CS50 Time 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? **Time 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

- time complexity
- Big O notation

Time Complexity Notation

Big O notation is the formal notation in computer science for "on the order of". Big O notation takes the leading term of the expression used to calculate time, in terms of the size, n, of the array without the coefficient. Linear search is relatively simple, if the worst case is that the element being searched for is at the end of the list, then it would take n steps to get there. The notation for this would be O(n). We can calculate 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 I have the numbers listed below, I 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) + ... + 1` can be simplified to `n(n-1)/2` which can be simplified even further to $n^2/2 - n/2$. As mentioned above, big O notation only uses the leading term without the coefficient (assuming the expression from left to right is in descending order in terms of exponents). So if we look at $n^2/2 - n/2$, the leading term would be $n^2/2$, this is essentially $(1/2)\ n^2$. So getting rid of the coefficient we are left with n^2. That is to say that 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. In linear search, the best case would be the element we are searching is the first in the array, so we could write that as Ω(1), since that is a constant(not dependent on the size of the array). In bubble sort, the best case scenario (an already sorted array), would still require `n-1` comparisons, since bubble sort only knows that a list is sorted if no swaps are made. Again since we only use the leading term without the coefficients, this can be expressed as Ω(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)	Ω(1)
binary search	O(log(n))	Ω(1)
bubble sort	$O(n^2)$	Ω(n)
insertion sort	$O(n^2)$	Ω(n)
selection sort	$O(n^2)$	$\Omega(n^2)$