Handout 0: Running Times, Asymptotics

The purpose of this handout is to clarify, and possibly dispel some misconceptions about, issues related to running times of algorithms and asymptotic bounds. The focus here is conceptual, with minimal mathematical notation. As always, you should refer to the precise mathematical definitions to understand the details and when writing out solutions to problems.

Here is the first thing to understand: the various notions of **running time** (e.g., worst case, best case, "typical" case) and the various **asymptotic notations** (e.g., big-O, big-Omega, big-Theta) are **entirely separate and independent concepts** from each other. Like peanut butter and jelly, each one is perfectly meaningful, important, and useful on its own. But they are even better together. Let's first discuss each one individually, and then see how they work together.

1 Running Times

In this section, put aside everything you (think you) know about asymptotic notation. The focus here is on *algorithms and their running times*, without any reference to asymptotic bounds and notation.

At the most basic level, an algorithm takes some input, does some work according to some specific instructions (its "code"), and possibly halts with some output (or else it runs forever, like in an infinite loop). The *running time* of the algorithm *on that input* is the number of "basic steps" the algorithm performs when given that input.

A first important observation is that an algorithm might take varying numbers of steps on different inputs—even inputs of the same "size".[1] For example, when QuickSort is given a *nearly sorted* array of n elements as input, it finishes in roughly n steps, but when given a *highly unsorted* array, it takes roughly n^2 steps.

It is too burdensome to try to understand the running time of an algorithm on every specific input, because there are far too many possible inputs, and even very simple algorithms can work in very complex ways. Since algorithms of interest typically run longer on larger inputs, we usually try to understand and quantify certain kinds of "*summary*" running times, *as functions of the input size*. Here are the most commonly considered ones.

Worst-case running time. Computer scientists can be a pessimistic and paranoid bunch. They are concerned with how bad things can possibly be, like how long an algorithm might run before producing an answer—they don't want to be late or go over budget.

The *worst-case* running time of an algorithm is the *maximum* number of steps it can take when given an input of size n. (This would be infinite if some input causes the algorithm to run forever.) Because this maximum varies with n alone, an algorithm's worst-case running time is a *function* of n, which we usually write as $T(n)$ or similar (but any other name is valid too). It is important to understand that this $T(n)$ is a function in the *mathematical* sense; it is typically not an algorithm or piece of code we can "run." Instead, for every possible input size n, there *exists* a corresponding value $T(n)$, which captures important summary information about the algorithm's behavior.

Like QuickSort, an algorithm might run much faster on some inputs than others of the same size. Or, *every* input might be a worst-case input, i.e., the algorithm takes exactly the same number of steps on every input of the same size. When we make a statement about the worst-case running time, we are not saying anything about how the algorithm behaves on non-worst-case inputs, or

[1]Remember that the "size" of an input is essentially the number of symbols, typically bits, that are required to represent (or "write down") the input.

whether such inputs even exist. For example, if we say "the worst-case running time of Algorithm X is $T(n) = n^2/2$," we are merely describing the *slowest* X can possibly run, nothing more. It might be the case that X runs faster on certain inputs, or not. In sum, the worst-case running time is merely an *upper bound* on the number of steps an algorithm takes.

Best-case and "typical"-case running times. If we are feeling more optimistic, we might consider other summary types of running times. The *best-case* running time of an algorithm is the *minimum* number of steps it can take when given an input of size n. For example, QuickSort has a best-case running time of roughly n, because of how fast it runs on an already-sorted input.

All of the above comments about worst-case running times apply analogously to best-case running times. A statement about the best case does say anything about whether an algorithm runs slower on certain inputs; it is merely a *lower bound* on the number of steps. Because best-case running times are so "optimistic," they are usually not too useful, except perhaps as a basis for criticism: if an algorithm's best-case running time is large, then it will be slow no matter what input it is given.

As the name suggests, the "typical"-case running time of an algorithm is how many steps it performs on a "normal" kind of input (of size n). Defining what "normal" means is tricky, and depends on the surrounding application and how inputs are generated. More precisely, if inputs are drawn from some known (or assumed) probability distribution, then we can define the *average-case* running time to be the average (or expected) number of steps the algorithm takes for a *randomly* chosen input from that distribution. Average-case running times are not widely used, because they inherently rely on the questionable assumption that inputs really do come from some specific distribution, and because analyzing average-case behavior can be quite cumbersome.

2 Asymptotic Bounds

In this section, put aside everything you know about algorithms and their running times, including what is written above. The focus here is on *asymptotic bounds for functions*, without any reference to what those functions may mean or correspond to.

Suppose we have some function $f(n)$. This can be any function with any meaning: it could be a worst-case running time (as a function of input size), or a best-case running time, or something completely different, like the amount of *space* an algorithm uses, or even the amount of money in your bank account (as a function of the day). The main point of asymptotic notation is to **bound how a function $f(n)$ "scales" as the argument n grows large**. In particular, asymptotic notation **ignores (and hence says nothing about) constant factors and lower-order terms, and "small" values of n.**

Let's briefly see how the definitions of asymptotic notation do this, and what the strengths and limitations are. For (non-negative) functions $f(n)$ and $g(n)$, the precise definition of "$f(n) = O(g(n))$" is:

> there exist positive constants c, n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

The key features here are:

- "there exists a positive constant $c \ldots f(n) \leq c \cdot g(n)$": this says that $f(n)$ is *upper bounded* by some *constant multiple* of $g(n)$. That is, O-notation "hides," or ignores, constant factors.

Note also that it captures only an *upper bound*: $f(n)$ might actually be *much smaller* than $c \cdot g(n)$, or not. A big-O bound by itself says nothing about which is the case.

- "there exists a positive constant n_0 ... for all $n \geq n_0$": this says that the upper bound holds for all n *above some constant "threshold."* However, it says nothing about the relationship between $f(n)$ and $g(n)$ *below* the threshold; it could be that $f(n)$ greatly exceeds (even a huge multiple of) $g(n)$ in that range. Moreover, the threshold n_0 can be *any constant*—it could be one, or ten, or a billion, or a googolplex—and the notation hides its exact value.

The definition of big-Omega notation, written $f(n) = \Omega(g(n))$, is exactly the same as above, but with a *lower* bound $f(n) \geq c \cdot g(n)$ in place of the upper bound. All of the above comments about big-O bounds apply analogously to big-Omega bounds. The most important point is that a big-Omega bound captures only a *lower bound*.

Finally, if we have matching upper (big-O) and lower (big-Omega) asymptotic bounds on a function, then we can combine them into a "big-Theta" bound. Formally, $f(n) = \Theta(g(n))$ if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ hold. This means that $f(n)$ is "sandwiched" between two positive constant multiples of $g(n)$, for all large enough n. (The hidden constant factors c from the big-O and big-Omega bounds do not have to be the same, and typically are not.)

3 Putting Things Together

Now let's see how these two concepts go so well together, like a delicious PB&J sandwich.

Suppose we have some algorithm and are interested in its running time of some type. For concreteness, let's consider its worst-case running time $T(n)$, though the discussion applies equally well to other notions like best-case and typical-case running times. As explained above, this is a well defined function in the input size. For example, here is a hypothetical possible behavior: for $n = 1$ the worst-case running time is $T(1) = 10$ (and this is the running time for *every* size-1 input), for $n = 2$ it is $T(2) = 98$ (but the algorithm takes time 45 on certain size-2 inputs), for $n = 3$ it is $T(3) = 6$ (and for every $i \leq 6$ there is a size-3 input where the running time is i), etc.

Unfortunately, it is very unwieldy to determine exactly how many steps an algorithm takes in the worst case, especially for larger inputs. While it is a mathematically well defined function for any algorithm, it can be beyond our human abilities to analyze, even for very simple algorithms. In addition, there are many different kinds of computers and processors, so the exact worst-case running time might even vary from one kind of computer to another. We need a way to deal with all this mess.

Applying asymptotics to running times. A better alternative is to focus on more coarse-grained, but still meaningful and useful, measures of how the running time *scales* as the input size grows. Asymptotic notation is exactly the right tool for this! By definition, it deliberately hides/ignores a lot of lower-level details about the function: the hidden constant factor, any lower-order terms, and the "large enough" threshold. And as already discussed, summary running times also hide a lot of information about an algorithm's behavior. Altogether, this makes it much easier to derive useful and meaningful summary statements about an algorithm, because we only need to bound the number of operations to an appropriate degree of accuracy.

This strength can also be a weakness: there is a lot of room to fit "pathological" (but mathematically valid) behavior into statements like "the worst-case running time is $O(n \log n)$." Any of the following could be true:

- the hidden constant could be enormous (too big to tolerate in reality),
- the asymptotic behavior might hold only above a huge threshold (beyond the sizes of our inputs in reality),
- the actual worst-case running time might grow much more slowly than $n \log n$—like $\log n$ or even a constant—because big-O merely represents an upper bound.

If we care about any of these issues in practice, then we need to "look inside" the proof of the asymptotic bound to learn what we can about the hidden quantities. But as an initial method for classifying the "essential" performance of algorithms, asymptotic bounds are very useful.

Exercises. It is important to state and use asymptotic bounds appropriately and correctly. Below are some statements; for each one, try to determine whether it is well formed, or meaningless "gibberish". If it is well formed, determine whether it is true or not. The answers are on the next page.

1. "QUICKSORT's best-case running time is $O(n)$, and its worst-case running time is $\Omega(n^2)$."
2. "$O(n)$ is less than $O(n^2)$," and "$\Theta(n)$ is less than $\Theta(n^2)$."
3. "Algorithm X is $O(n \log n)$."

Try to come up with more examples of meaningless, meaningful but false, and meaningful true statements involving running times and asymptotic bounds.

Here are the answers to the exercises:

1. This statement is both well formed and true. As we have noted, the best-case input for QuickSort is an already-sorted array, on which it runs for about n steps. This is upper bounded by a constant multiple of n, so the best-case running time is $O(n)$. Similarly, on an input array that is far from sorted, QuickSort takes about n^2 steps, so its worst-case running time is at least a positive constant multiple of n^2, i.e., $\Omega(n^2)$.

   We could even strengthen the statement, by replacing both the O and Ω with Θ. Why does the statement remain meaningful and true?

2. This statement is meaningless. Because $O(n)$ is a *class* of many different functions, it is not meaningful to say that it is "less than" or "greater than" something else; we can only compare an *individual* function against another one.

   We could try to repair the statement to say "every $O(n)$ function is less than every $O(n^2)$ function," but this would be *false*. For example, $T(n) = n$ is $O(n)$, whereas $U(n) = 1$ is $O(n^2)$ (remember that big-O merely represents an upper bound). But $T(n)$ is not less than $U(n)$, for any value of $n \geq 1$.

   Noticing that the possible looseness of the big-O upper bound was the problem with the previous statement, we could try to repair it to say "every $\Theta(n)$ function is less than every $\Theta(n^2)$ function." But this would still not be true, because of the unspecified *thresholds* in the bounds. For example, $T(n) = 100n$ is $\Theta(n)$, and $U(n) = n^2$ is $\Theta(n^2)$. But $T(n)$ is not less than $U(n)$ when $n \leq 100$, though it is for all $n > 100$.

   We can repair the above statement to make it true, as follows: "for every $T(n) = \Theta(n)$ and $U(n) = \Theta(n^2)$, we have $T(n) < U(n)$ for all large enough n." Using the definitions, try to prove this! It will suffice to use the hypotheses that $T(n) = O(n)$ and $U(n) = \Omega(n^2)$.

3. To be precise, this statement is meaningless. The only things that can be considered $O(n \log n)$ are *mathematical functions* in the variable n, and Algorithm X is not such a function.

   We can try to repair the statement by saying "Algorithm X's *running time* is $O(n \log n)$." This is OK if the *type* of running time (worst case, best case, etc.) is clear from context; usually worst case is the default, and it always will be in this class. If there is any risk of ambiguity, we can instead say "Algorithm X's *worst-case running time* is $O(n \log n)$" to be fully precise.