Time Complexity and Big-O Notation

Alyssa Lytle

Fall 2025

1 Motivation

We've talked about problems being computationally solvable *in principle*, but what about in practice? *Complexity Theory* is the investigation of the time, memory, and other resources required to solve computational problems [Sip96]. For now, we will focus on time complexity specifically.

Time complexity measures how long an algorithm will take to compute. We can (and do) consider average case, best case, or worst case scenarios.

Why is this important? There are many real-world examples that are impacted by time complexity.

- Security: Modern cryptography relies on the fact that decryption and finding private keys takes too long to compute.
- User Experience: People want fast interfaces.
- Safety: Critical systems sometimes need to make fast decisions, and timing can be a matter of life and death.

Some questions we may want to answer about the speed of an algorithm are: Is this problem solvable on a modern computer in a reasonable amount of time? Does this problem take as long to solve as this other problem? Is this algorithm faster than this other algorithm?

For this lesson, we will work on formalizing the idea of time complexity to answer these questions directly.

2 Formalizing Runtime

The main approach to talking about the time complexity to solve a problem is to consider a specific algorithm used to solve it.

E.g. "I have a turing machine M that decides problem A. How much time does the algorithm for M take to decide A?"

This is mainly reasoned about as a number of "steps" taken to complete the algorithm. In time complexity, we consider the steps taken based on the size of the input. In other words, "How long will it take M to decide if w is in A?" will be determined by and expressed in terms of the size of w. Typically, we use $n = |w|$.

Definition: Running Time

Let M be a Turing machine that halts on all inputs. The *running time* or *time complexity* of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.
If $f(n)$ is the running time of M we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input. [Sip96]

We typically talk about $f(n)$ in terms of its bounds. As said previously, we can consider average case, best case, or worst case scenarios. What this really means is we consider average case, best case, or worst case *inputs*. "What w, with $|w| = n$, will cause M to make the most steps to decide if $w \in A$? How many steps will that be?"

For this lesson, we will focus on worst case runtime, or the upper bound of $f(n)$. This is expressed using the *big-O notation* for $f(n)$.

Definition:

Let f and g be functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

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

[Sip96]

In this definition, $g(n)$ would be called an upper bound for $f(n)$.

Essentially, this means that $f(n) = O(g(n))$ if f is less than or equal to $g(n)$ for a high enough n if we ignore constants.

Generally, g is determined by choosing the highest order term and suppressing its constant.

E.g. If $f(n) = 5n^3 + 2n^2 + 22n + 6$, g would be determined by choosing the highest order term $5n^3$ and supressing the constant 5. So, $g(n) = n^3$ and $f(n) = O(g(n)) = O(n^3)$.

Example: Big-O

For any $f(n) = O(\log_x n)$ we can write $f(n) = O(\log n)$. In other words we can suppress the log base. Why?

We can use the fact that $\log_b n = \frac{\log_x n}{\log_x b}$. Therefore, $\log_x n = (\log_b n)(\log_x b)$. Since $\log_x b$ is a constant, $\log_x n = O(\log_b n)$

Another definition that may be useful is small-o notation, or $o(n)$. This essentially lets you express that $f(n)$ is strictly smaller than $g(n)$.

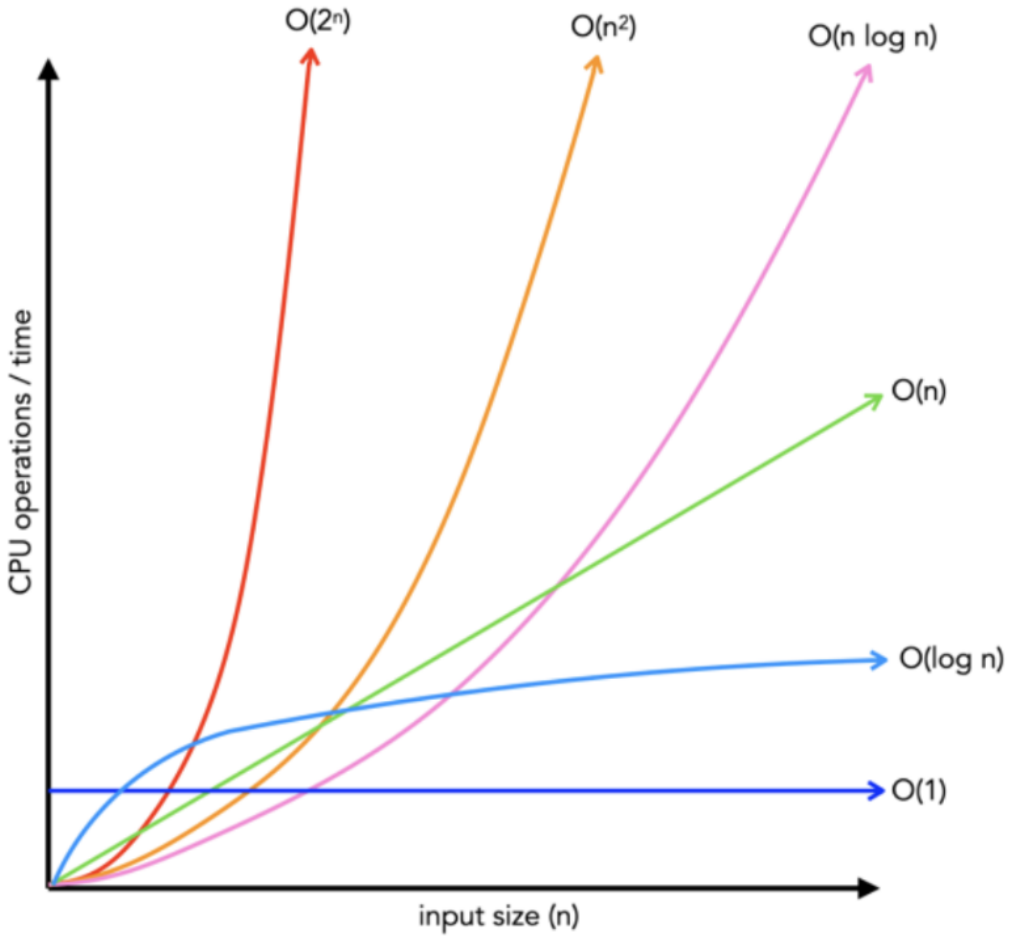
Definition: Small-o

et f and g be functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = o(g(n))$ if

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

[Sip96]

In other words, $f(n) = o(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0, f(n) < cg(n)$.



3 Analyzing and Comparing Algorithms

Now, we have a way to classify some algorithms based on their runtime. For example, problems with "linear runtime" are ones that run in $O(n)$ time. We can formally define time complexity classes.

Definition: Time Complexity Class

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the *time complexity class*, TIME($t(n)$), to be the collection of all languages that are decidable by an $O(t(n))$ time Turing Machine. [Sip96]

Example:

TIME(n^2) contains all languages that can be decided in $O(n^2)$ time.
So, if there exists a machine M that decides A in time $O(n^2)$, we say $A \in$ TIME(n^2)

4 Time Complexity Class P

Now, we will introduce the class of languages decidable in polynomial time, P.

Definition: P

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_k \text{TIME}(n^k).$$

[Sip96]

P is important because it describes the class of problems that are realistically solvable on a computer.

4.1 Problems in P

Now we can talk about some examples of problems that are in P and show how we can prove this. We prove a problem is in P by presenting an algorithm to solve it in polynomial time.

Essentially for our proof we should:

1. Express our problem as a language A
2. Present a Turing machine computable algorithm M that decides A
3. Argue the runtime of M

Example: S

ay we have a directed graph G that contains nodes s and t. The problem we want to solve is whether or not there exists a direct path from s to t in G.
So, first let's express this as a language:

$$PATH = \{\langle G, s, t\rangle | G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

Now, let's define an algorithm M to decide this.

M ="On input $\langle G, s, t\rangle$,

1. Place a mark on node s
2. Repeat until no additional nodes are marked:
3. Scan all edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Finally, we have to argue that M is in P by reasoning about its runtime.
Steps 1 and 4 are only computed once. Step 3 is potentially repeated once for every node in G, so m for m nodes in G.
Therefore the total runtime is $1 + 1 + m$.
Let n be the length of $\langle G, s, t\rangle$. Since m is the number of nodes in G, n is directly dependent on m, so $1 + 1 + m = O(n)$.
Note if we did this by brute force, the number of potential paths over each node is m^m so that's a lot of searching... and not polynomial!

References

[Sip96] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.