

CSC148 - Asymptotic Notation

Sean Henderson

October 20, 2009

Introduction

We would like to find a way of expressing how "good" an algorithm is with respect to how much space it uses (space) and how fast it is (time). We will primarily focus on time, although the concepts can easily applied to measure memory usage as well.

Consider the following python code which searches for value in a list.

```
def search(needle, haystack):  
    """Search for the value needle in the list haystack, return  
    the index at which it was found, or -1 if it is not in the  
    list."""  
    for i in range(len(haystack)):  
        if haystack[i] == needle:  
            return i  
    return -1
```

Asking "how fast is this program?" is a very open ended question. To give an actual value, measured in standard time units like seconds is not consistent across different computers, etc. A better metric would be to count how many instructions (e.g. lines of code the program executes). To use this, we must define what exactly we mean when we refer to an instruction, and then find a mathematical expression for how many of those instructions the program will execute. Ultimately, even this will prove to be too tedious, and we will really only be interested in the asymptotic performance of this function (how fast it grows as a function of the size of the input).

Counting Instructions

Continuing with our example, how many instructions does search perform? That depends entirely on how long haystack is, and where in it needle occurs (if it does at all).

Let us define X to be the set of all possible inputs to search, and X_n be the set of all inputs to search with $\text{len}(\text{haystack}) = n$. For simplicity, we will restrict X_n such that haystack is some permutation of the integers in $[1, n]$, and needle is chosen from $[0, n]$. Then we can define $X = \cup_{i=0}^{\infty} X_i$.

Let $t(x)$ be the number of instructions search will perform on input $x \in X$. For example, say $x = ([1, 2, 3, 4, 5], 4)$. Then the for loop will iterate 4 times before hitting $i = 3$, and then returning that value. Each iteration of the loop performs 2 instructions (the for loop increment/initialization and the if statement) plus one additional instruction for the return value for a total of $2(4) + 1 = 9$; so $t(x) = 9$.

Let $T(n)$ be the worst case running time for an input of size n . Formally, $T(n) = \max_{x \in X_n} t(x)$. For our search method, on an input of size n , the $t(x)$ is maximized when needle is 0 (i.e. needle is not in the list). In such a case the for loop will execute n times, each time performing 2 instructions, before finally returning using one more instruction. So the worst case runtime $T(n) = 2n + 1$. We would like to express that $T(n)$ is a linear function of n , without having to worry about the constants involved.

Upper Bound

To start with, we are interested in simply showing some upper bound on how fast $T(n)$ grows with respect to n .

For any two functions $f(n)$ and $g(n)$, we say that $f(n) \in O(g(n))$ if and only if there exists constants $n_0 > 0$ and $c > 0$ such that $n > n_0 \Rightarrow f(n) < c \cdot g(n)$. Informally, this says that there is a point after which $f(n)$ is bounded above by a constant times $g(n)$. We can also say that $f(n)$ does not grow faster (with respect to n) than $g(n)$.

Continuing with our example, we established that $T(n) = 2n + 1$. Let us prove that $T(n) \in O(n)$. Choose $n_0 = 5$ and $c = 3$. Clearly, for all $n > 5$, $2n + 1 < 3n$. Therefore, $T(n) \in O(n)$.

Generally, let $f(n) = an + b$ for any b and positive a . We can prove

$f(n) \in O(n)$. Choose $n_0 = b$ and $c = a + 1$. Then

$$n > b \Rightarrow (a + 1 - a)n > b \Rightarrow (a + 1)n > an + b$$

as desired.

In general, for any polynomial $f(n) = \sum_{i=0}^k a_i n^i$ with $a_k > 0$ we can show that $f(n) \in O(n^k)$. (Exercise: Try and prove this. Try $c = a_k + 1$, and take n_0 such that for $n > n_0$, $n^k > \sum_{i=0}^{k-1} a_i n^i$)

Just finding an upper bound on $T(n)$ doesn't necessarily tell you a lot about how fast $T(n)$ grows. Let's continue with our example where $T(n) = 2n + 1$. We can easily show that $T(n) \in O(n^2)$, or that $T(n) \in O(n!)$, but these are not very descriptive.

Lower Bound

Similar to $O()$, we can define a notation to say that $T(n)$ grows at least as fast as some other function.

For any two functions $f(n)$ and $g(n)$, we say that $f(n) \in \Omega(g(n))$ if and only if there exists constants $n_0 > 0$ and $c > 0$ such that $n > n_0 \Rightarrow f(n) > c \cdot g(n)$. Informally, this says that there is a point after which $f(n)$ is bounded below by a constant times $g(n)$. We can also say that $f(n)$ grows at least as fast (with respect to n) as $g(n)$.

Continuing with our example, let us show that $T(n) \in \Omega(n)$. Choose $n_0 = 1$, and $c = 1$. Clearly for all $n > 1$, $2n + 1 > n$.

The same result using polynomials above holds; we can show for any polynomial $f(n) = \sum_{i=0}^k a_i n^i$ with $a_k > 0$ that that $f(n) \in O(n^k)$.

Also similarly, showing just a lower bound does not necessarily tell a lot about the function. Note that $T(n) \in \Omega(\log n)$, and $T(n) \in \Omega(1)$.

Upper and Lower Bound

Ideally, we would like to find a function $f(n)$ such that $T(n)$ grows as fast as $f(n)$; that is $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$. If both of these conditions hold, we say that $T(n) \in \Theta(f(n))$.

Combining results we have stated above, we can say that for any polynomial $f(n) = \sum_{i=0}^k a_i n^i$ (with $a_k > 0$) that $f(n) \in \Theta(n^k)$.

Going back to our search example, we have shown that $T(n) \in O(n)$ and $T(n) \in \Omega(n)$, and we can therefore conclude (by definition) that $T(n) \in \Theta(n)$.

Let's look at a new program, and go through all the steps to calculate its runtime.

```
def my_sort(my_list):
    """Take as input a list, and sorts it in ascending
    order."""
    for i in range(len(my_list)):
        for j in range(i, len(my_list)):
            if my_list[i] > my_list[j]:
                # swap(my_list[i],my_list[j])
                temp = my_list[i]
                my_list[i] = my_list[j]
                my_list[j] = temp
```

We define X_n in this case to be the set of permutations of the integers in the range $[1, n]$. We can easily place an upper bound on $T(n)$ by noting that each of the for loops iterates at most from 0 to $n - 1$, and within the second for loop, at most 4 lines are executed. As such we can conclude that $T(n) \leq 4n^2$. Therefore $T(n) \in O(n^2)$, using results from above.

Next, we need to show a lower bound on $T(n)$. To do this, we present a particular $x_n \in X_n$ in which `my_sort` performs poorly. If $x_n = [1, 2, \dots, n]$ (already sorted) then the outer loop will iterate n times, and the inner loop will iterate n times, then $n - 1$ times, etc., each time performing only 1 operation. So in this case, $t(x_n) = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$. Therefore, we have $T(n) \geq t(x_n) = \frac{1}{2}n^2 + \frac{1}{2}n$ and we have $T(n) \in \Omega(n^2)$.

Therefore, $T(n) \in \Theta(n^2)$, and we can say that our algorithm has quadratic runtime.

Note that there were two steps; first, finding an upper bound based on the maximum number of times the loops can iterate, and then showing a lower bound by presenting a particular case on which the algorithm performs within a constant factor of the runtime.