Asymptotic Time and Space Complexity Analysis

1 Big-O and its Relatives

1.1 Introducing Big-O

To decide between algorithms (and the data structures that they use), we need to be able to measure the resources (time and space) that they require. We are primarily concerned with how the resources used grow with respect to the size of the input, as generally more input will require more time to process and will take up more space in a data structure. We would also like to measure this in a machine-independent way, so that we can make decisions without having to know the time needed for different operations on the individual machines. This will also prevent us from having to redo the analysis when an algorithm is reimplemented or transferred to a different machine, and enable us to decide which algorithm and structure is appropriate before spending time on implementation.

So, we want to express the amount of time (or space) used as a function of the input size (generally $f : \mathbf{N} \to \mathbf{N}$, usually written as $f(n)$ where n is the input size), and we want to compare the growth of these functions without considering the constant factors that would depend on the individual implementation, machine, or in how much detail we have decided to count the operations.

We can also usually ignore small input sizes, since the resources needed will not be very large anyway.

Essentially, what we need is a version of "$\leq$" that can ignore constant factors and results for small sizes, and characterizes functions by their growth as the size (n) gets large – their *asymptotic complexity*. This system of comparison is known as **Big-O**, and is defined as follows:

for any functions $f : \mathbf{N} \to \mathbf{N}$ and $g : \mathbf{N} \to \mathbf{N}$

$$f(n) \in O(g(n)) \ \text{ if and only if}$$

$$\exists c > 0,\ n_0 \geq 0 \ \text{ such that :} \quad f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

This is also sometimes expressed as: $f(n)$ **is** $O(g(n))$.

1.2 Proving Big-O

We can use the above definition to formally prove that the property holds (or does not hold, depending on the functions). Usually $g(n)$ is a more simply expressed function, but this is not a requirement.

For example, we can prove that $50n \in O(n \log_2 n)$ as follows:

We need to find a $c > 0$ and $n_0 \geq 0$ such that

$$50n \leq c \cdot n \log_2 n \quad \forall n \geq n_0.$$

This is our **goal**, so we do not start with it or derive from it; just write it down and look at it to see how you can prove it. It's important to note that neither c nor n_0 needs to be tight. We're not interested

[1]Copyright ©Dr. Ptricia Evans, Faculty of Computer Science, UNB

in finding an optimal pair, so we can let our proof guide our choice of c and n_0 in a way that simplifies the proof.

As we derive a value for c (which must be a constant **independent** of n), we note down the constraints we need on n in order for our math to work.

Here's one proof that derives working values for c and n_0:

$$50n \leq 1 \cdot n \log_2 n \quad \forall n \geq 2^{50}$$

Now, while $n_0 = 2^{50}$ is a constant, and therefore can legitimately be used to prove big-O, it isn't quite in the spirit of the original idea (ignoring smaller values of n). It also may not be workable when multiple terms are involved. We can instead prove the big-O property this way (note that $\log_2 n \geq 1$ as long as $n \geq 2$):

$$50n \leq 50 \cdot n \log_2 n \quad \forall n \geq 2$$

giving us $c = 50$ and $n_0 = 2$.

For functions with multiple terms, we need to look for a way to eliminate and combine them that is in keeping with the inequality direction that we need. Generally, it's a good idea to start on one side or the other, and simplify while ensuring that we are making the function either larger or smaller (whichever is appropriate).

Consider $f(n) = 3n^3 - 5n^2 + 9n + 2\log_2 n$ and $g(n) = n^3$. To prove that $f(n) \in O(g(n))$, we need to find $c > 0$ and $n_0 \geq 0$ such that

$$3n^3 - 5n^2 + 9n + 2\log_2 n \leq c \cdot n^2 \quad \forall n \geq n_0$$

We can find appropriate values of c and n_0 by using some simple algebraic techniques. We note that removing negative terms will make the expression larger or equal (never smaller as $n \geq 0$), and so will multiplying by additional factors of n as long as $n \geq 1$. So we start on the left-hand side of the goal inequality, and try to make it look like the right-hand side of the goal inequality, remembering that we can multiply any positive term by anything we want as long as the new factors are ≥ 1 for sufficiently large values of n.

$$\begin{aligned} 3n^3 - 5n^2 + 9n + 2\log_2 n &\leq 3n^3 + 9n + 2\log_2 n \quad \forall n \geq 0 \\ &\leq 3n^3 + 9n^3 + 2n^3 \qquad \forall n \geq 1 \\ &= 14n^3 \end{aligned}$$

So $3n^3 - 5n^2 + 9n + 2\log_2 n \in O(n^3)$, using $c = 14$ and $n_0 = 1$ (as witness). This means that if we have an algorithm whose running time is expressed as a function of its input size by $3n^3 - 5n^2 + 9n + 2\log_2 n \in O(n^3)$, we can say that the algorithm runs in $O(n^3)$ time. This is the usual way that algorithms' running times are expressed, since the coefficients are usually not important and also probably not entirely accurate.

What about the reverse? Can we prove that $n^3 \in O(3n^3 - 5n^2 + 9n + 2\log_2 n)$? We would need to find $c > 0$ and $n_0 \geq 0$ such that

$$n^3 \leq c \cdot (3n^3 - 5n^2 + 9n + 2\log_2 n) \quad \forall n \geq n_0$$

Here, since our right-hand function is more complicated, we can simplify things by starting on the right and working from there (working backwards a bit, making sure we are making the expression smaller or equal).

Since c will be positive, we know that for any c,

$$c \cdot (3n^3 - 5n^2 + 9n + 2\log_2 n) \geq c \cdot (3n^3 - 5n^2)$$

so we can drop any lower degree terms that have positive coefficients. The negative coefficient terms are still problematic, though, and require us to use a larger value of n_0.

Note that if $n \geq 5$, $n^3 \geq 5n^2$, so $n^3 - 5n^2 \geq 0$. This means that

$$c \cdot (3n^3 - 5n^2) \;=\; c \cdot (2n^3 + n^3 - 5n^2) \geq c \cdot 2n^3 \qquad \forall n \geq 5$$

So if we chain these inequalities together, we get:

$$n^3 \leq 2n^3 \leq 2n^3 + n^3 - 5n^2 \;=\; 3n^3 - 5n^2 \leq 3n^3 - 5n^2 + 9n + 2\log_2 n \quad \forall n \geq 5$$

So $c = 1$ and $n_0 = 5$ are witness to $n^3 \in O(3n^3 - 5n^2 + 9n + 2\log_2 n)$. We didn't really need to worry about c here, but we might if the coefficient of the lead term (term of largest degree) is less than 2. We could also have used $c = \frac{1}{2}$, but although it is a tighter bound, it is no more correct when it comes to proving that the property holds, so we have picked the option for c that is simpler to prove.

If both functions involve multiple terms, we may need to work from both sides, simplifying them into the same form while making sure the inequalities line up correctly. Make the LHS bigger, make the RHS smaller, and meet in the middle.

For example, we prove that $n^2 + 5n \in O(3n^2 - 2n + 1)$ by noting that

$$LHS \;=\; n^2 + 5n \leq n^2 + 5n^2 = 6n^2 \qquad \forall n \geq 1$$

$$RHS \;=\; 3n^2 - 2n + 1 \geq 3n^2 - 2n \geq 3n^2 - 2n^2 = n^2 \qquad \forall n \geq 1$$

and chaining them together to get

$$n^2 + 5n \leq 6n^2 \;=\; 6 \cdot (3n^2 - 2n^2) \leq 6 \cdot (3n^2 - 2n) \leq 6 \cdot (3n^2 - 2n + 1) \qquad \forall n \geq 1$$

You must ensure that the inequalities chain correctly; writing them all out in a long line at the end and checking that all of them point in the same direction is a very good way to check this. If you instead include one or more inequalities in the wrong direction, the proof is completely compromised. Note, for example, that if we have $a \leq b$ and $b \geq c$ we cannot say anything about the relative values of a and c – they're both no larger than b, but there's no way to compare them.

1.3 Disproving Big-O

Just as we can prove that $f(n) \in O(g(n))$ directly from the property's definition, we can prove that $f(n) \notin O(g(n))$ (when this is the case) by proving the property's logical negation:

$$\forall c > 0,\; n_0 \geq 0,\;\; \exists n \geq n_0 \text{ such that } \; f(n) > c \cdot g(n)$$

For example, consider the functions $3n^2$ and $7n$. To prove that $3n^2 \notin O(7n)$, we need to show that for **all** $c > 0$ and n_0, we can find an $n > n_0$ such that

$$3n^2 \;>\; c \cdot 7n$$

So for what values of n is this true? We need $n > \frac{7c}{3}$, as

$$n > \frac{7c}{3} \quad\Longrightarrow\quad 3n > c \cdot 7 \quad\Longrightarrow\quad 3n^2 > c \cdot 7n \quad \text{if } n \geq 1$$

Note that here it's ok for n to depend on c, since $\exists n \geq n_0$ is within the scope of the $\forall c > 0$ quantifier. When **proving** big-O, we cannot pick c dependent on n, since $\exists c > 0$ is outside the scope of the $\forall n \geq n_0$ quantifier.

1.4 Logarithms

Many algorithms have logarithms in their running time. This is usually due to their splitting of the data, which we will see examples of later in the course. When we use big-O analysis on functions like $\log_2 n$, we tend to leave off the base of the logarithm and merely say that it runs in $O(\log n)$ time. This is because the base of the logarithm is not important when dealing with asymptotic analysis; for any two constant bases a and b,

$$\log_a n \;=\; \frac{\log_b n}{\log_b a}\ ,$$

which enables us to convert between logarithm bases by multiplying by the constant factor $\frac{1}{\log_b a} \;=\; \log_a b$. Thus any constant bases are equivalent under big-O analysis, enabling us to ignore the base entirely. When proving something involving $O(\log n)$ (or any function with a $\log n$ term where the base is not specified), we can use any constant > 1 as the logarithmic base; in these situations, we frequently use base 2. When working with logarithms, the property $\log_2 n \;<\; n$ can also be useful.

However:
The unimportance of the base is only true for logarithms by themselves and logarithmic factors. If a logarithm is being used inside another function, particularly if it is part of an exponent, then the base **is** important as exponentiation will turn a constant factor into a power.

To see this, consider $\log_2 n$ and $\log_4 n$, when used as exponents (e.g. with base 4).

$$4^{\log_2 n} \;=\; (2^2)^{\log_2 n} \;=\; 2^{2\log_2 n} \;=\; (2^{\log_2 n})^2 \;=\; n^2$$

while

$$4^{\log_4 n} \;=\; n$$

and $n^2 \notin O(n)$.

The same problem occurs for comparing different exponentiation bases. For example, $4^n \notin O(2^n)$, by the following proof:

$$4^n \;=\; 2^n \cdot 2^n \;>\; c \cdot 2^n \quad \forall n > \log_2 c$$

So for any c and n_0, we can pick an $n \;> n_0$ (e.g. $n = \max\{\log_2 c, n_0\} + 1$) that will produce $4^n > c \cdot 2^n$, and thus $4^n \notin O(2^n)$.

1.5 Relatives of Big-O: Ω and Θ

When we say that $f(n) \in O(g(n))$, the function $g(n)$ is essentially an asymptotic upper bound on $f(n)$. To be precise about the growth of $f(n)$, we would like to also have an asymptotic lower bound or version of "$\geq$", which we call Big-Omega (Ω).

We say $f(n) \in \Omega(g(n))$ if and only if:

$$\exists c > 0, n_0 \geq 0 \text{ such that } f(n) \geq c \cdot g(n) \qquad \forall n \geq n_0$$

Since all that has changed from the big-O definition is the direction of the inequality, we can prove that $f(n) \in \Omega(g(n))$ if and only if $g(n) \in O(f(n))$; we do this by noting that if c and n_0 are witness to $g(n) \in O(f(n))$, then $\frac{1}{c}$ and n_0 are witness to $f(n) \in \Omega(g(n))$, and vice versa.

When using Ω, we need to remember that we are not just asymptotically bounding the function $f(n)$; we are really trying to bound the running time of the algorithm (or space needed for the data structure). Usually we are making some shortcuts in determining $f(n)$, ignoring constant factors, and probably also taking a worst-case perspective. If the analysis that produces $f(n)$ is loose, then it will affect the further analysis that we can use on $f(n)$.

Specifically, if we have only shown that

$$\text{running time} \quad \leq \quad f(n)$$

then finding $f(n) \in \Omega(g(n))$ will **not** give us an asymptotic lower bound on the running time, since a lower bound of an upper bound has an unknown relationship to the original thing. If we want to find both upper and lower bounds, we need to have $f_1(n)$ and $f_2(n)$ such that

$$f_1(n) \quad \leq \quad \text{running time} \quad \leq \quad f_2(n)$$

and then find an asymptotic lower bound (Ω) for $f_1(n)$, and an asymptotic upper bound (O) for $f_2(n)$. This approach is usually what is done when analysing algorithms, since determining the actual running time function $f(n)$ can often be as complicated and time-consuming to do as implementing the algorithm in the first place, and one key reason for analysing algorithms is so that the best one can be chosen *prior to having to implement it*, so that only the best algorithm needs to be implemented.

If the upper and lower bounds are asymptotically the same, i.e. we have both $\in O(g(n))$ and $\in \Omega(g(n))$, then we can combine these as $\in \Theta(g(n))$. So for something like $5n^2 + 3\log_2 n - 6$, it is more precise to state that it is $\in \Theta(n^2)$ instead of $\in O(n^2)$.

1.6 Why Big? Introducing little-o and little-omega

We refer to $O(g(n))$ as Big-O, and to $\Omega(g(n))$ as Big-Omega. There are also "little" versions of these, $o(g(n))$ and $\omega(g(n))$, that are the asymptotic equivalents of "$<$" and "$>$" respectively.

We consider $f(n) \in o(g(n))$ if $g(n)$ is an upper bound on f that is not attained. Unfortunately this cannot be done just by using $<$ instead of $\leq$ in the proof, since you can easily turn $\leq$ into $<$ just by increasing the constant c. Instead, we need to show that whatever c is, we can increase n and keep the result below the upper bound. So $f(n) \in o(g(n))$ if and only if $\forall c > 0 \;\; \exists n_0 > 0$ such that $f(n) > c \cdot g(n) \; \forall n \geq n_0$. Thus f grows more slowly than g.

You won't see little-o used very much; it can show up in very complicated time functions, but it's more commonly used to compare functions and state goals. For example, mathematically, an algorithm that runs in **sublinear** time can be represented as running in $o(n)$ time.

There is a similar lower-bound that is not attained: $f(n) \in \omega(g(n))$ if and only if $\forall c > 0 \;\; \exists n_0 > 0$ such that $f(n) < c \cdot g(n) \; \forall n \geq n_0$. For example, $\omega(n^k)$, where k is any constant, is one way of representing **superpolynomial** functions, so that we can state precisely what it means for an algorithm not to run in polynomial time.

$\Theta(g(n))$ is the asymptotic equivalent of "=", so it has no "little" version.

1.7 Why Asymptotic?

These analyses are considered to be *asymptotic* because we are considering the behaviour of the functions (and, ultimately, the algorithms) as the input size becomes large. More mathematical discussions of asymptotic complexity use alternative definitions involving limits and absolute values, *e.g.*, $f(x) \in O(g(x)$ if and only if

$$\lim_{x\to\infty} \left| \frac{f(x)}{g(x)} \right| < \infty$$

However, this uses more math than you need, and also ignores important restrictions due to the source of the functions. Since we are counting sizes and units (time units, space units, operations of specific types), the functions are restricted to positive integers only, eliminating the need to consider absolute values and continuous functions.

The continuous definition of big-O has fallen into disuse for these reasons. Similar definitions are still useful for $o()$ and $\omega()$, however, especially when using them to discuss error analyses.

2 Analysing Algorithms

Now that we know the relevant math, we need to use it to analyse algorithms. We should always keep in mind the context we're working in, in that our goal is to be able to assess an algorithm's resource needs in a way that enables us to make decisions about what algorithm to use for a particular task (or, sometimes, whether the task is feasible at all). We can use the context to ensure that we have useable results while avoiding doing unnecessary work.

2.1 What Cases are We Counting?

It's important to realize that these asymptotic analyses such as big-O are used to describe the behaviour of an algorithm once you've decided which behaviour to analyse. Most commonly, *worst case* complexity is analysed. However, you can apply the exact same definitions to *average case* complexity as well (though average case complexity requires a probability distribution on the input and usually a significant amount of statistics).

First you determine what behaviour is being considered, then you analyse it. An upper bound on the worst case is particularly useful since it is also an upper bound on *all* cases, but you can also find a lower bound on the worst case, an upper bound on the average (or best) case, and so on.

2.2 Making Strong Statements to Enable Algorithm Comparison

If we want to be able to compare algorithms from different sources, without having to redo the analyses ourselves, it helps if the strongest statements possible has been made (and proven) about the algorithms' time (and space) complexities.

For example, if algorithm A is stated to run in $O(n^2)$ time, and algorithm B is stated to run in $O(n \log n)$ time, we technically do not know which one is the fastest, since we do not know if A really attains its stated upper bound of $O(n^2)$ time. In practice we would go with B, since its stated upper bound is smaller, but we still cannot say that it's faster. Many a good algorithm can be, and has been, diminished by an excessively loose analysis that produces an overly high and unattained upper bound.

Usually, to be sure about our bounds and how the algorithm is being presented, tight bounds with a Θ result are the most desirable. Usually these are proven by the means discussed previously, showing that

$$c_1 \cdot g(n) \leq f_1(n) \leq \text{ running time } \leq f_2(n) \leq c_2 \cdot g(n)$$

Various shortcuts are used to find f_1 and f_2, depending on the characteristics of the algorithm. As long as the bounds match and the analysis is accurate, the operation counting itself can be quite loose.

2.3 Using Big-O, Big-Omega, and Theta

The tools of asymptotic analysis (as previously discussed) are defined using functions, as if we are given two functions f and g and then have to prove mathematically whether $f(n) \in O(g(n))$. However, in practice, this is seldom the case. Nor is it common to have a precise $f(n)$ then simplify it to an appropriate $g(n)$ such that $f(n) \in \Theta(g(n))$, though it may often appear as though we do.

Why? Because in practice, we do not usually have $f(n)$ at all – we do not have the precise running time function for the algorithm, nor do we need it. What we have instead is either a pair of functions (f_1 and f_2 as referred to above) or simply an implied function – the idea that there is some function that describes the running time of the algorithm in terms of its input size. We do not know what this function is, and it is usually a lot of work to figure it out so we would rather not have to. That's fine, since all we need is to find how it behaves asymptotically. We do not need the precise function itself.

There are also parts of the function that come from implementation details, such as whether particular compound-looking instructions count as one step or two (or three), whether a loop returns to the loop control line one last time before exiting, and so on. These are affected by language choice, compilers, optimization steps, use of registers, and the machine's instruction set. We would prefer our analysis to be independent of these, since we want to be able to compare algorithms at a higher level; we also do not want to have to do that much work, since a key purpose of analysing algorithms is so we can decide which algorithm is best without having to implement and test them, and figuring out the precise $f(n)$ for an algorithm would be as much work or more.

We need to have enough information to be able to compare the algorithm to other algorithms, so we can ignore information that doesn't affect those results – either by discarding it, or by not figuring it out in the first place.

We usually do not need to figure out what $f(n)$ is, because those details would only affect the coefficients and lower-order terms, which we would discard anyway. So, instead of figuring $f(n)$ out, we leave it as implied: $f(n)$ is the running time of the algorithm, whatever that may be, and we find upper and lower bounds for it and report those.

This approach may be familiar to you from calculus, where it is used to find the limits of a complicated looking function, such as a sum or integral.[2] If we have f, g_1, g_2 such that $g_1(n) \leq f(n) \leq g_2(n)$, and $\lim_{n \to A} g_1(n) = \lim_{n \to A} g_2(n)$, then $\lim_{n \to A} f(n)$ must be the same, no matter how messy or difficult to calculate it might be on its own. We take the same approach with analysing algorithms, so that if we find that an algorithm is $O(g(n))$ and $\Omega(g(n))$, we report it as $\Theta(g(n))$. We can simplify rounding up, and simplify again rounding down, and if the results are within a constant factor of each other then we know the algorithm behaves that way asymptotically – even if we never figure out the actual running time function for the algorithm.

[2]Known as the squeeze theorem or the policeman principle.

Consider this example:

Algorithm 1: Example Algorithm

```
Input  : array $A[1..n]$ of integers
Output: sum-of-products $S$

$S \leftarrow 0$
for $i = 1$ to $n - 1$ do
    for $j = i + 1$ to $n$ do
        $P \leftarrow 1$
        for $k = i$ to $j$ do
            $P \leftarrow P \cdot A[k]$
        end
        $S \leftarrow S + P$
    end
end
return $S$
```

Instead of coming up with a nested set of sums and solving them, we find upper and lower bounds on the running time.

Upper bound:
The outer loop (on i) iterates exactly $n-1$ times. For each of these iterations, the middle loop (on j) iterates at most $n-1$ times. And for each of these iterations on j, the innermost loop (on k) iterates at most n times.

The rest of the algorithm contributes at most some constant factor to the running time, so the algorithm as a whole runs in $O((n-1) \cdot (n-1) \cdot n) \;=\; O(n^3)$ time.

Once we have an upper bound, we look for a lower bound to match it. (If you have great difficulty doing this, consider that your upper bound may be too loose and try to tighten it.) For this algorithm, in order to get a lot of iterations of the middle and inner loops, we need to restrict how many iterations we're counting of the outer loop. Since we're finding a lower bound, it's okay not to count everything, as long as we count enough to get as asymptotically large a lower bound as we can. Unlike for an upper bound, we cannot count combinations of iterations that do not exist in the algorithm; we have to use the least number of iterations that could happen, given whatever restrictions we've selected.

Lower bound:
Consider i such that $1 \le i \le \frac{n}{2}$, and j such that $\frac{3n}{4} < j \le n$. All such i, j will have $1 \le i < j \le n$, consistent with the algorithm.

There are at least $\lfloor \frac{n}{2} \rfloor$ such iterations of i, and for each of these there are at least $\lfloor \frac{n}{4} \rfloor$ such iterations of j.

For each of these iterations of j, there are at least $\min(j-i) \;=\; (\min j) - (\max i) \;=\; \frac{3n}{4} - \frac{n}{2} \;=\; \frac{n}{4}$ iterations of the innermost loop.

So, the algorithm runs in $\Omega(\lfloor \frac{n}{2} \rfloor \cdot \lfloor \frac{n}{4} \rfloor \cdot \frac{n}{4}) \;=\; \Omega(n^3)$ time.

Since the algorithm runs in $O(n^3)$ and $\Omega(n^3)$ time, it runs in $\Theta(n^3)$ time.

It may look as though the restrictions on i and j came out of nowhere. They come from experimentation and consideration of the possible problems. If we follow the derivation without those restrictions, at some point we end up substituting too small a value to get the minimum number of iterations, such as when we're looking at the min $(j - i)$ value. Once we realize where the problem is (that j can be too small and that i can be too large to give us a minimum count that's a constant fraction of the maximum), we can go back up to the start and put restrictions on. Proofs are organized to make them easier to follow, but (like algorithms and programs) they're often not invented in the order presented. We either think ahead and put in restrictions and premises that we think we're going to need later, or we go back and add conditions that it turned out we needed. We only restrict the number of iterations we count for this algorithm because we want a good minimum later, there's no other purpose for it. We also could have picked other similar restrictions, such as $1 \leq i \leq \frac{n}{3}$ and $\frac{2n}{3} \leq j \leq n$, since these would also be consistent with the algorithm (make sure you check the loops themselves since these will change from algorithm to algorithm) and allow sufficient minimum iterations for all loops. The constant factor at the end will be different, but that will be discarded anyway.

2.4 Case Analysis

The previous analysis is for an algorithm whose running time is entirely determined by the size of the input. However, some algorithms' running times are affected by the contents of the input as well. For these, the implied function $f(n)$ is based not only on the algorithm but on what input cases we are considering for it.

Commonly we consider *worst-case* time complexity, the longest that any input case could make the algorithm take. An upper bound on the worst case is an upper bound on all cases, so if an algorithm takes worst-case $O(f(n))$ time then we can refer to it simply as $O(f(n))$ time. For the lower bound, a lower bound on any specific case is also lower bound on the worst case.

We also sometimes consider best case and average case. *Best case* tends not to be considered unless comparing algorithms that otherwise have equal standing, since it's easy to have a linear-time special case hardcoded into a program. Good best-case performance is often considered to be more of a feature of certain implementations, to have noticed that a particular input case is easier to deal with and the appropriate shortcuts taken, or to notice when the necessary computation has been done so that the algorithm can stop.

Average case is more complicated and requires more information. It can provide a more accurate assessment of the time to be taken for typical or common input, but in order to know what is common, you need to know what the source of the input is and the probability distribution of the input. Average case requires you to calculate

$$\sum_{\text{all inputs } x} Pr[x] \cdot Time(\text{input } x)$$

which has three key drawbacks:

- you need to know what $Pr[x]$ is for all inputs x
- the calculation is often very complicated
- the results only hold if the probability distribution stays the same, so if you reuse the algorithm for a different set of data, or in a different setting, you may need to redo the entire calculation

So, while it can be informative with respect to actual running times for a specific application, it's often not feasible to have an average-case result, especially since we can't transfer it from application to application. Many considerations of "averages" assume that all possible inputs are equally likely, even in the complete absence of any knowledge as to whether this is reasonable; this approach should not be emulated.

For randomized algorithms, the *expected time* can also be considered. Expected time is also an average, but it is the average of all possible runs of the algorithm on the same input, so it calculates

$$\sum_{\text{all sequences of random choices } y} Pr[y] \cdot Time(\text{input } x \text{ with random choices } y)$$

and can be calculated for any input case. Since the distribution of the random choices are under the algorithm's control, the probabilities can be determined and are tolerant of changes in the source of the data and how the algorithm is used. For example, for randomized quicksort we often refer to worst-case expected time complexity, and find an upper bound on it.

Whatever case we consider, this becomes part of the implied function that we are analysing. So when we say that an algorithm's worst-case time complexity is $\Theta(f(n))$, we are saying that the way the algorithm's running time grows in the worst case is asymptotically equivalent to $f(n)$. Just as we do not need to know the algorithm's precise running time, we also do not need to know its actual worst case (it may have many, or ones that produce the same asymptotic running time). We can limit our work to what is needed to prove the result.

2.5 Worst-Case Example: Bubble Sort

Though this may initially seem strange, we usually do not start a worst-case analysis by trying to figure out what case makes the algorithm take the longest. As with figuring out the algorithm's precise running time function, proving a worst case convincingly is often far more work than what is needed to establish an asymptotic worst-case running time. Instead, we examine the algorithm to determine a theoretical maximum on how long it could take, use that as an upper bound, and then use a specific case to show that the upper bound is attained. If we have difficulty finding such a case, we should consider going back to working on the upper bound, which may be too loose.

Algorithm 2: Bubble Sort

```
Input  : array $A[1..n]$ of integers
Output: sorted array $A$

$continue$ ← True
while $continue$ do
    $continue$ ← False
    for $i = 1$ to $n - 1$ do
        if $A[i] > A[i+1]$ then
            $temp \leftarrow A[i]$
            $A[i] \leftarrow A[i+1]$
            $A[i+1] \leftarrow temp$
            $continue$ ← True
        end
    end
end
```

Consider Bubble Sort as an example, where the specific input case can affect how many iterations the while loop has. One case would be if the array is already sorted; in that case no elements are swapped, so the while loop iterates only once, for a $\Theta(n)$ running time. Since the algorithm must execute the while loop at least once, this is thus a best case.

For the worst case, we need to figure out the largest number of times (within a constant factor) that the while loop will iterate, since each iteration of the while loop produces $n - 1$ iterations of the for loop.

Upper bound:
We figure out the upper bound by considering what the algorithm does on any input. During execution, we can consider the values in the array in two parts: the sorted values at the end, and the remaining unsorted values. Each iteration of the while loop will swap the largest remaining value with all unsorted values that are after it, moving it to its correct position in the array. It will remain in that position on all future iterations, because all values after it are greater than or equal to it, and all values before it are less than or equal to it. Thus after iteration j of the while loop, the jth largest value will be in its correct position of

the array and will stay there.

There are n values in the array, and the smallest value will be in its correct position once the others have been correctly sorted. So after the while loop has iterated $n-1$ times the array must be correctly sorted, and one further iteration will verify this, leave *continue* set to false, and the algorithm will terminate.

The rest of the algorithm contributes at most some constant factor, so the worst-case running time of bubble sort is $O(n \cdot (n-1)) \;=\; O(n^2)$.

Note that we do not know if this upper bound is attained; we have worked out a theoretical upper bound based on an analysis of the workings of the algorithm. We do not have a specific case yet that has this performance. In order to show that this bound is tight, and get a matching lower bound, we need a specific case that produces this running time, or at least a constant fraction of it.

Lower bound:
Consider the case where the input is in reverse sorted order.
For such input, after each iteration of the while loop, the remaining unsorted values are still in reverse sorted order, with the largest remaining value first. Each while loop iteration will compare this value the other unsorted values in turn, gradually shifting it to its correct place.
Thus we know that, on reverse sorted values, each iteration of the while loop (other than the last) moves exactly one value up, and the remaining unsorted values down.

Since the values are in reverse sorted order, half of them ($\lfloor \frac{n}{2} \rfloor$) are before the midpoint and need to be after the midpoint; each of these values need to be moved up, which we have determined will require separate iterations of the while loop. Therefore the while loop must iterate at least $\lfloor \frac{n}{2} \rfloor$ times. Each such iteration will produce $n-1$ iterations of the for loop, so the algorithm runs in $\Omega(\lfloor \frac{n}{2} \rfloor \cdot (n-1)) \;=\; \Omega(n^2)$ time for this case, giving us a lower bound on the worst case.

Therefore bubble sort runs in $\Theta(n^2)$ worst-case time.

As a byproduct of this proof, we have also determined an asymptotically worst case. This is not necessarily the worst in terms of total time, nor do we need to get into how often the body of the if statement will execute, because that only affects the constant factors, which we are not going to keep anyway. We also did not need to show that the while loop iterates n times for this case (though we could have done so), because a constant fraction of the maximum is sufficient to get this asymptotic result.

3 Holistic Analysis

For some algorithms, or parts of algorithms, it makes more sense to consider how often certain things happen over the entire execution of the algorithm, instead of breaking it down loop-by-loop in order to multiply. This is a good approach to consider if the amount of work (such as number of iterations) is variable in a way that makes the maxima in different parts incompatible. If getting more iterations for one part of the execution means fewer iterations for a different part, we should try considering what happens overall.

For example, consider the running time of this Merge algorithm, which merges two sorted lists (often part of MergeSort).

Algorithm 3: Merge

```
Input  : sorted lists $A$ and $B$ of comparable items
Output: sorted list $S$ of items

$S \leftarrow$ empty
while $A$ is not empty or $B$ is not empty do
    while $A$ is not empty and ($B$ is empty or $A.head \leq B.head$) do
        remove $A.head$ from $A$ and append it to $S$
    end
    while $B$ is not empty and ($A$ is empty or $B.head < A.head$) do
        remove $B.head$ from $B$ and append it to $S$
    end
end
return $S$
```

If we consider the different loops involved, the outer loop could iterate up to $\frac{n}{2}$ times (where n is the total number of items). Each of the two inner loops (first moving from A to S, then moving from B to S) could themselves iterate up to n times; even if the items are split equally between the two input lists, the loops could each iterate up to $\frac{n}{2}$ times. Multiplying the results (since the two inner loops are nested inside the outer one) would give us a $O(n^2)$ running time.

Well, that's all true. But it isn't very precise.

As a worst case analysis, the foregoing neglects to consider whether the upper bounds on the individual loops are compatible. They're not. The outer loop can indeed iterate up to $\frac{n}{2}$ times, but only if the inner loops iterate once each for every outer iteration. Additionally, each of the inner loops cannot always be at their maximum, for each outer iteration – they'll run out of items to move.

Instead, we should consider the total work that the algorithm does over its entire execution, with particular focus on how many times an item can be removed from an input list and appended to the output list.

Analysis:
Each time a loop iterates, one item is removed from an input list and appended to the output list. All other work is constant, so for n items the algorithm as a whole runs in $\Theta(n)$ time.