Computational Complexity

S. V. N. Vishwanathan, Pinar Yanardag

January 29, 2013

1 Introduction to computational complexity: What, how and why?

An algorithm is a well defined computational procedure that takes some value, or set of values, as input and produces some values or set of values as output. We can think of an algorithm as a recipe for solving a particular problem such as matrix multiplication, sorting, etc.

It is important to understand the asymptotic performance of an algorithm in order to compare their relative performance. By analysing algorithms, we determine the necessary amount of resources to execute them. Usually we are interested in estimating the compexity of an algorithm in asymptotic form rather than it's exact speed. This way, it is easier to estimate the complexity function for reasonably large inputs.

Intuitively, we could just code the algorithm and measure the time of how long it takes for different sized inputs. However, this approach is not healthy since the runtime can vary on different factors such as hardware, programming language, programming style and so on. Therefore, we need a way to measure the algorithms which is not too general nor too precise.

Efficiency of an algorithm can be determined by many criteria, including the amount of work done (time complexity) or the amount of space used (space complexity). Usually, when analysing algorithms, the number of basic operations are good enough to estimate the total number of operations. For example, in matrix multiplication problem, a basic operation would be multiplying two numbers or when sorting a list of numbers, a basic operation would be comparing two numbers or similarly, when traversing a tree, a basic operation would be processing an edge.

For example the rate of growth of the time complexity as the input gets larger (n) would be 1 (constant), $\log n$ (logarithmic), n (linear), n^2 (quadratic) or 2^n (exponential).

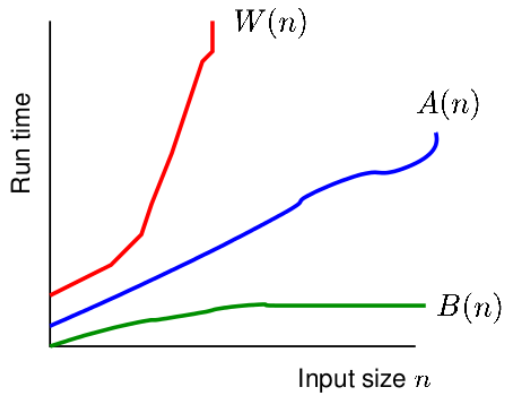


Figure 1: Worst case W(n) vs Average case A(n) vs Best case B(n)

Given sufficiently large input, an algorithm with a lower growth rate will always take less time even though it was run on a slower computer and it was written by a bad programmer. For example, consider two sorting algorithms which we will discuss later in this class: Insertion sort ($c_1 n^2$) and Merge sort ($c_2 n(\log n)$). Assume that insertion sort is coded by the best programmer ($c_1 = 2$) with a low level language on a one billion/second computer. On the other hand, Merge sort is coded by a bad programmer ($c_2 = 50$) with a high level language on a ten million/second computer. Thus, sorting ten million numbers with Insertion sort takes 2.3 days while Merge sort takes 20 minutes.[1]

Therefore, it is important to understand the asymptotic performance of an algorithm in order to compare their relative performance. We use Θ, O, and Ω notations for this purpose:[2]

2 Worst case, average case and best case complexity

2.1 Best case complexity

We use best-case performance to measure the behaviour of the algorithm under optimal conditions. For example, for a list with n items, the best case for linear search algorithm is when the value is equal to the first element of the list, so that we only need to do one comparison.

[1] Example is taken from: cs.iupui.edu/~xkzou/teaching/CS580/Introduction.ppt

[2] All the functions we consider are asymptotically non-negative. What this means is that there exists a n_+ such that for all $n > n_+$ the function $f(n)$ is non-negative.

Given D_n = the set of input size n, $t(I)$ = number of basic operations needed for input I; we can define best-case complexity as the minimum number of basic operations performed by the algorithm on any input of a certain size; $B(n) = min\{t(I)|I \in D_n\}$.

2.2 Average case complexity

In average case complexity, we measure the average number of basic operations performed by the algorithm on any input of a certain size. For example, for linear search algorithm, if we assume that each element in the list is equally likely to be the value that we were looking for, algorithm only visits $n/2$ elements on average. Given D_n = the set of input size n, $t(I)$ = number of basic operations needed for input I and $P(I)$ is the probability that input I occurs; then average case complexity is: $\sum I \in D_n P(I).t(I)$.

We can estimate the probability of occurance by experience, assumption (e.g. all numbers are equally likely to match in linear search).

2.3 Worst case complexity

Worst-case complexity is usually the most important one comparing to others since we often want a guarantee that the algorithm will always finish on time. For example, for linear search algorithm, the worst case is when the value we are looking for is not in the list or when it's at the end of the list (therefore, we have to make n comparisons).

Given D_n = the set of input size n, $t(I)$ = number of basic operations needed for input I; we can define worst-case complexity as the maximum number of basic operations performed by the algorithm on any input of a certain size; $W(n) = max\{t(I)|I \in D_n\}$.

3 Order of Growth

We saw average, best and worst-case time complexity, however exactly determining $W(n)$, $W(n)$ or $A(n)$ is very hard and we want an asymptotic analysis to compare algorithms against one another. Main intuition here is when input size n goes to infinity, we ignore constant factors and drop lower order terms in order to come up with an asymptotic growth rate. Figure 2 and Figure 3 [3] shows a comparison between order of growths.

[3] Images are taken from jriera.webs.ull.es/Docencia/lec1.pdf

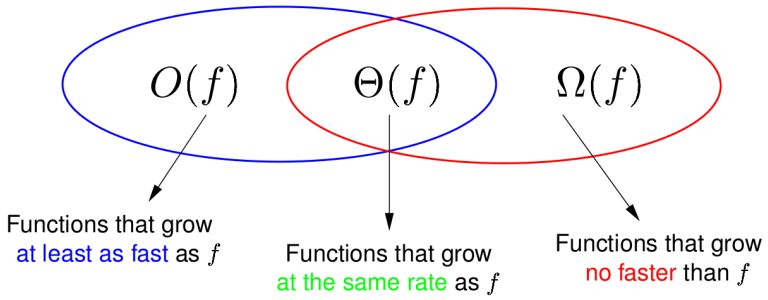


Figure 2: Big O vs Omega vs Big Theta Orders

3.1 Θ Notation

Denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) := \{f(n) : \exists\, c_1, c_2, n_0 > 0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}.$$

In other words, $f(n)$ is sandwiched between $c_1 g(n)$ and $c_2 g(n)$. Although technically $\Theta(g(n))$ is a set, and we should write $f(n) \in \Theta(g(n))$, we will abuse notation and write $f(n) = \Theta(g(n))$.

Another way to test if $f(n) = \Theta(g(n))$ is to check if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{const.} > 0$$

In summary, Θ class gives a lower bound and upper bound on complexity of a function. $\Theta(f)$ is the set of functions that grow **at the same rate** as f. $c_2 g(n)$ is an upper bound on $f(n)$ and $c_1 g(n)$ is a lower bound on $f(n)$.

3.2 O Notation

The O notation provides an asymptotically upper bound. Denote by $O(g(n))$ the set of functions

$$O(g(n)) := \{f(n) : \exists\, c, n_0 > 0 \text{ s.t. } 0 \leq f(n) \leq c g(n)\ \forall n \geq n_0\}.$$

$\emptyset(g(n))$ is the set of functions that grow no faster than $g(n)$.

Note: The O notation bounds the worst case performance of the algorithm. But saying that an algorithm is $O(n^2)$ does **not** imply that the algorithm takes cn^2 time for *every* input.

3.3 Ω Notation

The Ω notation provides an asymptotically lower bound. Denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) := \{f(n) : \exists\, c, n_0 > 0 \text{ s.t. } 0 \leq c g(n) \leq f(n)\ \forall n \geq n_0\}.$$

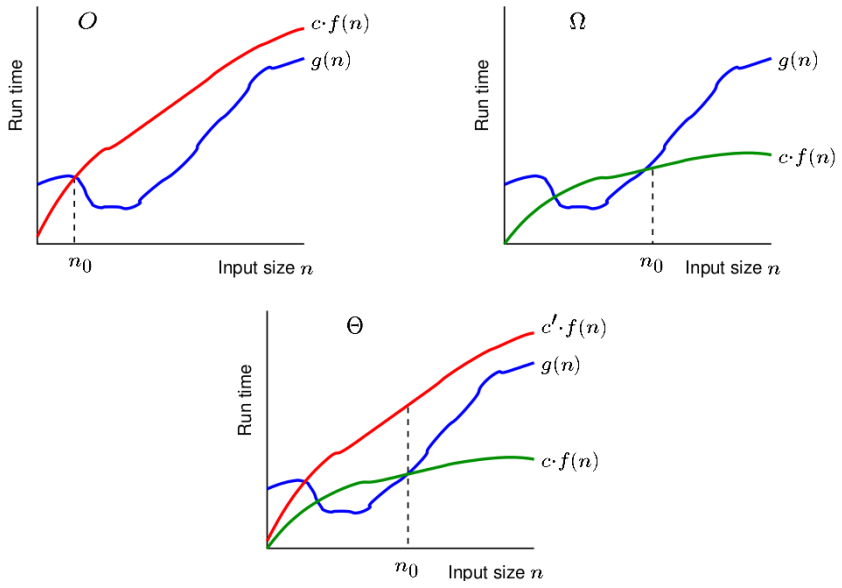


Figure 3: Big O vs Omega vs Big Theta Orders

$\Omega(g(n))$ is the set of functions that grow at least as fast as $g(n)$.

Note: The Ω notation bounds the best case performance of the algorithm. Saying that an algorithm is $\Omega(n^2)$ implies that the algorithm takes at least cn^2 time for *every* input.

Theorem 1 *For any two function f and g we have $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

3.4 o Notation

The bounds provided by the O notation might not be asymptotically tight (For instance, $n = O(n^2)$). In such cases we use the o notation. Denote by $o(g(n))$ the set of functions

$$o(g(n)) := \{f(n) : \exists\, n_0 > 0 \text{ s.t. } 0 \leq f(n) < cg(n)\ \forall\, c > 0 \text{ and } n \geq n_0\}.$$

Carefully note the order of the qualifiers. The main difference from the O notation is that the upper bound now holds for all constants c. Intuitively, in the o notation the function $f(n)$ becomes insignificant compared to $g(n)$ as $n \to \infty$. Another way to see this is via limits:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

3.5 ω Notation

Denote by $\omega(g(n))$ the set of functions

$$\omega(g(n)) := \{f(n) : \exists\ n_0 > 0 \text{ s.t. } 0 \leq cg(n) < f(n)\ \forall\ c > 0 \text{ and } n \geq n_0\}.$$

In terms of limits:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

Roughly, the English interpretations of the notations would be: $O(f)$; functions that grow **no faster** than f, $\Omega(f)$; functions that grow **no slower** than f, $\Theta(f)$; functions that grow at the **same rate** as f, $o(f)$; functions that grow **slower** than f, $\omega(f)$; functions that grow **faster** than f.

3.6 L'Hospital's Rule

If f and g are two functions which satisfy either

$$\lim_{n\to c} f(n) = \lim_{n\to c} g(n) = 0$$

or

$$\lim_{n\to c} g(n) = \pm\infty \text{ and } \lim_{n\to c} \frac{f'(n)}{g'(n)} \text{ exists.}$$

Then

$$\lim_{n\to c} \frac{f(n)}{g(n)} = \lim_{n\to c} \frac{f'(n)}{g'(n)},$$

where $'$ denotes the derivative. This is often used to simply computation of the limits.

3.7 Recurrences

Sometimes we can solve a problem by first solving smaller sub-problems and using their solutions to solve the larger problem. As a concrete example, consider the following pseudo-code for sorting an array of n numbers:

The time taken by the merge sort algorithm to sort n numbers can be written as the following recurrence:

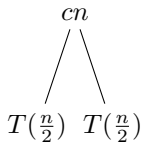
$$T(n) = 2T(n/2) + cn.$$

Here c is a constant and the cn denotes the time taken to merge two sorted sub-arrays of size $n/2$ into a single sorted array. Note our slightly sloppy notation; we implicitly assume that n is always divisible by 2.

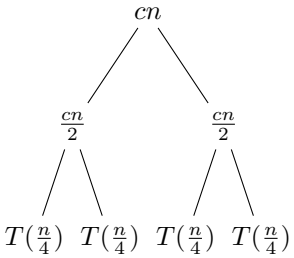
Algorithm 1 Merge Sort

1: **Input:** Array of size n
2: **if** $n = 1$ **then**
3: **Return:** input array
4: **else**
5: Split array into two sub-arrays of size $n/2$
6: Apply Merge Sort on the two sub-arrays
7: Merge the two sorted sub-arrays into a single sorted array
8: **end if**
9: **Return:** Sorted array

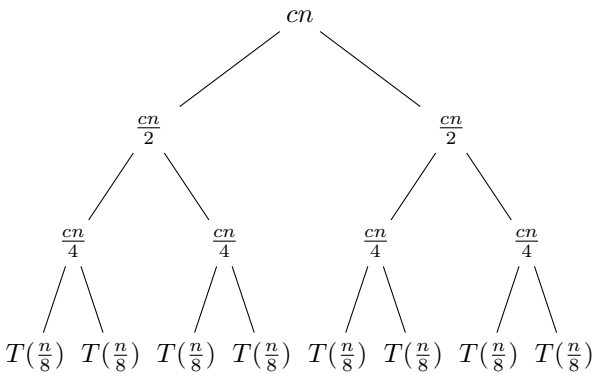
In order to deduce the time complexity of merge sort we write out a recurrence tree. First observe that to solve a problem of size n we need to solve two sub problems of size $n/2$ and expend cn effort to merge the two solutions together. This is represented by the following tree:



Next we observe that solving each problem of size $n/2$ entails solving a problem of size $n/4$ and then expending $cn/2$ work to merge the solutions. This is represented by the following expanded tree:



Continuing the same line of argument one gets to the next level:



and so on and so forth. It is easy to see that this process will stop when the size of the input is 1 (a single element need not be sorted). This happens after k levels, where $n/2^k = 1$ or in other words when $k = \log_2(n)$. The total work done by the algorithm can now be found by summing over all the nodes of the tree. Observe that at every level of the tree we need to expend cn work. Since there are $\log_2(n)$ levels, one can easily conclude that $T(n) = O(n \log_2(n))$.

Exercise: Write the recursion tree for the following recurrences and use it to find the complexity of $T(n)$ in O (or if applicable Θ) notation.

$$T(n) = 2T(n/2) + 1$$
$$T(n) = T(n/2) + 1$$