Computational Complexity

S. V. N. Vishwanathan

January 24, 2013

Intuitively 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*. 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:[1]

0.1 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

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

0.2 O Notation

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

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

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.

[1] 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.

0.3 Ω Notation

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

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

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 $f(n) = \Omega(g(n))$.*

0.4 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:

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

0.5 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:

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

0.6 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.

0.7 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:

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
```

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

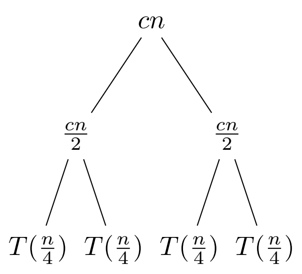
$$T(n) = 2T(n/2) + cn.$$

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.

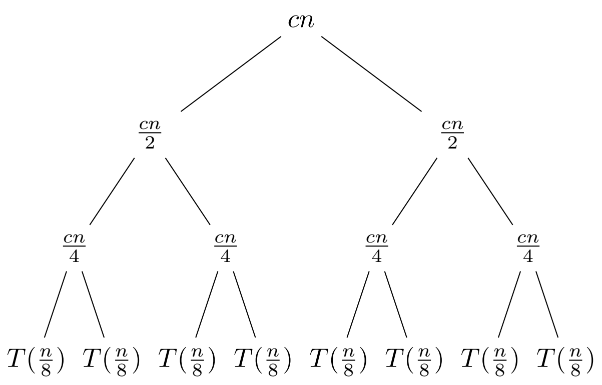
In order to deduce the time complexity of merge sort we write out a recurrence tree. First observe that to solve a problem of size n we need to solve two sub problems of size $n/2$ and expend cn effort to merge the two solutions together. This is represented by the following tree:

$$\begin{tikzcd} & cn \arrow[dl, no head] \arrow[dr, no head] & \\ T(\frac{n}{2}) & & T(\frac{n}{2}) \end{tikzcd}$$

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:


cn
$\frac{cn}{2}$
$\frac{cn}{2}$
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$

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


cn
$\frac{cn}{2}$
$\frac{cn}{2}$
$\frac{cn}{4}$ $\frac{cn}{4}$ $\frac{cn}{4}$ $\frac{cn}{4}$
$T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$

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.

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