ICCS313 - Algorithm Analysis, T1 2019/20

Lecture 1-2: Asymtotic Notations

14 September 2019

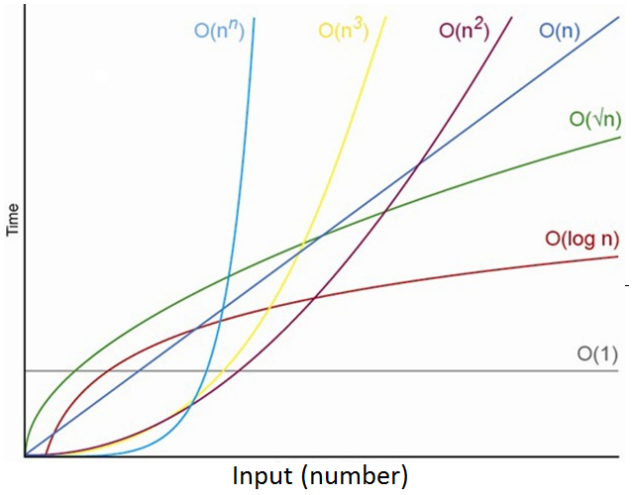
Lecturer: Chaya Hiruncharoenvate *Scribe: -*

This is a review for asymptotic notations. For more details, read CLRS Chapter 3. This lecture note is adapted from previous year's by Dr.Sunsern Cheamanunkul.

1 Review of Asymptotic Notations

Recall that we are really only interested in the *Order of Growth* of an algorithm's complexity i.e. how well the algorithm perform as the input size $n \to \infty$.

Potentially, we could write out the cost functions of algorithms with respect to their input size n (or number of elementary operations). However, in practice, we really care about its asymptotic behavior anyway.



Why asymptotic analysis? We want to analyze the algorithm independent of hardware or implementations. For example, suppose we have 2 machines where machine A is 100x faster than machine B. The very same algorithm running on machine A has running time of $2n^2$ and on machine B has the running time $200n^2$. In this scenario, we consider $2n^2$ and $200n^2$ to be *asymptotically equivalent*.

Definition 1.1 (Big-O)**.** Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in O(g(n))$$

if there exists a constant $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n)$$

Definition 1.2 (Big-Omega)**.** Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in \Omega(g(n))$$

if there exists a constant $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$f(n) \geq cg(n)$$

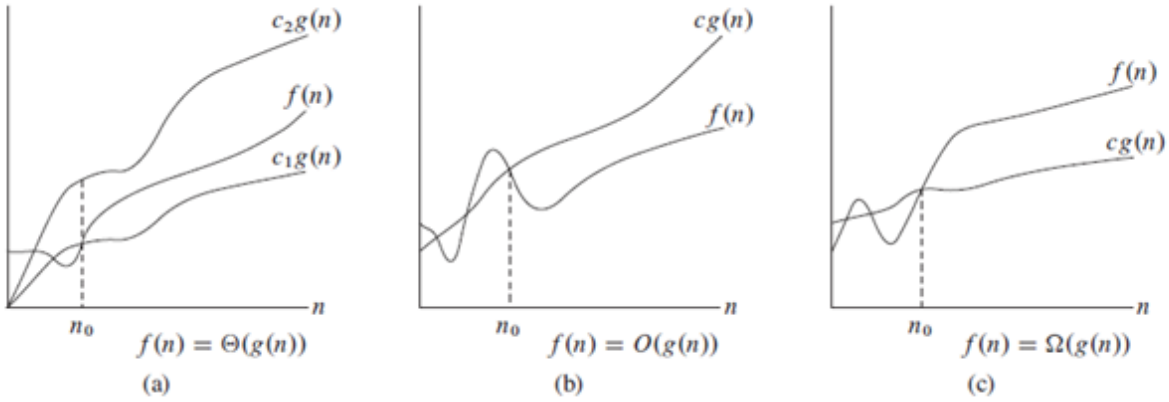
Definition 1.3 (Big-Theta)**.** Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in \Theta(g(n))$$

if there exists a constant $c_1, c_2 \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Here's a visual comparison of O, Ω, and Θ.



1.1 Useful Corollary

Corollary 1.4. *Given $f(n)$ and $g(n)$, if $f(n) \in \Theta(g(n))$, then $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$*

Corollary 1.5. *Given $f(n)$ and $g(n)$, if $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$*

Example: Show that $T(n) = 32n^2 + 17n + 1$ is $O(n^2)$

Proof.

$$T(n) = 32n^2 + 17n + 1 \leq 32n^2 + 17n^2 + n^2; \ \forall n \geq 1$$
$$T(n) \leq 50n^2; \ \forall n \geq 1$$

So we can say that $T(n) \leq cn^2$ when $c = 50$ and for all $n \geq n_0 = 1$ □

1.2 Limit Method

Apart from using formal definitions, you may use *limit method* to show Big-O, Big-Theta, Big-Omega. Given $f(n)$ and $g(n)$. We consider the following limit:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0 & \text{then } f(n) \in O(g(n)) \\ c > 0 & \text{then } f(n) \in \Theta(g(n)) \\ \infty & \text{then } f(n) \in \Omega(g(n)) \end{cases}$$

You have to be careful when f and g both diverge or converge to 0 or ∞. Then, you need to apply L'Hospital's Rule.

Reminder: L'Hospital's Rule states that if

$$\lim_{x\to a}\frac{f(x)}{g(x)} = \frac{0}{0} \text{ or } \lim_{x\to a}\frac{f(x)}{g(x)} = \frac{\pm\infty}{\pm\infty}$$

where a can be any real number, infinity, or negative infinity, then

$$\lim_{x\to a}\frac{f(x)}{g(x)} = \lim_{x\to a}\frac{f'(x)}{g'(x)}$$

Example: Show that $2^n \in O(3^n)$.

Proof. Let's use limit method.

$$\lim_{n\to\infty}(\frac{2^n}{3^n}) = \lim_{n\to\infty}(\frac{2}{3})^n$$

We know that $\lim_{n\to\infty}\alpha^n = 0$ when $\alpha < 1$. Therefore, we conclude that $2^n \in O(3^n)$ □

2 Asymptotic Analysis In Practice

- Loops – The more nested loops the algorithm contains, the slower it is. If there are k nested loops, the time complexity is $O(n^k)$.
- Phases – If the algorithm consists of consecutive phases, the total time complexity is the largest time complexity of a single phase.

Example: What is the running time in term of O-notation of the following algorithm?

```
1 for (i=0;i<n;i++) {
2   for (j=0;j<n;j++) {
3     ...
4   }
5 }
6 for (i=0;i<2*n;i++) {
7
8 }
```

3 Complexity classes

- Constant – $O(1)$
- Logarithmic – $O(\log n)$
- Polylogarithmic – $O(\log^k n)$
- Sub-linear – $O(\sqrt{n})$
- Linear – $O(n)$
- Quasilinear or log-linear – $O(n \log n)$
- Quadratic – $O(n^2)$
- Cubic – $O(n^3)$
- Exponential – $O(2^n)$
- Factorial – $O(n!)$

4 Analysis of Insertion Sort

Let's revisit our Insertion Sort algorithm and examine its running time:

```
InsertionSort(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j-1
4      while i > 0 and A[i] < key
5          A[i+1] = A[i]
6          i = i-1
7      A[i+1] = key
```

Let n be the length of array A. Let C_i be the running time of line i. Let t_j be the number of times the while loop is executed for the value of j.

Cost	times
C_1	n
C_2	n-1
C_3	n-1
C_4	t_2+t_3+t_4... = $\sum_{j=2}^{n} t_j$
C_5	$\sum_{j=2}^{n} (t_j$-1)
C_6	$\sum_{j=2}^{n} (t_j$-1)
C_7	n-1

4.1 Best-case running time

If A is already sorted, $t_j = 1 \; \forall \; 2 \leq j \leq n$

$$T(n) = C_1(n) + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} 1 + C_5 \sum_{j=2}^{n} (1-1) + C_6 \sum_{j=2}^{n} (1-1) + C_7(n-1)$$
$$= C_1(n) + (C_2 + C_3 + C_4 + C_5)(n-1)$$

So, $T(n) \in O(n)$

4.2 Worst-case running time

If A is reverse sorted, $t_j = j \; \forall j$

$$T(n) = C_1(n) + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} j + C_5 \sum_{j=2}^{n} (j-1) + C_6 \sum_{j=2}^{n} (j-1) + C_7(n-1)$$

We know that

$$\sum_{j=2}^{n} j = 2 + 3 + 4 + \ldots + n = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = 1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

Therefore,

$$T(n) = C_1(n) + C_2(n-1) + C_3(n-1) + C_4(\frac{n(n+1)}{2} - 1) + C_5(\frac{n(n-1)}{2}) + C_6(\frac{n(n-1)}{2}) + C_7(n-1)$$

Hence, $T(n) \in O(n^2)$

4.3 Show Correctness using Loop Invariant

Loop invariant is a common technique used to help us understand why an algorithm is correct. First, we state a useful invariant that will help us in showing correctness. Then, we must show three things:

1. Initialization – The invariant is true before the first iteration of the loop.
2. Maintenance – If it is true before iteration, it remains true before the next.
3. Termination – When the loop terminates, the invariant give us useful property that helps show correctness

You can think of loop invariant as a variant of proof by induction.

Claim 4.1. *Insertion Sort algorithm is correct.* [1]

Proof. Consider the following loop invariant:

At the start of each iteration of the for loop, the subarray `A[1...j-1]` *consists of the elements originally in* `A[1...j-1]` *but in sorted order*

Then, we show:

- Initialization – This is true, when we start $j = 2$, `A[1..1]` is just one single number, which we consider as sorted.
- Maintenance – If `A[1..j-2]` is sorted before, the current iteration will cause `A[1..j-1]` to be sorted.
- Termination – When $j = n+1$ after the termination, We will have `A[1..n]` in sorted order.

So, the invariant is true. Hence, after the algorithm terminates, this will leave the array A sorted as we want. □

[1] You can read the full formal proof in CLR Chapter 2.