big O notation takes the leading term of an algorithm's expression for a worst case scenario (in terms of n) without the coefficient. For example, for linear search of an array of size n , the worst case is that the desired element is at the end of the list, taking n steps to get there. Using Big O notation, we'd say linear search runs in $O(n)$ time. We can calculate the computational complexity of bubble sort in the same way, albeit with a little more math. Remember that bubble sort involved comparing things by pairs. In a list of length n , $n - 1$ pairs were compared. For example, if we have an array of size 6, we would have to compare `array[0]` and `array[1]`, then `array[1]` and `array[2]`, and so on until `array[4]` and `array[5]`. That's 5 pairs for an array of size 6. Bubble sort ensures that after k passthroughs of the array, the last k elements will be in the correct location. So in the first passthrough there are $n-1$ pairs to compare, then on the next passthrough only $n-2$ comparisons and so forth until there is only 1 pair to be compared. In math $(n-1) + (n-2) + \dots + 1$ can be simplified to $n(n-1)/2$ which can be simplified even further to $n^2/2 - n/2$. Looking at the expression $n^2/2 - n/2$, the leading term would be $n^2/2$, which is the same as $(1/2)n^2$. Getting rid of the coefficient, we are left with n^2 . Therefore, in the worst case scenario, bubble sort is on the order of n^2 , which can be expressed as $O(n^2)$. Similar to big O, we have **big Ω** (omega) notation. Big Ω refers to the best case scenario. In linear search, the best case would be that the desired element is the first in the array. Because the time needed to find the element does not depend on the size of the array, we can say the operation happens in constant time. In other words, linear search is $\Omega(1)$. In bubble sort, the best case scenario is an already sorted array. Since bubble sort only knows that a list is sorted if no swaps are made, this would still require $n-1$ comparisons. Again, since we only use the leading term without the coefficients, we would say bubble sort is $\Omega(n)$.



Comparing Algorithms

Big O and big Ω can be thought of as upper and lower bounds, respectively, on the run time of any given algorithm. It is now clear to see which algorithms might be better to use given a certain situation. For instance, if a list is sorted or nearly sorted, it would not make sense to implement a selection sort algorithm since in the best case, it is still on the order of n^2 , which is same as it's worst case run time. Binary search may seem to be the fastest search but it is clear to see that searching a list once with linear search is more efficient for a one time search, since binary search run requires a sort algorithm first, so it could take $O(\log(n)) + O(n^2)$ to search a list using binary if the list is not already sorted.

Algorithm	Big O	Big Ω
linear search	$O(n)$	$\Omega(1)$
binary search	$O(\log(n))$	$\Omega(1)$
bubble sort	$O(n^2)$	$\Omega(n)$
insertion sort	$O(n^2)$	$\Omega(n)$
selection sort	$O(n^2)$	$\Omega(n^2)$