CIS 575. Introduction to Algorithm Analysis
Material for January 22, 2024

Motivating Asymptotic Notation

The topic of this note is also presented in *Cormen*'s Section 3.1.

1 The Essence of Resource Use

For a given algorithm, one may do a very detailed and elaborate analysis of its running time (or space use), as a function of the size of its input (often called n), and parametrized by the time it takes to execute a given (kind of) instruction. (An example of such an analysis, for the insertion sort algorithm, can be found on pages 30–31 of the *Cormen* textbook.) Assuming we know the details of a given implementation on a given machine, the running time may be expressed by an expression that could be rather complex, for example

$$4\mathbf{n^2} + 27n + 48$$

where we have put in boldface what we claim really matters: n^2, the quadratic part. But this seems a bold claim: does the constant 4 not matter, and does the expression $27n + 48$ not matter? We shall next argue why we shall (mostly) choose to ignore them.

1.1 Constants Don't Matter

Assume we have two algorithms, and we know that on a given machine we can implement the first to run in time $4n^2\ \mu s$, while we can implement the second to run in time $2n^2\ \mu s$. Naturally, we would prefer the second algorithm. But observe that

- it may be possible to do some simple optimization of the first algorithm so it runs twice as fast
- if processors keep getting faster, the future performance of the first algorithm will eventually catch up with the current performance of the second algorithm.

We may conclude that the second algorithm is not fundamentally better than the first algorithm. In this course, we shall focus on performance issues that can *not* be fixed by simple optimizations, or by increased processor speed.

1.2 The Long Run Matters

Assume we have two algorithms, and we know that on a given machine we can implement the first to run in time $1,000n\ \mu s$, while we can implement the second to run in time $n^2\ \mu s$.

If we are going to run this on only very small input, we would prefer the second algorithm. But when n is 1,000, both algorithms take 1 second to execute. And if n is 10,000, the first algorithm takes 10 seconds, while the second takes 100 seconds. In general, we would therefore prefer the first algorithm since it better scales to large input.

> Of course, this could be taken to extremes: for all practical purposes, we would prefer an algorithm with running time in $(1.000000000001)^n$ to an algorithm with running time in $n^{10000000}$, even though the latter is better (or rather not quite as horrible) than the former "in the long run" (when n gets exorbitantly large). But to quote the famous economist John Maynard Keynes (arguing for policies that have short-term benefits even though they may cause long-term damage): *In the long run we are all dead.*

In this course, for a given algorithm we shall focus on its *asymptotic* behavior, that is, how it performs as its input grows larger and larger.

Exponential is Bad An algorithm with exponential running time is infeasible for all but very small input. For example, assume that an algorithm depends on exploring all subsets of a given set; if that set has n elements, there will be 2^n subsets (for $n = 40$ there will be around one trillion subsets). For such algorithms, even input of moderate size will exhaust machine capacity (and/or our patience).

> While people often in their daily discourse talk about "exponential growth", this is usually only for a limited time. For example, not long ago it was of huge concern when in some area, the number of people infected with the new coronavirus doubled at regular intervals, the hallmark of exponential growth. Still, we knew that (even without measures such as social distancing) the exponential growth couldn't continue long, and that eventually the number of infections would go down (though this might not have happened until most of the population had been infected, and too many people had died...)