Complexity Analysis (Big O)

Analyzing Data Structure Performance

- As we learn about various data structures in this course, we'll want a way to *compare* these data structures to each other. Specifically, we'll want to be able to make an *informed choice* between the different data structures we might use to solve each particular problem we're working on.

- Most of the time, we'll want to be able to compare data structures in a ***platform-independent*** way. In other words, we want to be able to compare data structures in a way that doesn't depend on hardware, operating system, programming language, etc.

- The reason we want to approach the comparison of data structures in this way is because data structures themselves are platform independent. Indeed, for many of the problems you'll work on, you won't be able to predict exactly what kinds of platforms your data structures will be deployed on, so you'll want to factor platform-specific things out of your design process.

- To help make platform-independent comparisons of data structures, we'll use a tool called ***complexity analysis***.
    - You might have also heard complexity analysis referred to as "***Big O***."

- Complexity analysis will specifically allow us to assess a data structure or algorithm's resource usage (i.e. runtime and memory consumption) in an abstract way that's decoupled from platform-specific factors.

- In particular, using complexity analysis, we'll be able to better understand how a data structure or algorithm's resource usage *grows* as the size of the *input* to that data structure or algorithm grows. We refer to this input size as ***n***.

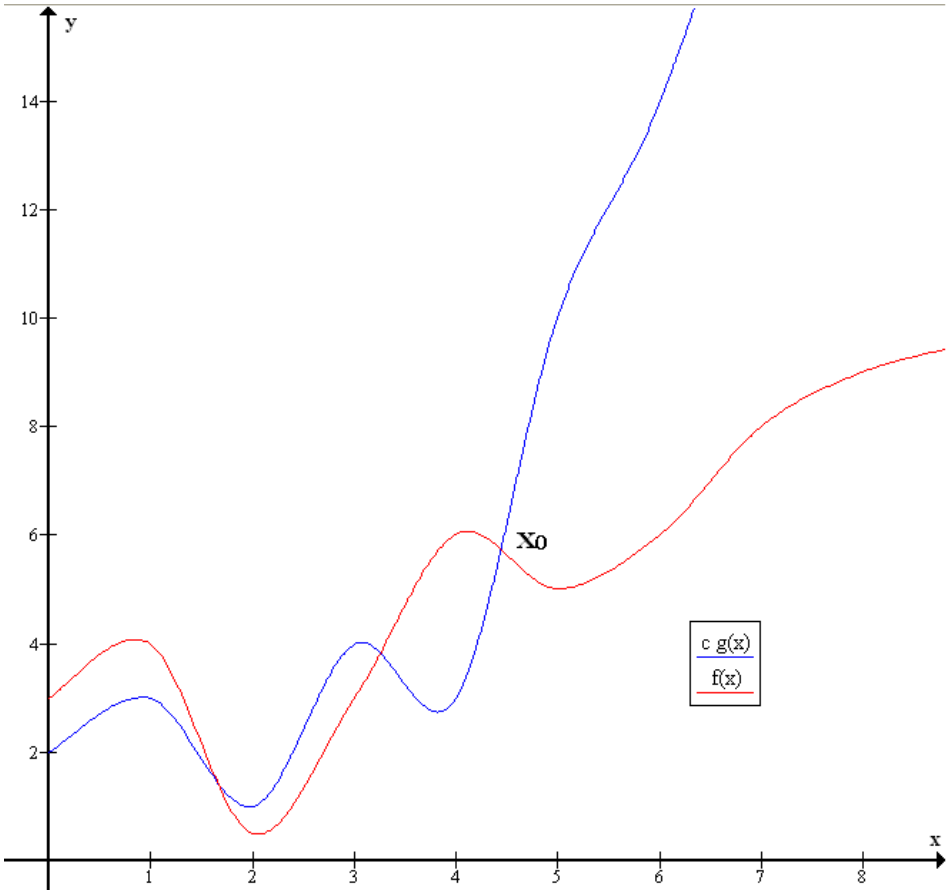
- Here, when we talk about a data structure's input, we typically mean the size of the collection being stored in the data structure. Thus, in this situation, *n* is the number of elements in that collection.

Big O notation

- When using complexity analysis to assess a data structure or algorithm's performance, we will use ***big O notation***.

- Big O notation is a tool for characterizing a function in terms of its growth rate. Specifically, big O notation is used to indicate an *upper bound* on that function's growth rate. This upper bound is known as the ***growth order*** of the function (hence the "O" in "big O") and is itself typically specified as a mathematical function.

- Formally, we say that a function *g(x)* provides an upper bound (i.e. specifies a growth order) for another function *f(x)* if there is some value x_0 beyond which *g(x)* > *f(x)* for every $x > x_0$. For example, in the image below, the function *g(x)* (the blue line) provides an upper bound on *f(x)* (the red line).



- In this case, we can say that $g(x) \in O(f(x))$. Or, more informally $g(x)$ is $O(f(x))$.

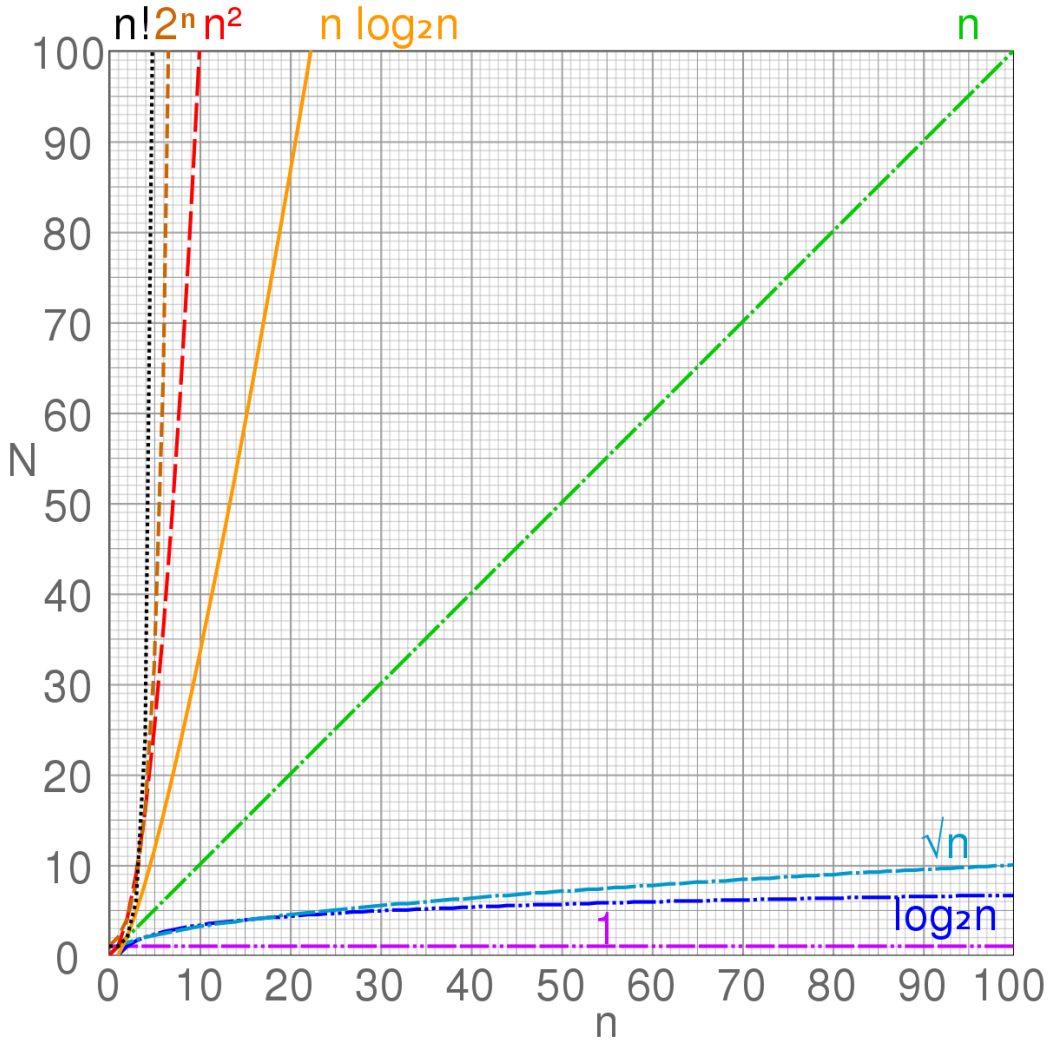
- When we want to assess a data structure or algorithm's resource usage using complexity analysis, we'll specifically compute a growth order for its runtime or memory usage as a function of the input size n.

- Importantly, the notion of a growth order provides a description of a function in the limit, as its input approaches ∞. In the context of characterizing the resource usage of data structures and algorithms, this property of growth orders allows us to ignore one-time costs associated with startup/initialization and to focus on the data structure or algorithm's long-run behavior, which is typically what we'll care most about.

- As we work on assessing data structures and algorithms via complexity analysis, we'll find that there are particular growth order functions we'll encounter over and over. Some of these growth order functions and their common names are:
    - $O(1)$ – constant complexity
    - $O(\log n)$ – log-n complexity
    - $O(\sqrt{n})$ – root-n complexity
    - $O(n)$ – linear complexity
    - $O(n \log n)$ – n-log-n complexity
    - $O(n^2)$ – quadratic complexity
    - $O(n^3)$ – cubic complexity
    - $O(2^n)$ – exponential complexity
    - $O(n!)$ – factorial complexity

- Note that the common growth order functions listed above are ordered from slowest growing ($O(1)$) to fastest growing ($O(n!)$).

- The plot below depicts these common growth order functions graphically:



By Cmglee on Wikipedia, CC BY-SA 4.0

Computing runtime complexity from code

- Now that we know what complexity analysis is, we want to start applying it to understand how our data structures and algorithms perform. Let's start by exploring how to read a piece of code and determine its ***runtime complexity***, that is, the growth order of the code's runtime, in terms of the size of its input.

- We'll start by exploring the following simple piece of code, which just calculates the sum of an array of values:

```
sum = 0;
for (i = 0; i < n; i++) {
     sum += array[i];
}
return sum;
```

- Importantly, the size of the array over which we're computing the sum here is *n*.

- Let's begin our assessment of this code by trying to calculate the exact amount of time it will take to run. We can do this by breaking the code down into smaller parts and first focusing on these individually.

- Let's first look at the very first instruction:

```
sum = 0;
```

- What can we say about how long this instruction will take to run? The instruction itself is very simple. We can't say much for certain about this instruction, though, since its exact will really depend on characteristics of the platform on which it's being run (e.g. processor speed, number of registers, compiler implementation, etc.). We do know, however, that the amount of time this instruction will take to run *does not* depend on the size *n* of the input array. So, for now, let's just say that it will take some constant amount of time t_1 to run.

- Next, let's look at the for loop. For now, let's just focus on the amount of time it'll take to run a *single iteration* of the loop. The following are (approximately) the instructions that will be run on every iteration of the loop:

```
i < n;
sum += array[i];
i++;
```

- Again, the main question we want to ask here is: Does the *amount of time* it will take to run any of these instructions depend on the size of the input (i.e. *n*)?

- In this case, again, the amount of time it will take to run each of these instructions will depend on several platform-dependent factors, but for none of them will it depend on *n*, the size of the input array.

    - Make sure you understand why this is so.

- So, again, let's just say for now that the combination of these instructions will take some constant amount of time t_2 to run.

- Importantly, though, the loop as a whole will run for exactly *n* iterations. Thus, the *total* time it will take to run the entire loop will be $n * t_2$.

- Finally, again, the amount of time it will take to execute the return statement does not depend on the size of the input array, so we'll just say that it will take a constant amount of time t_3 to run.

- Thus, by our analysis here, the total amount of time it will take to execute this code is:

  $t_1 + n * t_2 + t_3$

- Again, this number is a representation of the *exact* amount of time it will take to execute this code. However, we want to know what the runtime complexity of this code is, not its exact runtime, since the latter depends on platform-specific factors.

- We can figure out the runtime complexity, then, by isolating the platform-independent parts of the exact runtime we computed here. In other words, we want to isolate the parts of this computed runtime that depend *only* on the size of the input array, since runtime complexity is expressed as a growth order function on the input size *n*.

- As we described above, each of the constant time factors t_1, t_2, and t_3 depend on the specific platform on which we're running. Thus, in expressing the runtime complexity of the code in question here, we will *completely ignore* those factors.

- Indeed, the only part of our computed exact runtime that depends on the input size *n* is the factor of *n* that arises from the number of times the loop is executed. Thus, we can say that the runtime complexity of this piece of code is *O(n)*.

- In other words, the computational complexity of this piece of code depends entirely on the number of times the loop is executed. In fact, this is broadly true.

- In particular, a function's loop structure is one of the two main drivers of that function's runtime complexity. The other main driver is recursion (which we won't focus on for now).
    - Both of these factors are mechanisms by which a set of instructions is repeated a variable number of times.

- Indeed, as we'll see next, it is possible to quickly assess the runtime complexity of a function by examining its loop structure.

Runtime complexities of common loop structures

- An easy method for determining a function's runtime complexity is by simply counting/computing the number of times that function's innermost loop executes. Again, this works because loops (and code repeated a variable number of times in general) are the primary factor in determining runtime complexity.

- Indeed, there are a number of common loop signatures that correspond to a specific growth order function. We'll review some of these here. By doing so, the goal is to help enable you to be able to determine a function's runtime complexity with a quick reading of its code.

- We'll start with a simple loop structure that matches the one we explored in the array sum code above:

```
for (i = 0; i < n; i++) {
  ...
}
```

- Again, as we explored above, the code inside this loop will be executed *n* times. Thus, assuming the code within the loop itself runs in a constant amount of time (i.e. an amount of time that doesn't depend on *n*), the overall loop will have runtime complexity *O(n)*.

- Another simple loop structure you might see changes the termination criteria slightly:

```
for (i = 0; i * i < n; i++) {
  ...
}
```

- In this loop, we specifically terminate when i^2 becomes greater than n. In other words, this loop will execute $\sqrt{n}$ times and thus this loop has runtime complexity $O(\sqrt{n})$ (again, assuming the code within the loop runs in constant time).

- There are a few important types of loops that can primarily be understood as *halving* the remaining size of the problem at each iteration. This can happen either by starting at the beginning of the input or at the end of the input and jumping forward or backwards in steps whose size doubles at each iteration of the loop, i.e.:

```
for (i = 1; i < n; i *= 2) {
  ...
}
```

or

```
for (i = n - 1; i > 0; i /= 2) {
  ...
}
```

- Whenever we see this kind of behavior, where the remaining size of the problem is halved at each iteration, the associated runtime complexity is $O(\log n)$. In particular, the number of times we can halve a number n before reaching zero is $\log n$.
    - This kind of behavior can result from loop structures like the ones above. However, it most commonly arises through recursion.

Runtime complexity of nested loops

- Often, we'll want to determine the runtime complexity of a function that involves *nested* loops. We'll be able to do this in fundamentally the same way as we determined the runtime complexity for simple loop structures, i.e. by simply counting/computing the number of times the innermost loop executes.

- For example, the code within the innermost loop of the following nested loop structure will execute n^2 total times. Thus, the overall complexity of this nested structure, assuming the code in the innermost loop runs in constant time, is $O(n^2)$:

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    ...
  }
}
```

- Indeed, a simple rule we can follow here is that if we have a function with a nested loop structure, we can calculate the function's overall runtime complexity by multiplying the individual runtime complexities of the separate loops (isolated from each other).

- In the example above, the *i* loop itself has runtime complexity *O(n)*, as does the *j* loop. Thus, the overall complexity that results by nesting the *i* loop within the *j* loop is $O(n * n) = O(n^2)$.

- This works for any kinds of nested loops. For example the loop structure below nests an *O(log n)* loop within an *O(n)* loop. Thus, the overall complexity of the entire nested structure is *O(n log n)*:

```
for (i = 0; i < n; i++) {
  for (j = n - 1; j > 0; j /= 2) {
    ...
  }
}
```

- This also works for loops nested to any level. If we had three, four, five, or 100 levels of nesting, we could compute the overall runtime complexity by simply multiplying the individual complexities of each loop in isolation.

- For example, the following structure nests three *O(n)* loops within each other and thus has overall runtime complexity $O(n^3)$:

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
      ...
    }
  }
}
```

- One special kind of nested loop structure looks like this:

```
for (i = 0; i < n; i++) {
  for (j = 0; j < i; j++) {
    ...
  }
}
```

- What do you think is the runtime complexity of this loop structure? We can figure it out by again simply counting/computing the number of times the code in the innermost loop executes (again, assuming this innermost code runs in constant time).

- In particular, let's just start to count the number of times the code in the innermost *j* loop runs on each iteration of the outer *i* loop:
  - How many times does the *j* loop iterate when *i* = 0? Well, 0 times.
  - What about when *i* = 1? Here, the *j* loop iterates 1 time.
  - When *i* = 2, then, the *j* loop iterates 2 times.
  - When *i* = 3, the *j* loop iterates 3 times.
  - And so forth…

- We can see a pattern emerging here, and understanding this pattern, we can see that if we wanted to count the total number of times the code within the innermost *j* loop executed, we could do so by computing the following sum:

$$0 + 1 + 2 + 3 + \ldots + n - 1 = \sum_{i=0}^{n-1} i$$

- You might remember from your discrete math course that this sum can be computed as:

$$\sum_{i=0}^{n-1} i = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$$

- In other words, the total number of times the innermost *j* loop executes is exactly $\frac{1}{2}n^2 - \frac{1}{2}n$.

- In order to express this number as a growth order function (i.e. to describe the runtime complexity of the entire loop), we first drop the multiplicative factor of $\frac{1}{2}$, since this value is a constant.
    - As we saw above, we drop all constant values from expressions of computational complexity.

- In addition, for reasons we'll see in just a second, we'll also drop the smaller of the two additive terms from this expression, i.e. n. This leaves us with an overall runtime complexity of $O(n^2)$ for this loop structure.

Dominant components of growth order functions

- In the last example above, we simply dropped the *n* term (along with the constant factor of $\frac{1}{2}$) from the expression $\frac{1}{2}n^2 - \frac{1}{2}n$ in order to arrive at a final overall runtime complexity of $O(n^2)$. Why did we do this?

- The reason has to do with the notion of ***dominance***. In a mathematical function, we say that one term *f(n)* dominates another term *g(n)* if

  $\exists n_0 : \forall n > n_0,\ f(n) > g(n)$

- In other words, if beyond a certain point *f(n)* is always greater than *g(n)*, then *f(n)* dominates *g(n)***.**

- When computing a growth order function (i.e. a complexity function), we simply drop all terms other than the dominant term. This is why $\frac{1}{2}n^2 - \frac{1}{2}n$ in or example above became simply $O(n^2)$. We just dropped everything but the dominant term (i.e. n^2).

- We can do this because as a function's input *n* goes to ∞ (which is what we're focusing on here), the overall value of that function receives negligible contributions from all but the dominant term.

- For example, just compare the values of n and n^2 for some large n like n = 1,000,000. Specifically, for n = 1,000,000, n^2 = 1,000,000,000,000.

- These values are different by several orders of magnitude, and as n grows, the difference becomes only more significant.

- In other words, non-dominant terms just don't matter much when we're thinking about functions as their inputs go to ∞, so we simply ignore them in this context.

- To take a final example, let's say we've analyzed a particular algorithm and found that it grows on the order of $n^2 + n + \log n + 1$. Here again, the n^2 term dominates the others, so we say the algorithm's complexity is simply $O(n^2)$.

- The upshot of all this is that when we have loops that are executed *in sequence* (i.e. one loop executed *after* another), then the loop with the highest runtime complexity (i.e. the fastest-growing runtime) will determine the overall runtime complexity of the whole function.

- For example, if we had a function like the following, with an $O(n^2)$ loop executed in sequence between two $O(n)$ loops, the overall complexity would still be $O(n^2)$, since the non-dominant $O(n)$ terms would be dropped from the final growth order function:

```
for (i = 0; i < n; i++) {
  ...
}

for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    ...
  }
}

for (i = 0; i < n; i++) {
  ...
}
```

- This holds no matter how many loops are executed in sequence. For example, we could have a function with 1 million loops executed in sequence, and the

overall runtime complexity of that function would simply be the dominant term among the runtime complexities of the individual loops.

Extrapolating wall clock time from runtime complexity

- Sometimes, it will be important to us to be able to know approximately how long a function will take to run, in terms of ***wall clock time*** (i.e. minutes, seconds, milliseconds, etc.). Knowing the runtime complexity of the function can help us with this.

- Consider, for example, the *O(n)* array summing function we looked at above. What if we wanted to know approximately how long this function would take to run on a problem of a given size? What if, moreover, we already knew how long the function took to run on a problem of a different size? Could we use that to extrapolate?

- For example, what if we knew that this summing function took 32ms to run on an array of 10,000 elements. Could we then say approximately how long it would take to run on an array of 20,000 elements (in other words, with an input that was twice as big)?

- In this case, if we ran our summing function on an array twice as big, we'd expect the loop in that function to run twice as many times. Since the loop is the main contributor to the function's runtime, we'd thus expect the function to take approximately twice as long.

- In other words, if we knew that the function took 32ms on an array of 10,000 elements, we'd expect it to take 64ms on an array of 20,000 elements.

- For an *O(n)* function, we can say in general that if the input size *n* changes by a multiplicative factor of *k*, the runtime will also change by the same factor of *k*.

- What about for nonlinear functions? For example, what if we were working with bubble sort, an $O(n^2)$ function, and we knew that it took 32ms to sort an array of 10,000 elements. How long would we expect it to take, then, to sort an array of 20,000 elements?

- In this case, when we're dealing with an $O(n^2)$ function and we increase the problem size by a factor of 2, we'd actually expect the runtime to increase by a factor of 4. In other words, if bubble sort takes 32ms to sort an array of 10,000 elements, we'd expect it to take 128ms to sort an array of 20,000.

- Why is this? When we say that a function has runtime complexity $O(n^2)$, we're actually saying that the algorithm's runtime will be roughly proportional to n^2. In general, when a function has runtime complexity $O(f(n))$, its runtime is roughly proportional to $f(n)$.

- Thus, for a function with runtime complexity $O(f(n))$, we can actually set up an equation that will help us calculate how long, in wall-clock time, it will take to run the function on a problem of a given size if we know how long that function took to run on a problem of a different size.

- Specifically, for a function with runtime complexity $O(f(n))$ the following equation will hold, where t_1 is the function's runtime on a problem of size n_1, and t_2 is the function's runtime on a problem of size n_2:

$$\frac{f(n_1)}{f(n_2)} = \frac{t_1}{t_2}$$

- In other words, if we're dealing with a function like bubble sort that has runtime complexity $O(n^2)$ and we want to know how long the function will take to run on a problem twice as large as a problem for which we know the wall clock time (i.e. $n_2 = 2n_1$), we can set up the above equation and solve for t_2:

$$\frac{n_1^2}{n_2^2} = \frac{t_1}{t_2}$$

$$\frac{n_1^2}{(2n_1)^2} = \frac{t_1}{t_2}$$

$$\frac{n_1^2}{4n_1^2} = \frac{t_1}{t_2}$$

$$\frac{1}{4} = \frac{t_1}{t_2}$$

$$t_2 = 4t_1$$

- In other words, for an $O(n^2)$ algorithm, if we double the size of the input problem, we should expect the runtime to increase by a factor of 4 (i.e. $t_2 = 4t_1$).

- Let's walk through another example where we use the equation above to determine how much (wall clock) time it will take to run a function on a problem of a particular size given that we know how long it takes to run on another problem of known size.

- Specifically, let's say you're experimenting with merge sort, which is an $O(n \log n)$ algorithm, and you've determined that for your data, merge sort takes 96ms to sort an array of size 4000. Given this result, you'd like to know approximately how long merge sort will take to sort an array of size 1,000,000.

- First, it's important to recognize that when we're making a wall clock time computation like this, we're dealing with an approximation. Platform-specific factors and fluctuations always affect how long it takes for a piece of code to run, and these aren't accounted for by our simple formula here. In addition, as we saw above, the growth order functions we deal with when talking about computational complexity intentionally ignore some constant factors that still have an influence on a function's exact runtime.

- In other words, it's important to know that the formula we're working with here will not exactly compute the precise runtime for a function. Indeed, the runtime could vary somewhat even when running the function on the same data on the same machine.

- With all that in mind, it's generally OK here for us to make minor tweaks to make our calculations easier for us if we expect that those minor tweaks won't have a major impact on the final prediction the formula gives us.

- In this particular situation, for example, we'll make some minor approximations to the numbers we're dealing with make the math a little easier, since we're working with base-2 logarithms:
    - $4000 \approx 4096 = 2^{12}$
    - $1{,}000{,}000 \approx 1{,}048{,}576 = 2^{20}$

- With those approximations made, let's set up the formula as we did before, plugging in known values for n_1, n_2, and t_1 and then solving for t_2:

$$\frac{f(n_1)}{f(n_2)} = \frac{t_1}{t_2}$$

$$\frac{n_1 \log n_1}{n_2 \log n_2} = \frac{t_1}{t_2}$$

$$\frac{2^{12} \log 2^{12}}{2^{20} \log 2^{20}} = \frac{96ms}{t_2}$$

- Now, remember that $\log 2^x = x$. This allows us to simplify the above (after some cancelling) to:

$$\frac{12}{2^8 \cdot 20} = \frac{96ms}{t_2}$$

- Then, cancelling and solving for t_2 gives us:

$$t_2 = 32ms \cdot 5 \cdot 2^8 = 40,960ms \approx 41s$$

- In other words, given that we know merge sort takes 96ms on an array of size 4000 (which, again, is approximately $4096 = 2^{12}$), we'd expect it to take about 41 seconds on a problem of size 1,000,000.

Best-case, worst-case, and average complexity

- Up to now, we've only assessed the complexity of functions with a single execution path (i.e. functions without branching code, like if statements). In such situations, the runtime complexity of the function cannot be different, even if the conditions under which the function is executed change.

- For example, the array summing function we examined above will *always* have *O(n)* runtime complexity, regardless of the inputs to the function, the context in which the function is executed, etc.

- However, this is not always the case. Some functions can, in fact, exhibit a different complexity in different situations.

- To understand how this works, consider the following function, which simply searches for a specified query value `q` within an array. If `q` is found, the function returns its index in the array. Otherwise, it returns -1:

```
int linear_search(int q, int* array, int n) {
  for (int i = 0; i < n; i++) {
    if (array[i] == q) {
      return i;
    }
  }
  return -1;
}
```

- Note that, for the same array size `n`, this function can run more quickly or more slowly depending on the query value `q`. For example, if `q` corresponds to the *first* element in the array, the loop in this function will only run one iteration. If, on the other hand, `q` corresponds to the *last* element in the array, or if the value `q` doesn't exist in the array, then the loop in this function will run *n* iterations.

- Thus, we can say that this function actually has a ***best-case runtime complexity*** that's different from its ***worst-case runtime complexity***. Specifically, this function has a best-case complexity of *O(1)*, while its worst-case

complexity is *O(n)*.

- We can also talk about a function's ***average runtime complexity***. In this case, it's likely that, on average, the loop in the `linear_search()` function here will have to run about *n / 2* iterations to find the query value `q`. Thus, the average complexity of this function is also *O(n)* (since we drop the constant factor of ½ when we're talking about complexities)**.**

Analyzing the complexity of dynamic array insertion

- Insertion into a dynamic array is an interesting case of an operation with different best-case, worst-case, and average runtime complexities.

- Recall that inserting a value into a dynamic array whose capacity is greater than the number of elements stored (i.e. where `size` < `capacity`) is a simple operation.

- In this situation, we simply need to write the new value into the next open space in the array. Because the amount of time it will take to execute this simple operation doesn't depend on the size of the array (*n*), we can say that, in the best case, insertion into a dynamic array is an *O(1)* operation.

- In the worst case, however, inserting a new value into the dynamic array will require allocating a new array, copying all of the values from the old array into the new one, and then inserting the new value. Because this operation entails iterating through the *n* elements in the old array and copying them into the new array, it's runtime complexity is *O(n)*. This is the worst-case complexity for insertion into a dynamic array.

- What about the average complexity, though? Determining this is not as straightforward for dynamic array insertion. In particular, we know in this case that insertion will sometimes be very inexpensive (*O(1)*) and will sometimes be fairly expensive (*O(n)*), but it's hard to say at a glance how expensive dynamic array insertion will be on average.

- To determine how expensive dynamic array insertion will be on average, we'll use a method known as ***amortized analysis***. This form of analysis draws its name from the accounting concept of ***amortization***, in which a large cost is

defrayed by spreading smaller payments over a period of time.

- In amortized analysis, we do something similar by defraying the worst-case cost of an operation (e.g. dynamic array insertion) by considering it within the context of the long-term usage of the operation, in which its worst-case cost might be experienced relatively infrequently in comparison with less costly operations.

- Indeed, this does seem to be the case with dynamic array insertion, since, as we saw when studying the dynamic array, the underlying data storage array is typically *doubled* in size when resizing is needed. Thus, the relatively expensive worst-case *O(n)* cost of dynamic array insertion should be experienced far less often than that operation's best-case *O(1)* cost.

- We can more precisely quantify the average runtime complexity of dynamic array insertion using a version of amortized analysis known as ***aggregate analysis***, which involves computing an upper bound *T* on the total cost of a sequence of *n* operations and then calculates the amortized cost over all *n* operations as the average cost *T* / *n*.

- To begin our analysis, imagine we are using a dynamic array whose capacity starts at 1 and that the array's capacity is doubled each time it needs to be resized. Imagine then that we perform a sequence of *n* insert operations into that dynamic array.

- What will be the total cost of this sequence of *n* insertions? We first need to understand how runtime costs are incurred during dynamic array insertion. We can simplify our analysis without changing its outcome by assuming that the main runtime cost associated with dynamic array insertion is the cost of writing a value into the underlying data storage array, i.e. when we run an operation like this:

```
data[i] = something;
```

- This cost is incurred in two different places:
    - When we first insert a new value into the data storage array when it already has enough capacity to hold a new value.
    - When we copy a value from the old data storage array to a new one during array resizing.

- Of course, there are other costs associated with dynamic array insertion, but in the end, these are relatively negligible in comparison with the cost of writing data.

- Given that we understand *how* we will incur runtime costs during dynamic array insertion, let's figure out *how much* runtime cost we will accumulate over our sequence of *n* total insertions. In particular, let's look at the cost of each *individual* insertion operation in the sequence:
    - In the first insertion (when the array is empty), we will have to perform one single write operation to insert the new value into the array. Thus, we can say that the cost of this operation will be 1.
    - In the second insertion, we will have to resize the array (doubling its capacity from 1 to 2). Thus, for this insertion, we will incur a cost of 2: a cost of 1 to copy the one original value from the old array to the new one and a cost of 1 to write the new value into the array.
    - In the third insertion, we will again have to double the capacity of the array from 2 to 4. Thus, for this insertion, we will incur a total cost of 3: a cost of 2 to copy two values from the old array to the new one and a cost of 1 to write the new value into the array.
    - In the fourth insertion, there will be enough capacity to simply write the new value into the array, so the cost of this operation will again be 1.
    - In the fifth insertion, we will again have to double the size of the array from 4 to 8, so we will incur a total cost of 5: 4 to copy values to the new array and 1 to write the new value.
    - The sixth, seventh, and eighth insertions will each have a cost of 1, since there will be enough capacity to simply insert the new value into the array during these operations.
    - The ninth insertion will have a cost of 9, since we will have to resize the array during this operation.
    - And so forth…

- We can tabulate these costs as follows (where "initial write cost" represents the cost to write a new element into the array and "copy cost" represents the cost to copy values from the old data storage array to the new one during a resize):

Insertion # (resize # (*k*))	1	2 (1)	3 (2)	4	5 (3)	6	7	8	9 (4)	10	...
Initial write cost	1	1	1	1	1	1	1	1	1	1	...

Copy cost	0	1	2	0	4	0	0	0	8	0	...

- It should be clear here that, over *n* total insertions, the total, cumulative cost of initial writes will be *n*.

- What about the total copy cost? We only incur copy costs during array resizes, so let's first figure out *how many* resizes we'll perform. Remember that we double the capacity of the data storage array each time we resize. Starting with a capacity of 1, then, the number of times we must double the capacity before we can hold *n* elements is *log n***.** Thus, assuming for simplicity that *log n* evaluates to a whole number, we will perform *log n* resize operations during this sequence of *n* insertions.

- Further, from the table above, it should be clear that the *k*'th resize operation incurs a copy cost of 2^{k-1}. Thus, the total, *cumulative* copy cost incurred over the entire sequence of *n* insertions (again, knowing that we will perform *log n* total resizes) can be expressed as the following geometric series:

$$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{\log n - 1}$$

- The sum of this series is $2^{\log n} - 1$ or, using the definition of a (base 2) logarithm, simply $n - 1$. Thus, over the entire sequence of *n* insert operations, we incur a total, cumulative copy cost of $n - 1$.

- So, the *overall* cost of the entire sequence of *n* insert operations—equal to the total, cumulative cost of initial writes plus the total, cumulative copy cost—is:

$$n + (n - 1) = 2n - 1$$

- Again, this represents the *total* cost over the entire sequence of *n* operations. The final step of aggregate analysis involves computing the amortized cost as the *average* of this total cost over *n* operations.

- Thus, the amortized cost and average runtime complexity of dynamic array insertion is:

$$\frac{2n-1}{n} = O(1)$$

- In other words, on average, dynamic array insertion is a constant time operation, despite the fact that it occasionally incurs an expensive *O(n)* cost.

- The analysis above should make it clear why doubling the capacity of the array each time a resize is needed is actually an important strategy for controlling the average cost of dynamic array insertion.
    - Can you explain, for example, why it would be a *bad* strategy to increase the capacity of the array by some constant amount (e.g. 100) each time a resize was needed?

Runtime complexity of linked list operations

- It should now be straightforward for us to analyze the runtime complexities of linked list operations.

- Let's start with insertion into a linked list. Importantly, for now, let's assume that we already know exactly where in the list we want to insert a new value (e.g. at the head or at the tail).

- In this case, the only operations involved in inserting a new value into a linked list (allocating a new link, updating next pointers, etc.) all run in constant time. Thus, when the insertion location is known, linked list insertion has runtime complexity *O(1)*.
    - Because there are no alternative code paths during linked list insertion, best-case, worst-case, and average runtime complexity are all *O(1)*.

- Let's turn our attention now to removal from a linked list. Assuming again that we know the location of the link we want to remove, linked list removal is also an *O(1)* operation, since, again, the operations involved all run in constant time.
    - Linked list removal also has no alternative code paths, so it also has the same best-case, worst-case, and average case complexity of *O(1)*.

- Because linked lists have worst-case *O(1)* insertions, they are often used in lieu of dynamic arrays when guarantees are needed on how fast any given operation will take to run. In other words, linked lists avoid the dynamic array's occasional

very expensive insert operation.

- A major drawback of linked lists is that they *do not* allow direct access to their elements. Specifically, assuming we know the "index" of the element we want to access in a dynamic array, it will take longer to access elements further from the head of the list (in a singly-linked list), since the only way to reach these elements is to start at the head and iterate through the list, stepping from link to link until we reach the link we want to access.

- Because this iteration is needed, accessing an element in a linked list is an *O(n)* operation (in the worst case and also on average).
    - By contrast, dynamic arrays support direct access of the elements they contain. In other words, accessing an element in a dynamic array is an *O(1)* operation.

How and when do we use complexity analysis?

- Having now covered complexity analysis in a fairly detailed way, you might be wondering at this point when and how we actually use complexity analysis.

- Remember that one of the primary objectives of this course is to help you be able to compare data structures and weigh their trade-offs, so you can choose the right one for a particular problem.
- Complexity analysis is a tool that can help make this kind of comparison.

- However, it's also important to understand that complexity analysis paints with a very broad brush. In other words, complexity analysis gives us a view of data structures and algorithms that is purposefully lacking in fine detail.

- Thus, given a set of candidate data structures or algorithms for solving a problem, complexity analysis might best be seen as a tool that makes it easy to recognize which of those candidates will simply have too high of a cost to work well for our problem and to focus on those candidates whose cost is manageable.

- For example, if you were working on a problem where runtime efficiency was crucial, and you were considering some candidate data structures/algorithms with $O(n^2)$ runtime complexity and others with *O(n log n)* complexity, it will likely make the most sense to quickly eliminate the $O(n^2)$ ones from consideration and

to focus on the *O(n log n)* ones, since the $O(n^2)$ data structures/algorithms will likely be too expensive for large amounts of data compared to the *O(n log n)* ones.

- Then, once you've used complexity analysis to quickly eliminate some candidates, you might spend more time thoroughly testing the remaining ones, e.g. measuring their actual wall-clock runtimes on real instances of the problem you're trying to solve, making your final choice based on this more thorough analysis.
    - Here, the more thorough analysis can help you to uncover some of the constant factors and platform-dependent details that complexity analysis ignores.

- At the same time, runtime (or memory) complexity is only one factor to consider among many when choosing a data structure or algorithm to solve a given problem.

- For example, you may be working on a problem in which you'll need to sort many small arrays and where code simplicity is of more concern to you than speed of execution. In such a case, it might make the most sense to choose an algorithm like insertion sort because of its simple implementation, despite the fact that its $O(n^2)$ runtime complexity is higher than that of many other sorting algorithms.