10-600: **M**achine **L**earning **P**rimer

Matt Gormley

2022

Contents

Chapter 1

Big-O

1 Overview

We can measure the run time, spatial complexity, or general resource consumption of a given function through Big-O notation. Informally, if the function `my_fn(x)` runs in $O(f(n))$, this means that when the input `x` is of size n, the worst-case scenario run time of `my_fn(x)` is asymptotically a constant multiple of $f(n)$.

Note that run time here refers to the number of algorithmic steps that the function takes rather than wall-clock time.

2 Definition and Mathematical Properties

For the purposes of this course, (and *most* future MLD courses you may or may not take), it's best to think of big-O for comparing two arbitrary functions, which we will call f and g respectively. For simplicity, we'll assume that both f and g are defined over $\mathbb{R}_+$ (that is, f and g only take positive real numbers as inputs) only, which is enough for this course. Formally, Big-O is defined as:

Definition 1.1 (Big-O Notation)**.** If $f(n) \in O(g(n))$, then there exists some $c, n_0 \in \mathbb{R}_+$ such that:

$$\forall n \geq n_0, \quad f(n) \leq cg(n)$$

(Note: if you've seen Big-O before, you may be more familiar with the notation $f(n) = O(g(n))$. We avoid using this alternative as the equality operator implies a certain degree of symmetry that is not present.)

In words, this means that if $f(n) \in O(g(n))$, then at some point as the inputs increase, $f(n)$ will ***always*** be less than or equal to a constant multiple of $g(n)$.

2.1 An aside on verbiage

It can sometimes be confusing to talk about big-O in the context of computer science and the math behind it, as functions like $n!$ are simultaneously described as "fast-growing" and "slow". This is not incorrect: calling $n!$ "fast-growing" refers to the fact that as n increases $n!$ quickly explodes (1, 1, 2, 6, 24, 120,720, 540, ...), while calling $n!$ "slow" refers to the fact that an algorithm that runs in $O(n!)$ time takes a constant multiple of $n!$ steps to run, which can take a long time! Normally, "fast-growing" is used more when talking about big-O in the pure mathematical sense (e.g. comparing arbitrary functions), while "slow" is used more in computer science when talking about run-time complexity, so we expect you to be comfortable with both!

2.2 Standard Complexity Classes

While the big-O bound for any function $f$can be defined by any other valid function g, in computer science we generally only talk about the following complexity classes (shown in increasing order of complexity):

1. $O(1)$: Constant time
2. $O(\log(n))$: logarithmic time
3. $O(n)$: linear time
4. $O(n\log(n))$: log-linear time
5. $O(n^2)$: quadratic time
6. $O(n^p)$: polynomial time, for some $p > 2$
7. $O(p^n)$: exponential time
8. $O(n!)$: factorial time

2.3 Big-O properties

Based on this definition, we get a few interesting properties as consequence:

Definition 1.2 (Transitivity)**.** For three functions f, g, h all on $\mathbb{R}_+$, if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$.

A result of the transitivity property is that a function f does not have a unique big-O bound, as we can always come up with faster growing functions than f. However, in computer science we normally want the *tightest* Big-O bound in order to formalize what realistically happens "in the worst case."

Definition 1.3 (Lower-Order Ambivalence)**.** If $f(n) = f_1(n) + f_2(n)$ for some functions f_1, f_2 such that $f_2(n) \in O(f_1(n))$, then $f(n) \in O(f_1(n))$.

In words, this means that we only care about the *fastest* growing part of a function to determine its Big-O complexity. So if $f(n) = 5n^3 + 2n$, then $f(n) \in O(n^3)$ (i.e. we can disregard the $2n$ term, as asymptotically $5n^3$ becomes much larger than $2n$).

Definition 1.4 (Log Equivalence)**.** If $f(n) = \log_p(n)$ and $g(n) = \log_q(n)$ for two bases $p, q \in \mathbb{N}$, then $f(n) \in O(g(n))$ **and** $g(n) \in O(f(n))$.

To show that this is true, recall the log rule that $\log_a(b) = \frac{\log(b)}{\log(a)}$ where $\log$ is the natural log. Given this, $f(n) = \frac{1}{\log(p)} \log(n)$ and $g(n) = \frac{1}{\log(q)} \log(n)$, and thus $f(n)$ and $g(n)$ are equivalent up to a constant.

3 Code Examples

Algorithm 1.5 (H)**.** Linear Search

```
procedure LINEARSEARCH(list A, size of list n, value v)
    for i = 0 to n − 1 do
        if A[i] == v then return i

```

–