1 Goals

To introduce and motivate asymptotic notation. Give students some practice at using asymptotics. Show how to analyse order of running times of simple algorithms.

2 The need for asymptotics

2.1 Goal:

This part of the class explains why asymptotic notation is used by algorithm designers. By thinking of algorithms within the broader context of the entire computer architecture, students realize that we are limited in our ability to assign exact times to algorithms.

In addition to the particular topic, the discussion will encourage students to think of what we are learning as a micro-cosm of computer science as a whole. It should reinforce or foreshadow (depending on the previous courses a student has taken) concepts from programming languages and computer architecture.

By discussing the various scales at which algorithms are used, we motivate the need for understanding how algorithm times will scale. The conclusion is that asymptotic analysis (to be introduced next) is a reasonable compromise between what we need to know about algorithms and what we can know.

2.2 Discussion:

Lecturer (as students enter, repeating if necessary): Algorithms break up complex tasks into a series of steps. For example, in bubble sort from last class, in each iteration, we compare the current element to the one in front of it in the array, and then swap them if they are out of order. So the steps involve comparing, swaps, and also keeping track of where we are in the algorithm.

I’d like you to, over the next few minutes, break up into small groups and discuss:

(1st slide) What counts as a single step of a program? Does it depend on the algorithm? On the programming language?

(Possible answers: comparisons, arithmetic, read access, memory access, following pointers, Boolean operations, what functions the CPU has in hardware, instructions in PL get translated into assembly instructions, probably lots of other good points.)

Discussion: Going around the room, instructor gets groups to contribute types of operations, and compiles a list on black-board. Admits that “single step” is a somewhat ambiguous concept, and does depend on circumstances. If appropriate, for some of the list of operations that are normally viewed as single steps, give circumstances (e.g., arithmetic on long numbers) where this may not be valid.
The time for a “single step” should not increase with the input size. When it does, we need to change what a “single step” is.

Move on to second discussion point. “OK, in the same groups, take a few minutes to discuss the following question:”

What affects the actual time a single step takes?

(Possible answers: frequency of machine cycle, whether operation is hardware or software, how the step gets interpreted or compiled, contention with other users for resources, amount of memory, whether the data is already in cache, parallelism (both hardware and multi-core), many others.)

Discussion: Compile a list of factors on board in a similar way. For each, ask does the algorithm designer get to control this factor? I can’t think of any where the answer is yes.

A) Different types of steps require different exact times.
B) The algorithm controls how many times steps are performed, but the time steps take is outside the control of the algorithm designer.

So if we wrote a formula for say, the time Bubble Sort takes, it might look something like

\[ T(n) = \frac{n(n-1)}{2}(c_{\text{compare}}) + n(n-1)(c_{\text{write}}) + n^2(c_{\text{increment inside for loop}}) + \ldots \]

where \( c_{\text{compare}} = \) time for a compare step, \( c_{\text{write}} = \) time for a write step, \( c_{\text{increment inside for loop}} = \) time for increment inside for loop.

This is a mess.

We want to re-write this formula in a way that emphasizes: the factors we control; the most significant terms; and the way the time scales for larger inputs. We’d like to ignore the part we don’t control (exact factors) and the small stuff that eventually becomes irrelevant.

Order notation does exactly this. We say that \( T(n) \in O(f(n)) \) if there is some constant \( c > 0 \) and some \( n_0 > 0 \) so that \( T(n) \leq cf(n) \) for all \( n \geq n_0 \).

By treating all constants \( c \) as equivalent, we acknowledge that we do not control the exact time for individual operations, at the same time we allow ourselves to combine the counts for the different operations. By looking at “sufficiently large” \( n \), only those bigger than some \( n_0 \), we allow ourselves not to sweat the small stuff.

So for bubble sort, the formula becomes:

\[ T(n) \in O(n^2), \]

because all of the most significant terms are of the form \( cn^2 \) for some constant \( c \).

Con: it doesn’t tell us the exact time bubble sort will take on specific \( n \).

Pro: it tells us how bubble sort scales succinctly and cleanly. It allows us to compare bubble-sort to other sorting algorithms.

Let’s try it for another algorithm, such as insert sort. (Repeat pseudo-code for insert sort from previous class).

(Repeat formula for number of comparisons, another for moving information. Write them on board).
There are many terms, but the most significant are both \( n^2 \), so just like for bubble sort, we can summarize as \( T(n) \in O(n^2) \).

This doesn’t mean that on every input, bubble sort and insert sort take the same time. It just means that they scale in the same way for inputs of different sizes. If our implementation of bubble sort is twice as fast as our implementation of insert sort on inputs of size 1000, we expect it to also be about twice as fast on inputs of size 10000. Both will take about \( (10,000/1000)^2 = 100 \) times as long for the larger input.

We keep talking about constants versus variables. Both are numbers. What makes a number a “constant”?

Just as we decided for what makes an operation a “single step”, a constant is a number that doesn’t change as the input grows.

So 7 is a constant, as is “The time taken by a compare operation” “The time taken to merge the two lists in mergesort” is NOT a constant, because it will increase as the lists grow.

So order notation is a mathematical tool to help us express the way algorithm time scales to larger inputs without getting caught up in messy details beyond our control.