WEEK-07 DAY-2
Tuesday - *Big-O and Optimizations*

Big O Learning Objectives

The objective of this lesson is get you comfortable with identifying the time and space complexity of code you see. Being able to diagnose time complexity for algorithms is an essential for interviewing software engineers.

At the end of this, you will be able to

1. Order the common complexity classes according to their growth rate
2. Identify the complexity classes of common sort methods
3. Identify complexity classes of code

Memoization And Tabulation Learning Objectives

The objective of this lesson is to give you a couple of ways to optimize a computation (algorithm) from a higher complexity class to a lower complexity class. Being able to optimize algorithms is an essential for interviewing software engineers.

At the end of this, you will be able to

1. Apply memoization to recursive problems to make them less than polynomial time.
2. Apply tabulation to iterative problems to make them less than polynomial time.

Recursion Videos

A lot of algorithms that we use in the upcoming days will use recursion. The next two videos are just helpful reminders about recursion so that you can get that thought process back into your brain.

Big-O By Colt Steele

Colt Steele provides a very nice, non-mathy introduction to Big-O notation. Please watch this so you can get the easy introduction. Big-O is, by its very nature, math based. It's good to get an understanding before jumping in to math expressions.

Complete Beginner's Guide to Big O Notation by Colt Steele.

Curating Complexity: A Guide to Big-O Notation

As software engineers, our goal is not just to solve problems. Rather, our goal is to solve problems efficiently and elegantly. Not all solutions are made equal! In this section we'll explore how to analyze the efficiency of algorithms in terms of their speed (*time complexity*) and memory consumption (*space complexity*).

> In this article, we'll use the word *efficiency* to describe the amount of resources a program needs to execute. The two resources we are concerned with are *time* and *space*. Our goal is to *minimize* the amount of time and space that our programs use.

When you finish this article you will be able to:

- explain why computer scientists use Big-O notation
- simplify a mathematical function into Big-O notation

Why Big-O?

Let's begin by understanding what method we should *not* use when describing the efficiency of our algorithms. Most importantly, we'll want to avoid using absolute units of time when describing speed. When the software engineer exclaims, "My function runs in 0.2 seconds, it's so fast!!!", the computer scientist is not impressed. Skeptical, the computer scientist asks the following questions:

1. What computer did you run it on? *Maybe the credit belongs to the hardware and not the software. Some hardware architectures will be better for certain operations than others.*
2. Were there other background processes running on the computer that could have effected the runtime? *It's hard to control the environment during performance experiments.*
3. Will your code still be performant if we increase the size of the input? *For example, sorting 3 numbers is trivial; but how about a million numbers?*

The job of the software engineer is to focus on the software detail and not necessarily the hardware it will run on. Because we can't answer points 1 and 2 with total certainty, we'll want to avoid using concrete units like "milliseconds" or "seconds" when describing the efficiency of our algorithms. Instead, we'll opt for a more abstract approach that focuses on point 3. This means that we should focus on how the performance of our algorithm is affected by increasing the size of the input. **In other words, how does our performance scale?**

> The argument above focuses on *time*, but a similar argument could also be made for *space*. For example, we should not analyze our code in terms of the amount of absolute kilobytes of memory it uses, because this is dependent on the programming language.

Big-O Notation

In Computer Science, we use Big-O notation as a tool for describing the efficiency of algorithms with respect to the size of the input argument(s). We use mathematical functions in Big-O notation, so there are a few big picture ideas that we'll want to keep in mind:

1. The function should be defined in terms of the size of the input(s).
2. A *smaller* Big-O function is more desirable than a larger one. Intuitively, we want our algorithms to use a minimal amount of time and space.
3. Big-O describes the worst-case scenario for our code, also known as the upper bound. We prepare our algorithm for the worst case, because the best case is a luxury that is not guaranteed.
4. A Big-O function should be simplified to show only its most dominant mathematical term.

The first 3 points are conceptual, so they are easy to swallow. However, point 4 is typically the biggest source of confusion when learning the notation. Before we apply Big-O to our code, we'll need to first understand the underlying math and simplification process.

Simplifying Math Terms

We want our Big-O notation to describe the performance of our algorithm with respect to the input size and nothing else. Because of this, we should to simplify our Big-O functions using the following rules:

- **Simplify Products:** if the function is a product of many terms, we drop the terms that *don't* depend on the size of the input.
- **Simplify Sums:** if the function is a sum of many terms, we keep the term with the *largest* growth rate and drop the other terms.

We'll look at these rules in action, but first we'll define a few things:

- **n** is the size of the input
- **T(f)** refers to an unsimplified mathematical **f**unction
- **O(f)** refers to the Big-O simplified mathematical **f**unction

Simplifying a Product

If a function consists of a product of many factors, we drop the factors that don't depend on the size of the input, n. The factors that we drop are called constant factors because their size remains consistent as we increase the size of the input. The reasoning behind this simplification is that we make the input large enough, the non-constant factors will overshadow the constant ones. Below are some examples:

Unsimplified	Big-O Simplified
T(5 * n^2)	O(n^2)
T(100000 * n)	O(n)
T(n / 12)	O(n)
T(42 * n * log(n))	O(n * log(n))
T(12)	O(1)

Note that in the third example, we can simplify `T( n / 12 )` to `O( n )` because we can rewrite a division into an equivalent multiplication. In other words, `T( n / 12 ) = T( 1/12 * n ) = O( n )`.

Simplifying a Sum

If the function consists of a sum of many terms, we only need to show the term that grows the fastest, relative to the size of the input. The reasoning behind this simplification is that if we make the input large enough, the fastest growing term will overshadow the other, smaller terms. To understand which term to keep, you'll need to recall the relative size of our common math terms from the previous section. Below are some examples:

Unsimplified	Big-O Simplified
T(n^3 + n^2 + n)	O(n^3)
T(log(n) + 2^n)	O(2^n)
T(n + log(n))	O(n)
T(n! + 10^n)	O(n!)

Putting it all together

The *product* and *sum* rules are all we'll need to Big-O simplify any math functions. We just apply the *product rule* to drop all constants, then apply the *sum rule* to select the single most dominant term.

Unsimplified	Big-O Simplified
T($5n^2$ + 99n)	O(n^2)
T(2n + nlog(n))	O(nlog(n))
T(2^n + $5n^{1000}$)	O(2^n)

> Aside: We'll often omit the multiplication symbol in expressions as a form of shorthand. For example, we'll write *O($5n^2$)* in place of *O(5 * n^2)*.

What you've learned

In this reading we:

- explained why Big-O is the preferred notation used to describe the efficiency of algorithms
- used the product and sum rules to simplify mathematical functions into Big-O notation

Common Complexity Classes

Analyzing the efficiency of our code seems like a daunting task because there are many different possibilities in how we may choose to implement something. Luckily, most code we write can be categorized into one of a handful of common complexity classes. In this reading, we'll identify the common classes and explore some of the code characteristics that will lead to these classes.

When you finish this reading, you should be able to:

- name *and* order the seven common complexity classes
- identify the time complexity class of a given code snippet

The seven major classes

There are seven complexity classes that we will encounter most often. Below is a list of each complexity class as well as its Big-O notation. This list is ordered from *smallest to largest*. Bear in mind that a "more efficient" algorithm is one with a smaller complexity class, because it requires fewer resources.

Big-O	Complexity Class Name
O(1)	constant
O(log(n))	logarithmic
O(n)	linear
O(n * log(n))	loglinear, linearithmic, quasilinear
O(n^c) - O(n^2), O(n^3), etc.	polynomial
O(c^n) - O(2^n), O(3^n), etc.	exponential
O(n!)	factorial

There are more complexity classes that exist, but these are most common. Let's take a closer look at each of these classes to gain some intuition on what behavior their functions define. We'll explore famous algorithms that correspond to these classes further in the course.

For simplicity, we'll provide small, generic code examples that illustrate the complexity, although they may not solve a practical problem.

O(1) - Constant

Constant complexity means that the algorithm takes roughly the same number of steps for any size input. In a constant time algorithm, there is no relationship between the size of the input and the number of steps required. For example, this means performing the algorithm on a input of size 1 takes the same number of steps as performing it on an input of size 128.

Constant growth

The table below shows the growing behavior of a constant function. Notice that the behavior stays *constant* for all values of n.

n	O(1)
1	~1
2	~1
3	~1
...	...
128	~1

Example Constant code

Below is are two examples of functions that have constant runtimes.

```
// O(1)
function constant1(n) {
  return n * 2 + 1;
}

// O(1)
function constant2(n) {
  for (let i = 1; i <= 100; i++) {
    console.log(i);
  }
}
```

The runtime of the `constant1` function does not depend on the size of the input, because only two arithmetic operations (multiplication and addition) are always performed. The runtime of the `constant2` function also does not depend on the size of the input because one-hundred iterations are always performed, irrespective of the input.

O(log(n)) - Logarithmic

Typically, the hidden base of O(log(n)) is 2, meaning O($\log_2$(n)). Logarithmic complexity algorithms will usual display a sense of continually "halving" the size of the input. Another tell of a logarithmic algorithm is that we don't have to access every element of the input. O($\log_2$(n)) means that every time we double the size of the input, we only require one additional step. Overall, this means that a large increase of input size will increase the number of steps required by a small amount.

Logarithmic growth

The table below shows the growing behavior of a logarithmic runtime function. Notice that doubling the input size will only require only one additional "step".

n	O($\log_2$(n))
2	~1
4	~2
8	~3
16	~4
...	...
128	~7

Example logarithmic code

Below is an example of two functions with logarithmic runtimes.

```
// O(log(n))
function logarithmic1(n) {
  if (n <= 1) return;
  logarithmic1(n / 2);
}

// O(log(n))
function logarithmic2(n) {
  let i = n;
  while (i > 1) {
    i /= 2;
  }
}
```

The `logarithmic1` function has O(log(n)) runtime because the recursion will half the argument, n, each time. In other words, if we pass 8 as the original argument, then the recursive chain would be 8 -> 4 -> 2 -> 1. In a similar way, the `logarithmic2` function has O(log(n)) runtime because of the number of iterations in the while loop. The while loop depends on the variable `i`, which will be divided in half each iteration.

O(n) - Linear

Linear complexity algorithms will access each item of the input "once" (in the Big-O sense). Algorithms that iterate through the input without nested loops or recurse by reducing the size of the input by "one" each time are typically linear.

Linear growth

The table below shows the growing behavior of a linear runtime function. Notice that a change in input size leads to similar change in the number of steps.

n	O(n)
1	~1
2	~2
3	~3
4	~4
...	...
128	~128

Example linear code

Below are examples of three functions that each have linear runtime.

```
// O(n)
function linear1(n) {
  for (let i = 1; i <= n; i++) {
    console.log(i);
  }
}

// O(n), where n is the length of the array
function linear2(array) {
  for (let i = 0; i < array.length; i++) {
    console.log(i);
  }
}

// O(n)
function linear3(n) {
  if (n === 1) return;
  linear3(n - 1);
}
```

The `linear1` function has O(n) runtime because the for loop will iterate n times. The `linear2` function has O(n) runtime because the for loop iterates through the array argument. The `linear3` function has O(n) runtime because each subsequent call in the recursion will decrease the argument by one. In other words, if we pass 8 as the original argument to `linear3`, the recursive chain would be 8 -> 7 -> 6 -> 5 -> ... -> 1.

O(n * log(n)) - Loglinear

This class is a combination of both linear and logarithmic behavior, so features from both classes are evident. Algorithms the exhibit this behavior use both recursion and iteration. Typically, this means that the recursive calls will halve the input each time (logarithmic), but iterations are also performed on the input (linear).

Loglinear growth

The table below shows the growing behavior of a loglinear runtime function.

n	$O(n * log_2(n))$
2	~2
4	~8
8	~24
...	...
128	~896

Example loglinear code

Below is an example of a function with a loglinear runtime.

```
// O(n * log(n))
function loglinear(n) {
  if (n <= 1) return;

  for (let i = 1; i <= n; i++) {
    console.log(i);
  }

  loglinear(n / 2);
  loglinear(n / 2);
}
```

The `loglinear` function has O(n * log(n)) runtime because the for loop iterates linearly (n) through the input and the recursive chain behaves logarithmically (log(n)).

$O(n^c)$ - Polynomial

Polynomial complexity refers to complexity of the form $O(n^c)$ where `n` is the size of the input and `c` is some fixed constant. For example, $O(n^3)$ is a larger/worse function than $O(n^2)$, but they belong to the same complexity class. Nested loops are usually the indicator of this complexity class.

Polynomial growth

Below are tables showing the growth for $O(n^2)$ and $O(n^3)$.

n	$O(n^2)$
1	~1
2	~4
3	~9
...	...

n	O(n^2)
128	~16,384

n	O(n^3)
1	~1
2	~8
3	~27
...	...
128	~2,097,152

Example polynomial code

Below are examples of two functions with polynomial runtimes.

```
// O(n^2)
function quadratic(n) {
  for (let i = 1; i <= n; i++) {
    for (let j = 1; j <= n; j++) {}
  }
}

// O(n^3)
function cubic(n) {
  for (let i = 1; i <= n; i++) {
    for (let j = 1; j <= n; j++) {
      for (let k = 1; k <= n; k++) {}
    }
  }
}
```

The `quadratic` function has O(n^2) runtime because there are nested loops. The outer loop iterates n times and the inner loop iterates n times. This leads to n * n total number of iterations. In a similar way, the `cubic` function has O(n^3) runtime because it has triply nested loops that lead to a total of n * n * n iterations.

O(c^n) - Exponential

Exponential complexity refers to Big-O functions of the form O(c^n) where n is the size of the input and c is some fixed constant. For example, O(3^n) is a larger/worse function than O(2^n), but they both belong to the exponential complexity class. A common indicator of this complexity class is recursive code where there is a constant number of recursive calls in each stack frame. The `c` will be the number of recursive calls made in each stack frame. Algorithms with this complexity are considered quite slow.

Exponential growth

Below are tables showing the growth for O(2^n) and O(3^n). Notice how these grow large, quickly.

n	O(2^n)
1	~2
2	~4
3	~8
4	~16
...	...
128	~3.4028 * 10^{38}

n	O(3^n)
1	~3
2	~9
3	~27
3	~81
...	...
128	~1.1790 * 10^{61}

Exponential code example

Below are examples of two functions with exponential runtimes.

```
// O(2^n)
function exponential2n(n) {
  if (n === 1) return;
  exponential_2n(n - 1);
  exponential_2n(n - 1);
}

// O(3^n)
function exponential3n(n) {
  if (n === 0) return;
  exponential_3n(n - 1);
  exponential_3n(n - 1);
  exponential_3n(n - 1);
}
```

The `exponential2n` function has O(2^n) runtime because each call will make two more recursive calls. The `exponential3n` function has O(3^n) runtime because each call will make three more recursive calls.

O(n!) - Factorial

Recall that `n! = (n) * (n - 1) * (n - 2) * ... * 1`. This complexity is typically the largest/worst that we will end up implementing. An indicator of this complexity class is recursive code that has a variable number of recursive calls in each stack frame. Note that *factorial* is worse than *exponential* because *factorial* algorithms have a *variable* amount of recursive calls in each stack frame, whereas *exponential* algorithms have a *constant* amount of recursive calls in each frame.

Factorial growth

Below is a table showing the growth for O(n!). Notice how this has a more aggressive growth than exponential behavior.

n	O(n!)
1	~1
2	~2
3	~6
4	~24
...	...
128	~3.8562 * 10^{215}

Factorial code example

Below is an example of a function with factorial runtime.

```
// O(n!)
function factorial(n) {
  if (n === 1) return;

  for (let i = 1; i <= n; i++) {
    factorial(n - 1);
  }
}
```

The `factorial` function has O(n!) runtime because the code is *recursive* but the number of recursive calls made in a single stack frame depends on the input. This contrasts with an *exponential* function because exponential functions have a *fixed* number of calls in each stack frame.

You may it difficult to identify the complexity class of a given code snippet, especially if the code falls into the loglinear, exponential, or factorial classes. In the upcoming videos, we'll explain the analysis of these functions in greater detail. For now, you should focus on the *relative order* of these seven complexity classes!

What you've learned

In this reading, we listed the seven common complexity classes and saw some example code for each. In order of ascending growth, the seven classes are:

1. Constant
2. Logarithmic
3. Linear
4. Loglinear
5. Polynomial
6. Exponential
7. Factorial

Memoization

Memoization is a design pattern used to reduce the overall number of calculations that can occur in algorithms that use recursive strategies to solve.

Recall that recursion solves a large problem by dividing it into smaller sub-problems that are more manageable. Memoization will store the results of the sub-problems in some other data structure, meaning that you avoid duplicate calculations and only "solve" each subproblem once. There are two features that comprise memoization:

- the function is recursive
- the additional data structure used is typically an object (we refer to this as the memo!)

This is a trade-off between the time it takes to run an algorithm (without memoization) and the memory used to run the algorithm (with memoization). Usually memoization is a good trade-off when dealing with large data or calculations.

You cannot always apply this technique to recursive problems. The problem must have an "overlapping subproblem structure" for memoization to be effective.

Here's an example of a problem that has such a structure:

> Using pennies, nickels, dimes, and quarters, what is the smallest combination of coins that total 27 cents?

You'll explore this exact problem in depth later on. For now, here is some food for thought. Along the way to calculating the smallest coin combination of 27 cents, you should also calculate the smallest coin combination of say, 25 cents as a component of that problem. This is the essence of an overlapping subproblem structure.

Memoizing factorial

Here's an example of a function that computes the factorial of the number passed into it.

```
function factorial(n) {
  if (n === 1) return 1;
  return n * factorial(n - 1);
}

factorial(6);       // => 720, requires 6 calls
factorial(6);       // => 720, requires 6 calls
factorial(5);       // => 120, requires 5 calls
factorial(7);       // => 5040, requires 7 calls
```

From this plain `factorial` above, it is clear that every time you call `factorial(6)` you should get the same result of `720` each time. The code is somewhat inefficient because you must go down the full recursive stack for each top level call to `factorial(6)`. It would be great if you could store the result of `factorial(6)` the first time you calculate it, then on subsequent calls to `factorial(6)` you simply fetch the stored result in constant time. You can accomplish exactly this by memoizing with an object!

```
let memo = {}

function factorial(n) {
  // if this function has calculated factorial(n) previously,
  // fetch the stored result in memo
  if (n in memo) return memo[n];
  if (n === 1) return 1;

  // otherwise, it havs not calculated factorial(n) previously,
  // so calculate it now, but store the result in case it is
  // needed again in the future
  memo[n] = n * factorial(n - 1);
  return memo[n]
}

factorial(6);       // => 720, requires 6 calls
factorial(6);       // => 720, requires 1 call
factorial(5);       // => 120, requires 1 call
factorial(7);       // => 5040, requires 2 calls

memo;   // => { '2': 2, '3': 6, '4': 24, '5': 120, '6': 720, '7': 5040 }
```

The `memo` object above will map an argument of `factorial` to its return value. That is, the keys will be arguments and their values will be the corresponding results returned. By using the memo, you are able to avoid duplicate recursive calls!

Here's some food for thought: By the time your first call to `factorial(6)` returns, you will not have just the argument `6` stored in the memo. Rather, you will have *all* arguments 2 to 6 stored in the memo.

Hopefully you sense the efficiency you can get by memoizing your functions, but maybe you are not convinced by the last example for two reasons:

- You didn't improve the speed of the algorithm by an order of Big-O (it is still O(n)).
- The code uses some global variable, so it's kind of ugly.

Both of those points are true, so take a look at a more advanced example that benefits from memoization.

Memoizing the Fibonacci generator

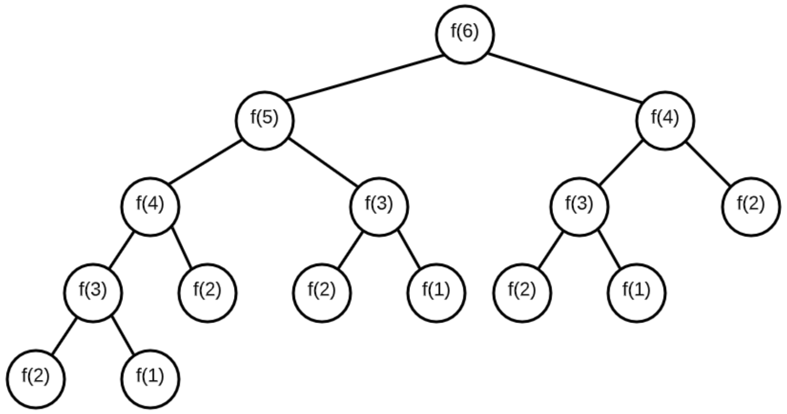
Here's a *naive* implementation of a function that calculates the Fibonacci number for a given input.

```
function fib(n) {
  if (n === 1 || n === 2) return 1;
  return fib(n - 1) + fib(n - 2);
}

fib(6);     // => 8
```

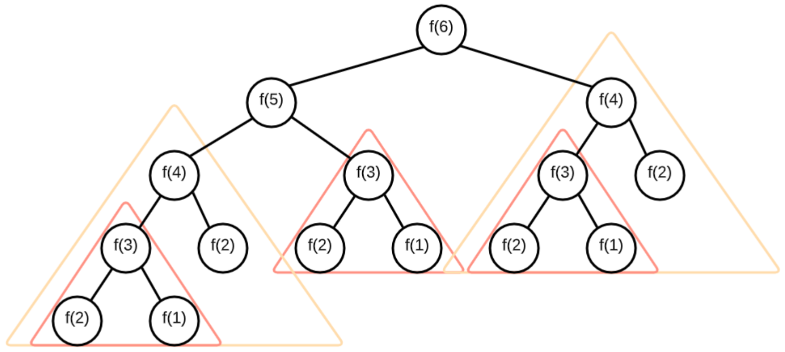
Before you optimize this, ask yourself what complexity class it falls into in the first place.

The time complexity of this function is not super intuitive to describe because the code branches twice recursively. Fret not! You'll find it useful to visualize the calls needed to do this with a tree. When reasoning about the time complexity for recursive functions, draw a tree that helps you see the calls. Every node of the tree represents a call of the recursion:



In general, the height of this tree will be n. You derive this by following the path going straight down the left side of the tree. You can also see that each internal node leads to two more nodes. Overall, this means that the tree will have roughly 2^n nodes which is the same as saying that the `fib` function has an exponential time complexity of 2^n. That is very slow! See for yourself, try running `fib(50)` - you'll be waiting for quite a while (it took 3 minutes on the author's machine).

Okay. So the `fib` function is slow. Is there anyway to speed it up? Take a look at the tree above. Can you find any repetitive regions of the tree?



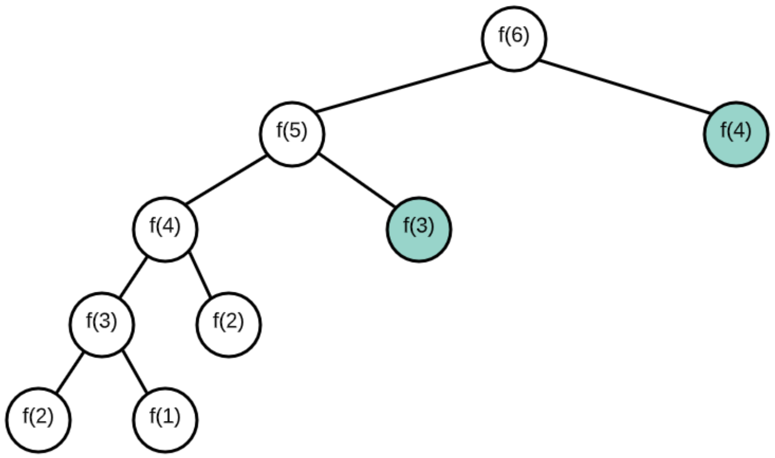
As the n grows bigger, the number of duplicate sub-trees grows exponentially. Luckily you can fix this using memoization by using a similar object strategy as before. You can use some JavaScript default arguments to clean things up:

```
function fastFib(n, memo = {}) {
  if (n in memo) return memo[n];
  if (n === 1 || n === 2) return 1;

  memo[n] = fastFib(n - 1, memo) + fastFib(n - 2, memo);
  return memo[n];
}

fastFib(6);     // => 8
fastFib(50);    // => 12586269025
```

The code above can calculate the 50th Fibonacci number almost instantly! Thanks to the `memo` object, you only need to explore a subtree fully once. Visually, the `fastFib` recursion has this structure:



You can see the marked nodes (function calls) that access the memo in green. It's easy to see that this version of the Fibonacci generator will do far less computations as n grows larger! In fact, this memoization has brought the time complexity down to linear `O(n)` time because the tree only branches on the left side. This is an enormous gain if you recall the complexity class hierarchy.

The memoization formula

Now that you understand memoization, when should you apply it? Memoization is useful when attacking recursive problems that have many overlapping sub-problems. You'll find it most useful to draw out the visual tree first. If you notice duplicate sub-trees, time to memoize. Here are the hard and fast rules you can use to memoize a slow function:

1. Write the unoptimized, brute force recursion and make sure it works.
2. Add the memo object as an additional argument to the function. The keys will represent unique arguments to the function, and their values will represent the results for those arguments.
3. Add a base case condition to the function that returns the stored value if the function's argument is in the memo.
4. Before you return the result of the recursive case, store it in the memo as a value and make the function's argument it's key.

What you learned

You learned a secret to possibly changing an algorithm of one complexity class to a lower complexity class by using memory to store intermediate results. This is a powerful technique to use to make sure your programs that must do recursive calculations can benefit from running much faster.

Tabulation

Now that you are familiar with *memoization*, you can explore a related method of algorithmic optimization: **Tabulation**. There are two main features that comprise the Tabulation strategy:

- the function is iterative and *not* recursive
- the additional data structure used is typically an array, commonly referred to as the table

Many problems that can be solved with memoization can also be solved with tabulation as long as you convert the recursion to iteration. The first example is the canonical example of recursion, calculating the Fibonacci number for an input. However, in the example, you'll see the iteration version of it for a fresh start!

Tabulating the Fibonacci number

Tabulation is all about creating a table (array) and filling it out with elements. In general, you will complete the table by filling entries from "left to right". This means that the first entry of the table (first element of the array) will correspond to the smallest subproblem. Naturally, the final entry of the table (last element of the array) will correspond to the largest problem, which is also the final answer.

Here's a way to use tabulation to store the intermediary calculations so that later calculations can refer back to the table.

```
function tabulatedFib(n) {
  // create a blank array with n reserved spots
  let table = new Array(n);

  // seed the first two values
  table[0] = 0;
  table[1] = 1;

  // complete the table by moving from left to right,
  // following the fibonacci pattern
  for (let i = 2; i <= n; i += 1) {
    table[i] = table[i - 1] + table[i - 2];
  }

  return table[n];
}

console.log(tabulatedFib(7));      // => 13
```

When you initialized the table and seeded the first two values, it looked like this:

i	0	1	2	3	4	5	6	7
`table[i]`	`0`	`1`						

After the loop finishes, the final table will be:

i	0	1	2	3	4	5	6	7
`table[i]`	`0`	`1`	`1`	`2`	`3`	`5`	`8`	`13`

Similar to the previous `memo`, by the time the function completes, the `table` will contain the final solution as well as all sub-solutions calculated along the way.

To compute the complexity class of this `tabulatedFib` is very straightforward since the code is iterative. The dominant operation in the function is the loop used to fill out the entire table. The length of the table is roughly `n` elements long, so the algorithm will have an *O(n)* runtime. The space taken by our algorithm is also *O(n)* due to the size of the table. Overall, this should be a satisfying solution for the efficiency of the algorithm.

Aside: Refactoring for O(1) Space

You may notice that you can cut down on the space used by the function. At any point of the loop, the calculation really only need the previous two subproblems' results. There is little utility to storing the full array. This refactor is easy to do by using two variables:

```
function fib(n) {
  let mostRecentCalcs = [0, 1];

  if (n === 0) return mostRecentCalcs[0];

  for (let i = 2; i <= n; i++) {
    const [ secondLast, last ] = mostRecentCalcs;
    mostRecentCalcs = [ last, secondLast + last ];
  }

  return mostRecentCalcs[1];
}
```

Bam! You now have O(n) runtime and O(1) space. This is the most optimal algorithm for calculating a Fibonacci number. Note that this strategy is a pared down form of tabulation, since it uses only the last two values.

The Tabulation Formula

Here are the general guidelines for implementing the tabulation strategy. This is just a general recipe, so adjust for taste depending on your problem:

1. Create the table array based off of the size of the input, which isn't always straightforward if you have multiple input values
2. Initialize some values in the table that "answer" the trivially small subproblem usually by initializing the first entry (or entries) of the table
3. Iterate through the array and fill in remaining entries, using previous entries in the table to perform the current calculation
4. Your final answer is (usually) the last entry in the table

What you learned

You learned another way of possibly changing an algorithm of one complexity class to a lower complexity class by using memory to store intermediate results. This is a powerful technique to use to make sure your programs that must do iterative calculations can benefit from running much faster.

Analysis of Linear Search

Consider the following search algorithm known as **linear search**.

```
function search(array, term) {
  for (let i = 0; i < array.length; i++) {
    if (array[i] == term) {
      return i;
    }
  }
  return -1;
}
```

Most Big-O analysis is done on the "worst-case scenario" and provides an upper bound. In the worst case analysis, you calculate the upper bound on running time of an algorithm. You must know the case that causes the maximum number of operations to be executed.

For *linear search*, the worst case happens when the element to be searched (`term` in the above code) is not present in the array. When `term` is not present, the `search` function compares it with all the elements of `array` one by one. Therefore, the worst-case time complexity of linear search would be O(n).

Analysis of Binary Search

Consider the following search algorithm known as the **binary search**. This kind of search only works if the array is already sorted.

```
function binarySearch(arr, x, start, end) {
  if (start > end) return false;

  let mid = Math.floor((start + end) / 2);
  if (arr[mid] === x) return true;

  if (arr[mid] > x) {
    return binarySearch(arr, x, start, mid - 1);
  } else {
    return binarySearch(arr, x, mid + 1, end);
  }
}
```

For the *binary search*, you cut the search space in half every time. This means that it reduces the number of searches you must do by half, every time. That means the number of steps it takes to get to the desired item (if it exists in the array), in the worst case takes the same amount of steps for every number within a range defined by the powers of 2.

- 7 -> 4 -> 2 -> 1
- 8 -> 4 -> 2 -> 1
- 9 -> 5 -> 3 -> 2 -> 1
- 15 -> 8 -> 4 -> 2 -> 1
- 16 -> 8 -> 4 -> 2 -> 1
- 17 -> 9 -> 5 -> 3 -> 2 -> 1
- 31 -> 16 -> 8 -> 4 -> 2 -> 1
- 32 -> 16 -> 8 -> 4 -> 2 -> 1
- 33 -> 17 -> 9 -> 5 -> 3 -> 2 -> 1

So, for any number of items in the sorted array between 2^{n-1} and 2^n, it takes *n* number of steps. That means if you have *k* items in the array, then it will take log_2k.

Binary searches are $O(log_2n)$.

Analysis of the Merge Sort

Consider the following divide-and-conquer sort method known as the **merge sort**.

```
function merge(leftArray, rightArray) {
  const sorted = [];
  while (leftArray.length > 0 && rightArray.length > 0) {
    const leftItem = leftArray[0];
    const rightItem = rightArray[0];

    if (leftItem > rightItem) {
      sorted.push(rightItem);
      rightArray.shift();
    } else {
      sorted.push(leftItem);
      leftArray.shift();
    }
  }

  while (leftArray.length !== 0) {
    const value = leftArray.shift();
    sorted.push(value);
  }
```

```
    while (rightArray.length !== 0) {
      const value = rightArray.shift();
      sorted.push(value);
    }

    return sorted
  }

  function mergeSort(array) {
    const length = array.length;
    if (length == 1) {
      return array;
    }

    const middleIndex = Math.ceil(length / 2);
    const leftArray = array.slice(0, middleIndex);
    const rightArray = array.slice(middleIndex, length);

    leftArray = mergeSort(leftArray);
    rightArray = mergeSort(rightArray);

    return merge(leftArray, rightArray);
  }
```

For the *merge sort*, you cut the sort space in half every time. In each of those halves, you have to loop through the number of items in the array. That means that, for the worst case, you get that same $log_2 n$ but it must be multiplied by the number of elements in the array, n.

Merge sorts are $O(n*log_2 n)$.

Analysis of Bubble Sort

Consider the following sort algorithm known as the **bubble sort**.

```
function bubbleSort(items) {
  var length = items.length;
  for (var i = 0; i < length; i++) {
    for (var j = 0; j < (length - i - 1); j++) {
      if (items[j] > items[j + 1]) {
        var tmp = items[j];
        items[j] = items[j + 1];
        items[j + 1] = tmp;
      }
    }
  }
}
```

For the *bubble sort*, the worst case is the same as the best case because it always makes nested loops. So, the outer loop loops the number of times of the items in the array. For each one of those loops, the inner loop loops again a number of times for the items in the array. So, if there are n values in the array, then a loop inside a loop is $n * n$. So, this is $O(n^2)$. That's polynomial, which ain't that good.

LeetCode.com

Some of the problems in the projects ask you to use the LeetCode platform to check your work rather than relying on local mocha tests. If you don't already have an account at LeetCode.com, please click https://leetcode.com/accounts/signup/ to sign up for a free account.

After you sign up for the account, please verify the account with the email address that you used so that you can actually run your solution on LeetCode.com.

In the projects, you will see files that are named "leet_code_«number».js". When you open those, you will see a link in the file that you can use to go directly to the corresponding problem on LeetCode.com.

Use the local JavaScript file in Visual Studio Code to collaborate on the solution. Then, you can run the proposed solution in the LeetCode.com code runner to validate its correctness.