# Big O Notation

CMSC 27230: Honors Theory of Algorithms

January 6, 2023

Corresponding section(s) of Kleinberg-Tardos: 2.2

# 1 Big O Notation

## 1.1 Motivating Example: Finding the Largest Number in an Array

Consider the following simple problem. Given an array of $n$ numbers $x_1, \ldots, x_n$, find the largest number in this array. In other words, find $\max_{i \in [n]} \{x_i\}$. This problem can be solved with the following simple algorithm. Go through the array while keeping track of the largest number we have seen so far. Whenever we see a larger number, we take that number instead. More precisely, we do the following:

1. Stored data: We keep track of the current index $i$ and the largest number $x$ which we have seen so far (i.e $x = \{\max_{j \in [i]} x_j\}$)
2. Inititalization: We start with $i = 1$ and $x = x_1$
3. Iterative step: If $i = n$ then we stop and output $x$. Otherwise we check if $x_{i+1} > x$. If $x_{i+1} > x$ then we set $x = x_{i+1}$ and otherwise we leave $x$ as is. We then increment $i$ by 1 and repeat.

Now consider how many steps this algorithm takes. If we look at comparisons then the algorithm needs $n - 1$ comparisons as it automatically takes the first number and then does one comparison for each number after that. However, comparisons are not the only kind of step which is needed. We also have to increment the array index and replace the largest number we've seen so far if needed. If we dig deeper into what the computer is doing, there will be other steps as well. Thus, figuring out exactly how many steps are needed would be quite painful and would depend heavily on the model.

To avoid this, instead of trying to determine exactly how many steps an algorithm takes, we only try to get the answer up to a constant factor. This allows us to capture the qualitative behavior of the algorithm without being too precise and getting bogged down in the details of how the algoithm is implemented and what we consider to be a step. For example, in the above algorithm, we need a constant number of steps for each new number so the total number of steps is at most $Cn$ for some constant $C$. Using big O notation, which is defined below, we can express this bound by saying that this algorithm takes $O(n)$ steps.

## 1.2 Big O Notation Definition

**Definition 1.1** (Big O Notation)**.** *Given functions $f, g : \mathbb{N} \to \mathbb{R}^+$,*

1. *We say that $f(n)$ is $O(g(n))$ if there exist $n_0, C > 0$ such that $\forall n \geq n_0 (f(n) \leq Cg(n))$.*
2. *We say that $f(n)$ is $\Omega(g(n))$ if there exist $n_0, c > 0$ such that $\forall n \geq n_0 (f(n) \geq cg(n))$.*
3. *We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$. Equivalently, we say that $f(n)$ is $\Theta(g(n))$ if there exist $n_0, c, C > 0$ such that $\forall n \geq n_0 (cg(n) \leq f(n) \leq Cg(n))$.*

**Example 1.2.** *Some examples of Big O notation are as follows:*

1. *If we know that $f(n) \leq 100n + 2nlog(n) + 45$ then $f$ is $O(nlog(n))$ (as $n \to \infty$, $2nlg(n)$ is the dominant term in this upper bound).*
2. *If we know that $f(n) \geq 2n + \frac{n^2}{100} - 50$ then $f(n)$ is $\Omega(n^2)$ (as $n \to \infty$, $\frac{n^2}{100}$ is the dominant term in this lower bound).*
3. *If we know that $\frac{log(n)}{2} - 1 \leq f(n) \leq 2lg(n) + 3$ then $f(n)$ is $\Theta(log(n))$.*

## 1.3 Facts about Big O Notation

**Proposition 1.3.**

1. *Given functions $f, g, h : \mathbb{N} \to \mathbb{R}^+$, if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$.*
2. *Given functions $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}^+$, if $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) + f_2(n)$ is $O(\max\{g_1(n), g_2(n)\})$.*
3. *Given functions $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}^+$, if $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n)f_2(n)$ is $O(g_1(n)g_2(n))$.*

*Moreover, the same statements hold if we replace O by $\Omega$ or $\Theta$.*

*Proof.* For the first statement, there exist $n_1, C_1, n_2, C_2 > 0$ such that

1. $\forall n \geq n_1 (f(n) \leq C_1 g(n))$
2. $\forall n \geq n_2 (g(n) \leq C_2 h(n))$

Putting these statements together, $\forall n \geq \max\{n_1, n_2\} (f(n) \leq C_1 g(n) \leq C_1 C_2 h(n))$.

For the second statement, there exist $n_1, C_1, n_2, C_2 > 0$ such that

1. $\forall n \geq n_1 (f_1(n) \leq C_1 g_1(n))$
2. $\forall n \geq n_2 (f_2(n) \leq C_2 g_2(n))$

Putting these statements together,

$$\forall n \geq \max\{n_1, n_2\} (f_1(n) + f_2(n) \leq C_1 g_1(n) + C_2 g_2(n) \leq (C_1 + C_2) \max\{g_1(n), g_2(n)\})$$

For the third statement, there exist $n_1, C_1, n_2, C_2 > 0$ such that

1. $\forall n \geq n_1(f_1(n) \leq C_1 g_1(n))$
2. $\forall n \geq n_2(f_2(n) \leq C_2 g_2(n))$

Putting these statements together, $\forall n \geq \max\{n_1, n_2\}(f_1(n)f_2(n) \leq C_1 C_2 g_1(n)g_2(n))$.

The corresponding statements for $\Omega$ and $\Theta$ can be proved in a similar way. □

**Remark 1.4.** *If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, this does not imply that $\frac{f_1(n)}{f_2(n)}$ is $O(\frac{g_1(n)}{g_2(n)})$ because we have an upper bound on $g(n)$ but we don't have a lower bound on $g(n)$. For example, we could take $f_1(n) = n^2$, $f_2(n) = 1$, $g_1(n) = n^2$, and $g_2(n) = n$. If so, we have that $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ but $\frac{f_1(n)}{f_2(n)} = n^2$ is not $O(\frac{g_1(n)}{g_2(n)}) = O(n)$.*

*That said, if $f_1(n)$ is $\Theta(g_1(n))$ and $f_2(n)$ is $\Theta(g_2(n))$, this does imply that $\frac{f_1(n)}{f_2(n)}$ is $\Theta(\frac{g_1(n)}{g_2(n)})$*

## 1.4 Comparing Logarithms, Polynomials, and Exponential Functions

It is important to know that as $n$ goes to infinity, logarithms grow more slowly than any polynomial while exponential functions grow more quickly than any polynomial. More precisely, we have the following statements:

**Lemma 1.5.**

1. $\forall C, c > 0\ \exists n_0 > 0\ \forall n \geq n_0\ (Clog_2(n) \leq n^c)$
2. $\forall C, c, c' > 0\ \exists n_0 > 0\ \forall n \geq n_0\ \left(Cn^c \leq 2^{c'n}\right)$

*Proof.* To see the first statement, write $n = m^{\frac{1}{c}}$. We now have $Clog_2(n) = \frac{C}{c} log_2(m)$ while $n^c = m$. Thus, $Clog_2(n) \leq n^c$ whenever $\frac{m}{log_2(m)} \geq \frac{C}{c}$, which is true for sufficiently large $m$.

To see the second statement, observe that $Cn^c \leq 2^{c'n}$ if and only if

$$log(Cn^c) = log_2(C) + clog_2(n) \leq log_2(2^{c'n}) = c'n$$

This is true as long as $\frac{n}{log_2(n)} \geq \frac{log_2(C)}{log_2(n)} + \frac{c}{c'}$ which is true for sufficiently large $n$. □

## 1.5 Big O Notation in an Exponent

We can also use Big O notation in an exponent but we should be careful when doing so.

**Example 1.6.** *If we say that an algorithm takes $n^{O(1)}$ time, this means that if $T(n)$ is the maximum number of steps the algorithm takes on an input if size $n$ then*

$$\exists n_0, C > 0\ \forall n \geq n_0\ \left(T(n) \leq n^C\right)$$

*A more common way to say this is that the algorithm takes polynomial time.*

**Example 1.7.** *If we say that an algorithm takes $2^{O(n)}$ time, this means that if $T(n)$ is the maximum number of steps the algorithm takes on an input if size $n$ then*

$$\exists n_0, C > 0\ \forall n \geq n_0\ \left(T(n) \leq 2^{Cn}\right)$$

*A more common way to say this is that the algorithm takes exponential time.*

**Remark 1.8.** *Note that the following two statements are not the same:*

1. $T(n)$ *is* $O(2^n)$
2. $T(n)$ *is* $2^{O(n)}$

*The first statement implies the second statement but the second statement does not imply the first statement. To see this, note that if the first statement holds then* $\exists C, n_0 \left(T(n) \leq C2^n\right)$ *so for all* $n \geq \max\left\{n_0, \lceil log_2(C) \rceil\right\}$, $T(n) \leq C2^n \leq 2^{2n}$. *However, if* $T(n) = 4^n = 2^{2n}$ *then the second statement would be true but the first statement would be false.*