

---

# Table of Contents

1.1 Introduction	1.1
1.2 Getting Started	1.2
1.3 Defining computer science	1.3
1.4 Defining programming	1.4
1.5 Why Study Data Structures and Abstract Data Types?	1.5
1.6 Why Study Algorithms?	1.6
1.7 Review of Basic JavaScript	1.7
1.8 Setup your environment	1.8
1.9 Getting Started with Data	1.9
1.10 Input and Output	1.10
1.11 Control Structures	1.11
1.12 Exception Handling	1.12
1.13 Defining Functions	1.13
1.14 Object-Oriented Programming in JavaScript: Defining Classes	1.14
2 Analysis	1.15
2.1 What Is Algorithm Analysis?	1.16
2.2 Big-O Notation	1.17
2.3 An Anagram Detection Example	1.18
2.4 Performance of JavaScript Data Structure	1.19
3 Basic Data Structures	1.20
3.1 What Are Linear Structures?	1.21
3.2 Stack	1.22
3.3 Queue	1.23
3.4 Deque	1.24
3.5 List	1.25
4 Recursion	1.26
4.1 What is Recursion?	1.27
4.2 Calculating the Sum of a List of Numbers	1.28
4.3 The Three Laws of Recursion	1.29
4.4 Converting an Integer to a String in Any Base	1.30

---

4.5 Stack Frames: Implementing Recursion	1.31
4.6 Introduction: Visualizing Recursion	1.32
4.7 Sierpinski Triangle	1.33
4.8 Complex Recursive Problems	1.34
4.9 Dynamic Programming	1.35
Arrays	1.36

---

# 1.1 Why this book?

As a curious JavaScript developer, I was an early adopter of NodeJS. As I was moving to server-side programming, I need to implement data structures like linked lists, stacks and queues and there associated algorithms. Since I am an autodidact without computer science degree, I encountered several issues. Back in 2009, they was nothing about data structure in JavaScript so I have to study it in classic object-oriented languages like Java and C#. Today, things are slowly changing. But I still see many JavaScript developer moving to the server-side, don't know nothing about data structures and why studying them is important. Through this guide, I will explain you what are the different data structures, their usages and implementation for server-side as well as their limitation due to JavaScript.

At the end, you will be able to chose the right data structure for the problems you have to solve.

# 1.2 Getting Started

As developers, we are living in a world in rapid evolution. This is especially true in the JavaScript ecosystem and its constant growing number of tools, frameworks and platforms. Nonetheless, the basic principles of computer science are still the foundation on which we stand when we use computers to solve problems.

You may have spend a l...

## 1.3 Defining computer science

Computer science is a discipline that spans theory and practice. In short, it's the study of problems, problem-solving, and the solutions that come out of the problem-solving process. The solution is called an **algorithm**. An algorithm is a step-by-step set of operations for solving a clearly defined problem.

However, consider the fact that some problems may not have a solution. This assumption mean that we have to redefine computer science as the study of solutions to problems as well as the study of problems with no solutions.

As we are talking about computer science as a problem-solving process, it mean we have to approach the study of **abstraction**. Abstraction allows us to separate problem and solution. Let's illustrate this separation with example:

Consider the legs that you may use when you wake up the morning and go to the bathroom. You get up and put one foot before the other to your destination with a success conditioned by the level of alcohol you ingested the night before. From an abstraction point of view, we can say that you are seeing the logical perspective of the legs. You are using the functions provided by the evolution for the purpose of transporting you from one location to another. These functions are sometimes also referred to as the interface.

On the other hand, the surgeon who must treat a leg takes a very different point of view. She not only knows how to walk but must know all of the details necessary to carry out all the functions that we take for granted. She needs to understand how the body works, how the blood feeds the muscles, the way nerves transmit impulses from the brain, and so on. This is known as the physical perspective, the details that take place “under the hood.”

The same thing happens when we use computers. Most people use computers to surf the web, send and receive emails, without any knowledge of the details that take place to allow those types of applications to work. They view computers from a logical or user perspective. Peoples behind the scene take a very different view of the computer. They must know the details of how operating systems work, how network protocols are configured, and how to code various scripts that control function. They must be able to control the low-level details that a user simply assumes.

As another example of abstraction, consider the `math.js` library. Once we import the module, we can perform computations such as

```
import math from mathjs  
  
math.sqrt(-4);           // 2i
```

This is an example of **procedural abstraction**. We do not necessarily know how the square root is being calculated, but we know what the function is called and how to use it. If we perform the import correctly, we can assume that the function will provide us with the correct results. We know that someone implemented a solution to the square root problem but we only need to know how to use it. This is sometimes referred to as a “black box” view of a process. We simply describe the interface: the name of the function, what is needed (the parameters), and what will be returned.

## 1.4 Defining programming

**Programming** is the process of taking an algorithm (the formulation of a computing problem) and encoding it into an executable computer program by using a programming language. Without an algorithm there can be no program.

An algorithm is a self-contained step-by-step set of operations to perform to produce the intended result. It also describe the data needed to represent the problem instance.

Programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them.

This control constructs allow algorithmic steps to be represented in a convenient and unambiguous way.

To be used for algorithm representation a programming language must provide the following basic instructions:

- Input: Get in data from a keyboard, a file, or another source.
- Output: Display data on the screen or to a file or by another way.
- Arithmetic: Perform basic arithmetical operations like addition and multiplication.
- Conditional Execution: Check for certain conditions and execute the appropriate sequence of statements.
- Repetition: Perform some action repeatedly, usually with some variation.

All data items in the computer are represented as strings of binary digits. In order to give these strings meaning, we need to have **data types**.

For example, most programming languages provide a data type for integers. Strings of binary digits in the computer's memory can be interpreted as integers and given the typical meanings that we commonly associate with integers (e.g. 23, 654, and -19). In addition, a data type also provides a description of the operations that the data items can participate in. With integers, operations such as addition, subtraction, and multiplication are common. We have come to expect that numeric types of data can participate in these arithmetic operations.

Unlike lower-level languages, JavaScript is a loosely typed/dynamic language. That means you don't have to declare the type of a variable ahead of time. The type will get determined automatically while the program is being processed. That also means that you can have the same variable as different types:

```
var foo = 42;    // foo is now a Number
var foo = "bar"; // foo is now a String
var foo = true;  // foo is now a Boolean
```

The difficulty that often arises for us is the fact that problems and their solutions are very complex. These simple, language-provided constructs and data types, although certainly sufficient to represent complex solutions, are typically at a disadvantage as we work through the problem-solving process. We need ways to control this complexity and assist with the creation of solutions.



# 1.5 Why Study Data Structures and Abstract Data Types?

When working on the problem-solving process, we need to stay focus on the big picture and avoid getting lost in the details. Abstractions are the perfect conceptual tool for that by allowing us to create models of the problem domain. A such model describe the data that our algorithms will manipulate. We called it a **data abstraction**.

An **abstract data type**, or **ADT**, is a logical description of how we view the data and the allowed operations without regard to how they will be implemented. By providing this level of abstraction, we are creating an encapsulation around the data. By encapsulating the details of the implementation, we are hiding them from our view. This is called **information hiding**. This allow us to remain focused on the problem-solving process.

# 1.6 Why Study Algorithms?

We learn by experience, by solving problems by ourselves or by following examples. With a deep understanding of how algorithms are designed, it helps us to take on the next challenge. We can begin to develop pattern recognition so that the next time a similar problem arises, we are better able to solve it. Just like we must remember the greatest number of combinations possible at cheese.

For a given problem, let say compute a square root, there is many different ways to implement the solution. We would like to have some way to compare these solutions in terms of time and resource consumption. By studying algorithms, we learn analysis techniques to do so based solely on their own characteristics. It mean separate them from the characteristics of the program or computer used to implement them.

The worst case scenario is a problem that is intractable. Meaning that there is no algorithm that can solve it in a realistic amount of time. It is important to be able to distinguish between those problems, those that have solutions and those that do not.

In the end, there are often many ways to solve a problem. There will often be trade-offs that we will need to identify and decide upon. This is a tasks that we will do over and over again.

## 1.7 Review of Basic JavaScript

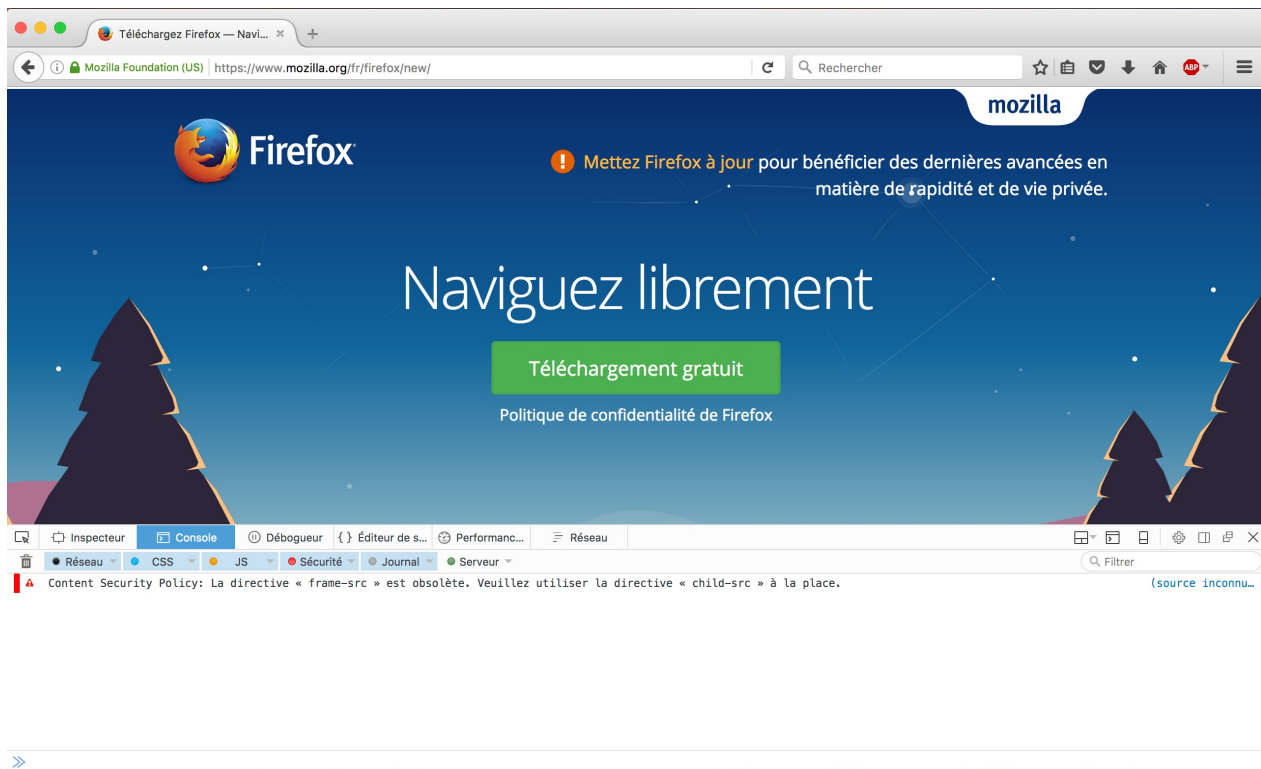
In this section, we will review the basis of this programming language. If you are new to JavaScript or need more information, don't hesitate to consult a resource such as the dedicated section on the [Mozilla Developer Network](#). The goal is to familiarize yourself with the language and reinforce the central concepts.

## 1.8 Setup your environment

These days, JavaScript is used both server and client side but it was originally made for a web browser usage only. The main difference is that this *web flavor* does not include any I/O, such as networking, storage, or graphics facilities. The *server flavor* however, is based on an event-driven architecture capable of asynchronous I/O.

### Work client side

Playing with JavaScript client side is pretty easy. Just open your web browser, Mozilla Firefox for example, and open the development tools. Use the `ctrl + shift + I` shortcut on Windows or `cmd + opt + I` on a Mac. You can do the same with Google Chrome.



Next, navigate to the `Console` tab. At the bottom, you have an input that lets you type JavaScript code. Your code will be directly interpreted by the browser and display the output. You can learn more about Firefox the console on MDN: [https://developer.mozilla.org/en-US/docs/Tools/Web\\_Console](https://developer.mozilla.org/en-US/docs/Tools/Web_Console).

### Work server side

Work client side will require extra steps. First, you need to install Node.js. You can find an installer for your OS on the official website <https://nodejs.org/en/download>. Next we will open the terminal (or the command line tool) and see how to execute JavaScript code. The procedure differ following your OS.

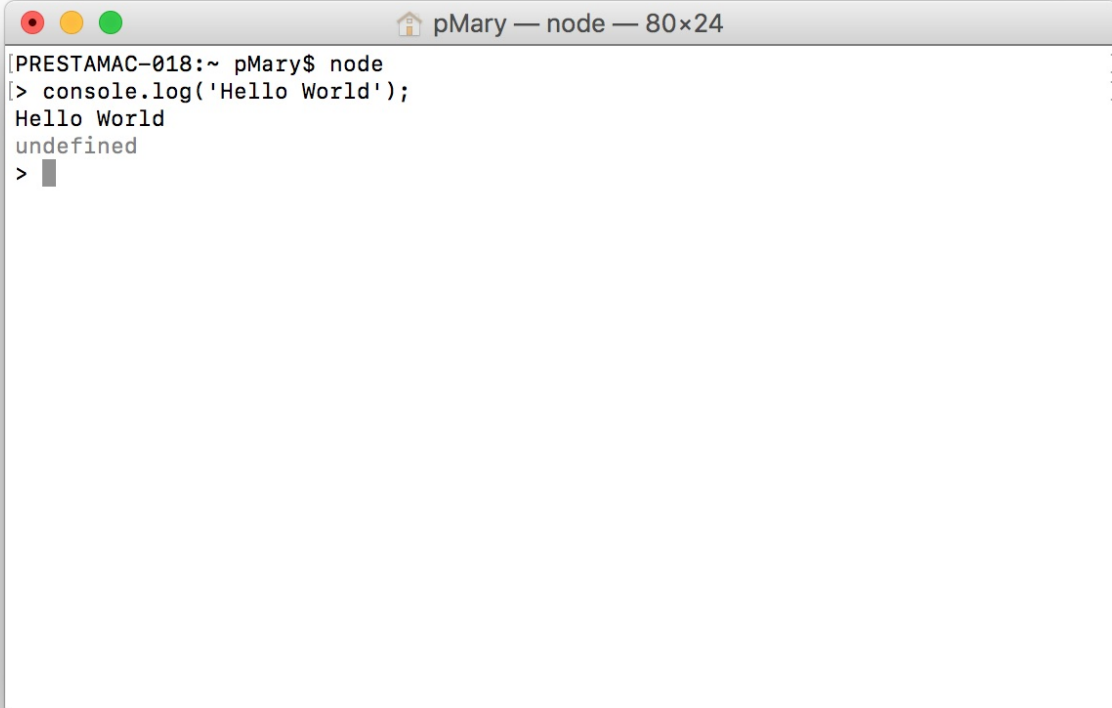
To open the Windows command line interpreter:

1. Click Start
2. In the Search or Run line type cmd and press enter.

To open the Mac terminal:

1. Use the `Cmd + Space` shortcut, type terminal and press enter.

Now you can type `node` and press enter. It will launch the Node.js console and everything you will type will be interpreted by Node.js. Try to type `console.log('Hello World')` for example:



```
[PRESTAMAC-018:~ pMary$ node
[> console.log('Hello World');
Hello World
undefined
> ]
```

The output will be the same as in the web browser console. You can also interpret files using this syntax:

```
$ node /my_file.js
```

You are now ready to work with Node.js!

## 1.9 Getting Started with Data

JavaScript is object-oriented to the core. It means that data is the focal point of the problem-solving process. This is also a prototype-based language and contains no class statement, such as is found in C++ or Java.

In JavaScript we don't have to define the type of a variable. This type is deducted from the value stored and may change as the program is running. It is said that JavaScript is a **dynamically** typed language. Other languages like C or Java required the definition of variable types. This is called **static** typing.

### 1.9.1 Built-in Data Types

According to the latest ECMAScript release, these are the seven data types:

- String
- Number
- Boolean
- Object
- Symbol (new in ECMAScript 6)
- Null
- Undefined

#### 1.9.1.1 Primitive values

##### String

A String value is a zero or more Unicode characters. They are useful for holding data that can be represented in text form.

Some of the most-used operations on strings are to check their **length**, to build and concatenate them using the **+** and **+=** **string operators**, checking for the existence or location of substrings with the **indexOf()** method, or extracting substrings with the **substring()** method.

You include string literals in your scripts by enclosing them in single or double quotation marks:

```
var myVar = 'Hello';  
var otherVar = "World";
```

They can also be created using the String global object directly:

```
var myVar = String(42);  
console.log(myVar); // "42"
```

## Number

In JavaScript, there is no distinction between integer and floating-point values; a JavaScript number can be either (internally, JavaScript represents all numbers as floating-point values). The standard arithmetic operations, +, -, \*, and / can be used with parentheses forcing the order of operations away from normal operator precedence.

Other very useful operation is the remainder (modulo) operator, %.

**Good to know:** The exponentiation operator \*\* is part of the ECMAScript 2016 (ES7), finalized in June 2016. It will eventually be implanted in all browsers but for the moment check the support before you use it.

```
console.log( 2+3*4 ); // 14  
console.log( (2+3)*4 ); //20  
console.log( 2**10 ); // 1024  
console.log( 6/3 ); // 2  
console.log( 7/3 ); // 2.3333333333333335  
console.log( 7%3 ); // 1  
console.log( 3/6 ); // 0.5  
console.log( 3%6 ); // 3  
console.log( 2**100 ); // 1.2676506002282294e+30
```

## Boolean

Pretty standard across all languages, booleans are `true` and `false`. They're often used for conditional statements.

```
var myBool = new Boolean(false);  
var myBool2 = true;
```

Since the variable type is dependent to it's value in JavaScript, the following will not be booleans:

```
var notABoolean1 = "true"; //string
var notABoolean2 = 1; //integer
```

## Symbol

This is new in ECMAScript 6, there is no EcmaScript 5 equivalent. A Symbol is a primitive data type whose instances are unique and immutable. In some programming languages they are also called atoms.

With Reflect, and Proxy, Symbol is one of the three new APIs that allow us to do metaprogramming.

You can find more about symbols on my blog: [Metaprogramming with ES2016 #1 - Symbol](#)

## Null

The value null is what you get when a variable has been defined but has no type or value. We can use it for exemple to assign a default value to a new variable:

```
var foo1 = null;
console.log(foo); // null
```

## Undefined

This is a very specific type returned when a variable is used without being declare before

Unfedefined is a very specific constant that is returned when a variable is not declared or when it is declared with no value assigned.

```
var foo;
console.log(foo); // undefined

console.log(bar); // throws a ReferenceError
```

### 1.9.1.2 Object

JavaScript is designed on a simple object-based paradigm. An object is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method. In addition to objects that are predefined in the browser, you can define your own objects.





# 1.10 Input and Output

We often have a need to interact with users, either to get data or to provide some sort of result. Most of the time we use an HTML form as a way of asking the user to provide some type of input. In our case, we will avoid this and use much simpler functions.

## 1.10.1 Client side

JavaScript provides us with a function that allows us to ask a user to enter some data and returns a reference to the data in the form of a string. The function is called `prompt()`.

A prompt dialog contains a single-line textbox, a Cancel button, and an OK button, and returns the (possibly empty) text the user entered into that textbox.

```
var fruit = prompt("What's your favorite fruit?");

if (fruit.toLowerCase() == "apple") {
    alert("Wow! I love apples too!");
}
```

## 1.10.2 Server side (Node.js)

On the server, we need to use the `readline` module. It provides an interface for reading data from a Readable stream (such as `process.stdin`) one line at a time.

```
const readline = require('readline');

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question("What's your favorite fruit? ", (answer) => {

    if (answer.toLowerCase() == "apple") {
        console.log("Wow! I love apples too!");
    }

    rl.close();
});
```

*Note* Once this code is invoked, the Node.js application will not terminate until the readline interface is closed because the interface waits for data to be received on the input stream.

## 1.10.3 String Formatting

As you may know, `console.log()` function provides a very simple way to output values from a JavaScript program.

This feature is non-standard so don't use it in production.

`console.log` takes zero or more parameters and displays them using a single blank as separator. In addition, each print ends with a newline character by default.

```
console.log('Hello'); // Hello
console.log('Hello', 'World'); // Hello World
```

It is often useful to have more control over the look of your output. Fortunately, Python provides us with an alternative called formatted strings. A formatted string is a template in which words or spaces that will remain constant are combined with placeholders for variables that will be inserted into the string. For example, the statement

```
var name = 'John';
var age = 42;
console.log(name, 'is', age, 'years old.');
```

contains the words `is` and `years old`, but the name and the age will change depending on the variable values at the time of execution. Using a formatted string, we write the previous statement as

```
var name = 'John';
var age = 42;
console.log('%s is %d years old.', name, age);
// John is 42 years old.
```

This simple example illustrates a new string expression. The `%` operator is a string operator called the **format operator**.

Character	Output Format
%s	String
%d , %i	Integer (numeric formatting is not yet supported)
%f , %.xf	Floating point number; x denotes the number of decimal places the number should be rounded to (if omitted, the number won't be rounded)
%o	Object hyperlink
%C	Style formatting

# 1.11 Control Structures

Algorithms require two important control structures: iteration and selection. Both of these are supported by JavaScript in various forms. The programmer can choose the statement that is most useful for the given circumstance.

For iteration, JavaScript provides a standard `while` statement and a very powerful `for` statement.

## 1.11.1 Iteration

### while loop

The `while` statement repeats a body of code as long as the condition is true.

```
var counter = 0;
while (counter < 3) {
  console.log('Counter at', counter);
  counter = counter + 1;
}
// Counter at 0
// Counter at 1
// Counter at 2
```

The condition on the `while` statement is evaluated at the start of each repetition. If the condition is `True`, the body of the statement will execute.

### do...while

In the previous example the statement block was executed only if the condition was true.

The `do...while` loop does the opposite

```
var counter = 0;
do {
  console.log('Counter at', counter);
  counter = counter + 1;
} while (counter < 3);
// Counter at 0
// Counter at 1
// Counter at 2
```

The block following `do` is executed first and then the condition is evaluated. If the while condition is true, the block is executed again and repeats until the condition becomes false. This is known as a *post-test* loop as the condition is evaluated after the block has executed.

## for loop

The most frequently used loop in JavaScript is the for-loop. It consists of three parts, separated by semicolons. The first is the *initializer* (`var i = 1`) which initializes the loop and is executed only once at the start. The second is a *test* condition (`i <= 50`). When a conditional expression evaluates to true, the body of the loop is executed. When false, the loop terminates. The third part is an *updater* (`i++`) which is invoked after each iteration. The updater typically increments or decrements the loop counter.

```
for (var i = 0; i < 3; i++) {  
    console.log(i);  
}  
// Counter at 0  
// Counter at 1  
// Counter at 2
```

## for-in loop

A for-in loop iterates through the properties of an object and executes the loop's body once for each enumerable property of the object.

```
var fruit = { name: 'Apple', kingdom: 'Plantae', family: 'Rosaceae' };  
for (var item in fruit) {  
    console.log(fruit[item]);  
}  
// Apple  
// Plantae  
// Rosoceleae
```

## break

When JavaScript encounters a break statement in a loop it immediately exits the loop without executing any other statements in the loop. Control is immediately transferred to the statement following the loop body.

```
for (var i = 0; i < 3; i++) {  
  if (i === 1) {  
    break;  
  }  
  console.log(i);  
}  
// 0
```

## continue

When JavaScript encounters a continue statement in a loop it stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration. The example below displays only even numbers:

```
for (var i = 1; i <= 10; i++) {  
  if ((i % 2) !== 0) {  
    continue;  
  }  
  console.log(i);  
}  
// 2  
// 4  
// 6  
// 8  
// 10
```

## 1.11.2 Selection

Selection statements allow programmers to ask questions and then, based on the result, perform different actions. Most programming languages provide two versions of this useful construct: the `ifelse` and the `if`. A simple example of a binary selection uses the `ifelse` statement.

### if

```
if (i < 0){  
  console.log('Sorry, value is negative');  
} else {  
  console.log( Math.sqrt(i) );  
}
```





# 1.12 Exception Handling

There are two types of errors that typically occur when writing programs. The first is known as a **syntax error**. It means that the programmer has made a mistake in the structure of a statement or expression. The other type of error is a **logic error**, denotes a situation where the program executes but gives the wrong result. For example, a user enter a string rather than a number, resulting in invalid results without raising any error.

## 1.12.1 Syntax error

```
var foo = 'Tom's bar';  
// Uncaught SyntaxError: Unexpected identifier
```

As we can see in the above code, the string is not properly escaped, leading to a runtime error that causes the program to terminate. These types of runtime errors are typically called **exceptions**.

Most of the time, beginners think of exceptions as fatal runtime errors that cause the end of execution. However, most programming languages provide a way to deal with these errors that will allow the programmer to have some type of intervention if they so choose. In addition, programmers can create their own exceptions if they detect a situation in the program execution that warrants it.

When an exception occurs, we say that it has been "raised". You can "handle the exception that has been raised by using a `try` statement. For example, consider a program that asks the user for an integer and then calculate its square root. If the user enters a negative value, the square root function will report a 'ValueError' exception.

## 1.12.2 Logic error

This can be due to an error in the underlying algorithm or an error in your translation of that algorithm.

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Enter an integer?', (num) => {

  var squareRoot = Math.sqrt(num);
  console.log('The square root of', num, 'is', squareRoot);

  rl.close();
});
```

In the case of the above code, if you enter a string or a negative number, the result will be `NaN` or *Not a Number* but no exception will occur.

## 1.13 Defining Functions

Generally, we can hide the details of any computation by defining a function. For example, earlier we use the `Math.sqrt()` function that act as a blackbox. You give it a number and it returns its square root. A function definition requires a name, a group of parameters, and a body. It may optionally return an output. A function in JavaScript is also an object. For example, the simple function defined below returns the square of the value you pass into it:

```
function square (n) {  
    return n**2  
}  
  
console.log( square(3) ); // 9
```

Because our function has a name, we call it a **named function**. At the opposite an anonymous function is a function without a function name:

```
function () {}
```

A function inside another function is an **inner function** (square in the exemple below). So an **outer function** is a function containing a function (addSquares in this case):

```
function addSquares(a,b) {  
    function square(n) {  
        return n**2;  
    }  
    return square(a) + square(b);  
}
```

A **recursive function** is a function that calls itself:

```
function loop(x) {  
    if (x >= 10)  
        return;  
    loop(x + 1);  
}
```

We can also define an **Immediately Invoked Function Expressions** (IIFE). This kind of function is called directly after being loaded into the browser's compiler.

We could implement our own square root function by using a well-known technique called "Newton's Method". Newton's Method for approximating square roots performs an iterative computation that converges on the correct value. The equation  $\text{newguess} = 1/2 * (\text{oldguess} + n/\text{oldguess})$  takes a value  $n$  and repeatedly guesses the square root by making each  $\text{newguess}$  the  $\text{oldguess}$  in the subsequent iteration. The initial guess used here is  $n/2$ . The function below shows a definition that accepts a value  $n$  and returns the square root of  $n$  after making 20 guesses. Again, the details of Newton's Method are hidden inside the function definition and the user does not have to know anything about the implementation to use the function for its intended purpose. It also shows the use of the `//` characters as a comment marker. Any characters that follow the `//` on a line are ignored.

```
function squareroot (n) {  
  root = n/2; // Initial guess will be 1/2 of n  
  for (var i = 0; i < 20; i++) {  
    root = (1/2) * (root + (n / root));  
  }  
  
  return root;  
}  
  
console.log( squareroot(9) ); // 3  
console.log( squareroot(4563) ); // 67.54998149518622
```

## 1.14 Object-Oriented Programming in JavaScript: Defining Classes

Object-oriented to the core, JavaScript features powerful, flexible OOP capabilities. So far, we have used a number of built-in classes to show examples of data and control structures. One of the most powerful features in an object-oriented programming language is the ability to allow a programmer (problem solver) to create new classes that model data that is needed to solve the problem.

Remember that we use abstract data types to provide the logical description of what a data object looks like (its state) and what it can do (its methods). By building a class that implements an abstract data type, a programmer can take advantage of the abstraction process and at the same time provide the details necessary to actually use the abstraction in a program. Whenever we want to implement an abstract data type, we will do so with a new class.

### 1.14.1. A Fraction Class

A very common example to show the details of implementing a user-defined class is to construct a class to implement the abstract data type `Fraction`.

A fraction such as  $\frac{3}{5}$  consists of two parts. The top value, known as the numerator, can be any integer. The bottom value, called the denominator, can be any integer greater than 0 (negative fractions have a negative numerator). Although it is possible to create a floating point approximation for any fraction, in this case we would like to represent the fraction as an exact value.

The operations for the `Fraction` type will allow a `Fraction` data object to behave like any other numeric value. We need to be able to add, subtract, multiply, and divide fractions. We also want to be able to show fractions using the standard “slash” form, for example  $\frac{3}{5}$ . In addition, all fraction methods should return results in their lowest terms so that no matter what computation is performed, we always end up with the most common form.

In JavaScript ES2016, we define a new class by providing a name, a constructor and a set of method definitions that are syntactically similar to function definitions.

```
class Fraction {  
  constructor() {  
  }  
  // The methods go here  
}
```

The first method that all classes should provide is the constructor. This is a special method with the name "constructor" for creating and initializing an object created with a class. To create a `Fraction` object, we will need to provide two pieces of data, the numerator and the denominator.

```
class Fraction {  
  constructor(top, bottom) {  
    this.num = top;  
    this.den = bottom;  
  }  
}
```

As described earlier, fractions require two pieces of state data, the numerator and the denominator. The notation `this.num` in the constructor defines the fraction object to have an internal data object called `num` as part of its state. Likewise, `this.den` creates the denominator. The values of the two formal parameters are initially assigned to the state, allowing the new `fraction` object to know its starting value.

To create an instance of the `Fraction` class, we must invoke the constructor. This happens by using the name of the class and passing actual values for the necessary state (note that we never directly invoke `constructor`). For example:

```
var myFraction = new Fraction(3, 5);
```

It creates an object called `myFraction` representing the fraction 3/5 (three-fifths). The next thing we need to do is implement the behavior that the abstract data type requires. To begin, consider what happens when we try to print a `Fraction` object.

```
var myFraction = new Fraction(3, 5);  
console.log(myFraction);  
// Fraction {num: 3, den: 5}
```

What it does is printing a reference to the object but we may want to print the fraction state in the proper format. For this, we will create a `show` method that allows us to view the fraction using the standard "slash" form.

```
class Fraction {
  constructor(top, bottom) {
    this.num = top;
    this.den = bottom;
  }

  show() {
    console.log(`${this.num}/${this.den}`);
  }
}

var myFraction = new Fraction(3, 5);
myFraction.show();
// 3/5
```

Next we would like to be able to create two `Fraction` objects and then add them together using the standard "+" notation. At this point, if we try to add two fractions, we get the following:

```
var f1 = new Fraction(1, 2);
var f2 = new Fraction(1, 4);
f1+f2;
"[object Object][object Object]"
```

Since we cannot overload operators in JavaScript, we can fix this by providing the `Fraction` class with a new `plus` method:

```
plus(fraction) {
  let newnum = this.num * fraction.den + this.den * fraction.num;
  let newden = this.den * fraction.den;

  return new Fraction(newnum, newden);
}
```

```
var f1 = new Fraction(1, 2);
var f2 = new Fraction(1, 4);
f3 = f1.plus(f2);
f3.show();
// 6/8
```

The `plus` method works as we desire, but one thing could be better. Note that 6/8 is the correct result ( $1/4 + 1/2$ ) but that it is not in the "lowest terms" representation. The best representation would be 3/4. In order to be sure that our results are always in the lowest

terms, we need a helper function that knows how to reduce fractions. This function will need to look for the greatest common divisor, or GCD. We can then divide the numerator and the denominator by the GCD and the result will be reduced to lowest terms.

The best-known algorithm for finding a greatest common divisor is Euclid's Algorithm. It states that the greatest common divisor of two integers  $m$  and  $n$  is  $n$  if  $n$  divides  $m$  evenly. However, if  $n$  does not divide  $m$  evenly, then the answer is the greatest common divisor of  $n$  and the remainder of  $m$  divided by  $n$ . So we will simply provide an iterative implementation of it.

Note that this implementation of the GCD algorithm only works when the denominator is positive. This is acceptable for our fraction class because we have said that a negative fraction will be represented by a negative numerator.

```
gcd(n = this.num, d = this.den) {
  while (m%n !== 0) {
    let oldm = m;
    let oldn = n;

    m = oldn;
    n = oldm%oldn;
  }

  return n;
}
```

```
var f1 = new Fraction(20, 10);
console.log( f1.gcd() );
// 10
```

Now we can use this function to help reduce any fraction. To put a fraction in lowest terms, we will divide the numerator and the denominator by their greatest common divisor. So, for the fraction 6/8, the greatest common divisor is 2. Dividing the top and the bottom by 2 creates a new fraction, 3/4:

```
plus(fraction) {
  let newnum = this.num * fraction.den + this.den * fraction.num;
  let newden = this.den * fraction.den;
  let common = this.gcd(newnum, newden);

  return new Fraction( Math.trunc(newnum/common), Math.trunc(newden/common) );
}
```



```
var f1 = new Fraction(1, 4);
var f2 = new Fraction(1, 2);
var f3 = f1.plus(f2);
f3.show();
// 3/4
```

`Math.trunc()` is a new ES6 function. It work like `Math.floor()` but work for negative numbers too.

Our `Fraction` object now has two very useful methods. An additional group of methods that we need to include in our example `Fraction` class will allow two fractions to compare themselves to one another. Assume we have two `Fraction` objects, `f1` and `f2`. `f1==f2` will only be `True` if they are references to the same object. Two different objects with the same numerators and denominators would not be equal under this implementation. This is called **shallow equality**.

```
var f1 = new Fraction(1, 4);
var f2 = new Fraction(1, 4);
f1==f2;
// false

var f1 = new Fraction(1, 4);
var f2 = f1;
f1==f2;
// true
```

We can create **deep equality** by the same value, not the same reference. So we will create a new method called `equal` to compares two objects and returns `True` if their values are the same, `False` otherwise.

```
equal(fraction) {
  let firstNum = this.num * fraction.den;
  let secondNum = fraction.num * this.den;

  return firstNum == secondNum;
}
```

```
var f1 = new Fraction(1, 4);
var f2 = new Fraction(1, 4);
f1.equal(f2);
// true

var f1 = new Fraction(1, 4);
var f2 = new Fraction(2, 8);
f1.equal(f2);
// true
```

Let's wrap it together:

```
class Fraction {
  constructor(top, bottom) {
    this.num = top;
    this.den = bottom;
  }

  plus(fraction) {
    let newnum = this.num * fraction.den + this.den * fraction.num;
    let newden = this.den * fraction.den;
    let common = this.gcd(newnum, newden);

    return new Fraction( Math.trunc(newnum/common), Math.trunc(newden/common) );
  }

  gcd(n = this.num, d = this.den) {
    while (n%d !== 0) {
      let oldm = n;
      let oldn = d;

      n = oldn;
      d = oldm%oldn;
    }

    return d;
  }

  equal(fraction) {
    let firstNum = this.num * fraction.den;
    let secondNum = fraction.num * this.den;

    return firstNum == secondNum;
  }

  show() {
    console.log(`${this.num}/${this.den}`);
  }
}
```

To make sure you understand how operators are implemented in Python classes, and how to properly write methods, write some methods to implement `*`, `/`, and `-`. Also implement comparison operators `>` and `<`.

## 1.14.2 Inheritance: Logic Gates and Circuits

Our final section will introduce another important aspect of object-oriented programming.

**Inheritance** is the ability for one class to be related to another class in much the same way that people can be related to one another. Children inherit characteristics from their parents. But in Javascript, there is no `class` implementation. The `class` keyword introduced in ES6 is just a syntactical sugar. JavaScript remains prototype-based. When it comes to inheritance, JavaScript only has one construct: objects. Each object has an internal link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. null, by definition, has no prototype, and acts as the final link in this **prototype chain**.

While this is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model is in fact more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model. We refer to a child classes as **subclasses** and **superclasses**.

By organizing classes in this hierarchical fashion, object-oriented programming languages allow previously written code to be extended to meet the needs of a new situation. In addition, by organizing data in this hierarchical manner, we can better understand the relationships that exist. We can be more efficient in building our abstract representations.

To explore this idea further, we will construct a **simulation**, an application to simulate digital circuits. The basic building block for this simulation will be the logic gate. These electronic switches represent boolean algebra relationships between their input and their output. In general, gates have a single output line. The value of the output is dependent on the values given on the input lines.

AND gates have two input lines, each of which can be either 0 or 1 (representing `False` or `True`, respectively). If both of the input lines have the value 1, the resulting output is 1. However, if either or both of the input lines is 0, the result is 0.

OR gates also have two input lines and produce a 1 if one or both of the input values is a 1. In the case where both input lines are 0, the result is 0.

NOT gates differ from the other two gates in that they only have a single input line. The output value is simply the opposite of the input value. If 0 appears on the input, 1 is produced on the output. Similarly, 1 produces 0.

By combining these gates in various patterns and then applying a set of input values, we can build circuits that have logical functions. In order to implement a circuit, we will first build a representation for logic gates. Logic gates are easily organized into a class inheritance hierarchy. At the top of the hierarchy, the `LogicGate` class represents the most general characteristics of logic gates: namely, a label for the gate and an output line. The next level of subclasses breaks the logic gates into two families, those that have one input line and those that have two.

We can now start to implement the classes by starting with the most general, `LogicGate`. As noted earlier, each gate has a label for identification and a single output line. In addition, we need methods to allow a user of a gate to ask the gate for its label.

The other behavior that every logic gate needs is the ability to know its output value. This will require that the gate perform the appropriate logic based on the current input. In order to produce output, the gate needs to know specifically what that logic is. This means calling a method to perform the logic computation. Here is the complete class:

```
class LogicGate {
  constructor(n) {
    this.label = n;
    this.output = null;
  }

  getLabel() {
    return this.label;
  }

  getOutput() {
    this.output = this.performGateLogic();
    return this.output;
  }
}
```

At this point, we will not implement the `performGateLogic` function. The reason for this is that we do not know how each gate will perform its own logic operation. Those details will be included by each individual gate that is added to the hierarchy. This is a very powerful idea in object-oriented programming. We are writing a method that will use code that does not exist yet. The parameter `this` is a reference to the actual gate object invoking the method. Any new logic gate that gets added to the hierarchy will simply need to implement the `performGateLogic` function and it will be used at the appropriate time. Once done, the gate can provide its output value. This ability to extend a hierarchy that currently exists and provide the specific functions that the hierarchy needs to use the new class is extremely important for reusing existing code.

We categorized the logic gates based on the number of input lines. The AND gate has two input lines. The OR gate also has two input lines. NOT gates have one input line. The `BinaryGate` class will be a subclass of `LogicGate` and will add two input lines. The `UnaryGate` class will also subclass `LogicGate` but will have only a single input line. In computer circuit design, these lines are sometimes called "pins" so we will use that terminology in our implementation.

```
class BinaryGate extends LogicGate {
  constructor(n) {
    this.pinA = null;
    this.pinB = null;
  }

  getPinA() {
    return prompt(`Enter Pin input for gate ${this.getLabel()}-->`);
  }

  getPinB() {
    return prompt(`Enter Pin input for gate ${this.getLabel()}-->`);
  }
}
```

```
class UnaryGate(LogicGate) {
  constructor() {
    this.pin = null;
  }

  getPin() {
    return prompt(`Enter Pin input for gate ${this.getLabel()}-->`);
  }
}
```

Here we implement the two classes. The constructors in both of these classes start with an explicit call to the constructor of the parent class using the parent's `constructor` method. When creating an instance of the `BinaryGate` class, we first want to initialize any data items that are inherited from `LogicGate`. In this case, that means the label for the gate. The constructor then goes on to add the two input lines ( `pinA` and `pinB` ). This is a very common pattern that you should always use when building class hierarchies. Child class constructors need to call parent class constructors and then move on to their own distinguishing data.

JavaScript also have a keyword called `super` and used to call functions on an object's parent. It must be used before the `this` keyword can be used. This keyword can also be used to call functions on a parent object.

The only behavior that the `BinaryGate` class adds is the ability to get the values from the two input lines. Since these values come from some external place, we will simply ask the user via an input statement to provide them. The same implementation occurs for the `UnaryGate` class except that there is only one input line.

Now that we have a general class for gates depending on the number of input lines, we can build specific gates that have unique behavior. For example, the `AndGate` class will be a subclass of `BinaryGate` since AND gates have two input lines. As before, the first line of the constructor calls upon the parent class constructor ( `BinaryGate` ), which in turn calls its parent class constructor ( `LogicGate` ). Note that the `AndGate` class does not provide any new data since it inherits two input lines, one output line, and a label.

```
class AndGate(BinaryGate) {
  constructor(n) {
    super(n);
  }

  performGateLogic() {
    a = this.getPinA()
    b = this.getPinB()
    if (a==1 && b==1) {
      return 1;
    }
    else {
      return 0;
    }
  }
}
```

The only thing `AndGate` needs to add is the specific behavior that performs the boolean operation that was described earlier. This is the place where we can provide the `performGateLogic` method. For an AND gate, this method first must get the two input values and then only return 1 if both input values are 1.

We can show the `AndGate` class in action by creating an instance and asking it to compute its output. The following session shows an `AndGate` object, `g1`, that has an internal label `"G1"`. When we invoke the `getOutput` method, the object must first call its `performGateLogic` method which in turn queries the two input lines. Once the values are provided, the correct output is shown.

```
g1 = AndGate("G1");
g1.getOutput();
// Enter Pin A input for gate G1-->1
// Enter Pin B input for gate G1-->0
0
```

The same development can be done for OR gates and NOT gates. The `OrGate` class will also be a subclass of `BinaryGate` and the `NotGate` class will extend the `UnaryGate` class. Both of these classes will need to provide their own `performGateLogic` functions, as this is their specific behavior.

We can use a single gate by first constructing an instance of one of the gate classes and then asking the gate for its output (which will in turn need inputs to be provided). For example:

```
g2 = OrGate("G2");
g2.getOutput();
// Enter Pin A input for gate G2-->1
// Enter Pin B input for gate G2-->1
1

g2.getOutput();
// Enter Pin A input for gate G2-->0
// Enter Pin B input for gate G2-->0
0

g3 = NotGate("G3");
g3.getOutput();
//Enter Pin input for gate G3-->0
1
```

Now that we have the basic gates working, we can turn our attention to building circuits. In order to create a circuit, we need to connect gates together, the output of one flowing into the input of another. To do this, we will implement a new class called `Connector`.

The `Connector` class will not reside in the gate hierarchy. It will, however, use the gate hierarchy in that each connector will have two gates, one on either end. It is called the **HAS-A Relationship**. Recall earlier that we used the phrase **"IS-A Relationship"** to say that a child class is related to a parent class, for example `UnaryGate` IS-A `LogicGate`.

**has-a (has\_a or has a)** is a composition relationship where one object (often called the constituted object, or part/constituent/member object) "belongs to" (is part or member of) another object (called the composite type), and behaves according to the rules of ownership. In simple words, has-a relationship in an object is called a member field of an object. Multiple **has-a** relationships will combine to form a possessive hierarchy.

Now, with the `Connector` class, we say that a `Connector` HAS-A `LogicGate` meaning that connectors will have instances of the `LogicGate` class within them but are not part of the hierarchy. When designing classes, it is very important to distinguish between those that have the IS-A relationship (which requires inheritance) and those that have HAS-A relationships (with no inheritance).

The following sample show the `Connector` class. The two gate instances within each connector object will be referred to as the `fromGate` and the `toGate`, recognizing that data values will “flow” from the output of one gate into an input line of the next. The call to `setNextPin` is very important for making connections. We need to add this method to our gate classes so that each `toGate` can choose the proper input line for the connection.

```
class Connector() {
  constructor(fGate, tGate) {
    this.fromGate = fGate;
    this.toGate = tGate;

    tGate.setNextPin(this);
  }

  getFrom() {
    return this.fromGate;
  }

  getTo(self) {
    return self.toGate;
  }
}
```

In the `BinaryGate` class, for gates with two possible input lines, the connector must be connected to only one line. If both of them are available, we will choose `pinA` by default. If `pinA` is already connected, then we will choose `pinB`. It is not possible to connect to a gate with no available input lines.

```
setNextPin(source) {
  if (this.pinA == null) {
    this.pinA = source;
  }
  else {
    if (this.pinB == null) {
      this.pinB = source;
    }
    else {
      throw new Error("Error: NO EMPTY PINS");
    }
  }
}
```

Now it is possible to get input from two places: externally, as before, and from the output of a gate that is connected to that input line. This requires a change to the `getPinA` and `getPinB` methods (see the sample at the end of the chapter). If the input line is not connected to anything (`null`), then ask the user externally as before. However, if there is a



connection, the connection is accessed and `fromGate` 's output value is retrieved. This in turn causes that gate to process its logic. This continues until all input is available and the final output value becomes the required input for the gate in question. In a sense, the circuit works backwards to find the input necessary to finally produce output.

```
getPinA() {
    if (this.pinA == null) {
        return prompt(`Enter Pin A input for gate ${this.getLabel()}-->`);
    }
    else {
        return this.pinA.getFrom().getOutput();
    }
}
```

The following fragment show an example of circuit we can now constructs:

```
g1 = new AndGate("G1");
g2 = new AndGate("G2");
g3 = new OrGate("G3");
g4 = new NotGate("G4");
c1 = new Connector(g1, g3);
c2 = new Connector(g2, g3);
c3 = new Connector(g3, g4);
```

The outputs from the two AND gates ( `g1` and `g2` ) are connected to the OR gate ( `g3` ) and that output is connected to the NOT gate (`g4`). The output from the NOT gate is the output of the entire circuit. For example:

```
g4.getOutput();
// Pin A input for gate G1-->0
// Pin B input for gate G1-->1
// Pin A input for gate G2-->1
// Pin B input for gate G2-->1
0
```

Here is the complete sample:

```
class LogicGate {
    constructor(n) {
        this.name = n;
        this.output = null;
    }

    getLabel() {
        return this.name;
    }
}
```

```

    getOutput() {
        this.output = this.performGateLogic();
        return this.output;
    }
}

class BinaryGate extends LogicGate {
    constructor(n) {
        super(n);

        this.pinA = null;
        this.pinB = null;
    }

    getPinA() {
        if (this.pinA == null) {
            return parseInt(prompt(`Enter Pin A input for gate ${this.getLabel()}-->`));
        }
        else {
            return this.pinA.getFrom().getOutput();
        }
    }

    getPinB() {
        if (this.pinB == null) {
            return parseInt(prompt(`Enter Pin A input for gate ${this.getLabel()}-->`));
        }
        else {
            return this.pinB.getFrom().getOutput();
        }
    }

    setNextPin(source) {
        if (this.pinA == null){
            this.pinA = source;
        }
        else {
            if (this.pinB == null) {
                this.pinB = source;
            }
            else {
                throw new Error("Cannot Connect: NO EMPTY PINS on this gate");
            }
        }
    }
}

class AndGate extends BinaryGate {
    constructor(n) {
        super(n);
    }
}

```

```
}

performGateLogic() {
    var a = this.getPinA();
    var b = this.getPinB();
    if (a == 1 && b == 1) {
        return 1;
    }
    else {
        return 0;
    }
}
}

class OrGate extends BinaryGate {
    constructor(n) {
        super(n);
    }

    performGateLogic() {
        var a = this.getPinA();
        var b = this.getPinB();
        if (a == 1 || b == 1) {
            return 1;
        }
        else {
            return 0;
        }
    }
}

class UnaryGate extends LogicGate {
    constructor(n) {
        super(n);

        this.pin = null;
    }

    getPin() {
        if (this.pin == null) {
            return parseInt(prompt(`Enter Pin input for gate ${this.getLabel()}-->`));
        }
        else {
            return this.pin.getFrom().getOutput();
        }
    }

    setNextPin(source) {
        if (this.pin == null) {
            this.pin = source;
        }
        else {
            throw new Error("Cannot Connect: NO EMPTY PINS on this gate");
        }
    }
}
```

```
    }  
  }  
}  
  
class NotGate extends UnaryGate {  
  constructor(n) {  
    super(n);  
  }  
  
  performGateLogic() {  
    if (this.getPin()) {  
      return 0;  
    }  
    else {  
      return 1;  
    }  
  }  
}  
  
class Connector {  
  constructor(fGate, tGate) {  
    this.fromGate = fGate;  
    this.toGate = tGate;  
  
    tGate.setNextPin(this);  
  }  
  
  getFrom() {  
    return this.fromGate;  
  }  
  
  getTo() {  
    return this.toGate;  
  }  
}  
  
function main() {  
  g1 = new AndGate("G1");  
  g2 = new AndGate("G2");  
  g3 = new OrGate("G3");  
  g4 = new NotGate("G4");  
  c1 = new Connector(g1, g3);  
  c2 = new Connector(g2, g3);  
  c3 = new Connector(g3, g4);  
  console.log(g4.getOutput());  
}  
  
main();
```



# Analysis

## Objectives

- To understand why algorithm analysis is important.
- To be able to use "Big-O" to describe execution time.
- To understand the "Big-O" execution time of common operations on JavaScript arrays and objects.
- To understand how the implementation of JavaScript data impacts algorithm analysis.
- To understand how to benchmark simple JavaScript programs.

## Chapters

- [2.1 What is Algorithm Analysis](#)
- [2.2 Big-O Notation](#)
- [2.3 An Anagram Detection Example](#)
- [2.4 Performance of JavaScript Data Structure](#)

## 2.1 What is Algorithm Analysis

If you ask two developers to solve the same problem you will certainly end with two different programs. This raises an interesting question. How do you know which one is better than the other?

In order to answer this question, we need to remember that there is an important difference between a program and the underlying algorithm that the program is representing. As we stated in Chapter 1, an algorithm is a generic, step-by-step list of instructions for solving any instance of a problem. A program, on the other hand, is an algorithm that has been encoded into some programming language. There may be many programs for the same algorithm, depending on the programmer and the programming language being used.

To explore this difference further, consider the function shown below. This function solves a familiar problem, computing the sum of the first  $n$  integers. The algorithm uses the idea of an accumulator variable that is initialized to 0. The solution then iterates through the  $n$  integers, adding each to the accumulator.

```
function sumOfN(n) {  
  var theSum = 0  
  for (var i = 0; i < (n + 1); i++) {  
    theSum = theSum + i;  
  }  
  
  return theSum;  
}  
  
console.log(sumOfN(10));
```

Now look at the function below. At first glance it may look strange, but upon further inspection you can see that this function is essentially doing the same thing as the previous one. The reason this is not obvious is poor coding. We did not use good identifier names to assist with readability, and we used an extra assignment statement during the accumulation step that was not really necessary.

```
function foo(tom) {  
  var fred = 0;  
  for (var bill = 0; bill < (tom + 1); bill++) {  
    var barney = bill;  
    fred = fred + barney;  
  }  
  
  return fred;  
}  
  
console.log(foo(10));
```

The question we raised earlier asked whether one function is better than another. The answer depends on your criteria. The function `sumOfN` is certainly better than the function `foo` if you are concerned with readability. In fact, you have probably seen many examples of this in your introductory programming course since one of the goals there is to help you write programs that are easy to read and easy to understand. In this course, however, we are also interested in characterizing the algorithm itself. (We certainly hope that you will continue to strive to write readable, understandable code).

Algorithm analysis is concerned with comparing algorithms based upon the amount of computing resources that each algorithm uses. We want to be able to consider two algorithms and say that one is better than the other because it is more efficient in its use of those resources or perhaps because it simply uses fewer. From this perspective, the two functions above seem very similar. They both use essentially the same algorithm to solve the summation problem.

At this point, it is important to think more about what we really mean by computing resources. There are two different ways to look at this. One way is to consider the amount of space or memory an algorithm requires to solve the problem. The amount of space required by a problem solution is typically dictated by the problem instance itself. Every so often, however, there are algorithms that have very specific space requirements, and in those cases we will be very careful to explain the variations.

As an alternative to space requirements, we can analyze and compare algorithms based on the amount of time they require to execute. This measure is sometimes referred to as the "execution time" or "running time" of the algorithm. One way we can measure the execution time for the function `Á` is to do a benchmark analysis. This means that we will track the actual time required for the program to compute its result. In JavaScript, we can benchmark a function by noting the starting time and ending time with respect to the system we are using. We can use the `performance.now()` method to returns a DOMHighResTimeStamp,



measured in milliseconds, accurate to one thousandth of a millisecond. By calling this function twice, at the beginning and at the end, and then computing the difference, we can get an exact number of seconds (fractions in most cases) for execution.

Server side, `performance.now()` is not available. You should use `process.hrtime()` instead. It returns the current high-resolution real time in a [seconds, nanoseconds] tuple Array.

```
function sumOfN(n) {
  var start = performance.now();
  var theSum = 0

  for (var i = 0; i < (n + 1); i++) {
    theSum = theSum + i;
  }

  var end = performance.now();
  console.log(`Computed in ${end-start} milliseconds`);
  return theSum;
}

console.log(sumOfN(10));
// Computed in 0.05500000016763806 milliseconds
```

Node.JS version

```
function sumOfN(n) {
  var start = process.hrtime();
  var theSum = 0

  for (var i = 0; i < (n + 1); i++) {
    theSum = theSum + i;
  }

  var end = process.hrtime(start);
  end = end[0] * 1000000 + end[1] / 1000;
  console.log(`Computed in ${end} milliseconds`);
  return theSum;
}

console.log(sumOfN(10));
// Computed in 128.915 milliseconds
```

In the code above you can see that the `sumOfN` function now log the time elapsed between the start and then end of it's execution. If we perform 5 invocations of the function, each computing the sum of the first 10,000 integers, we get the following:

## 2.1 What Is Algorithm Analysis?

```
for (let i = 0; i < 5; i++) {  
    sumOfN(10000);  
}  
// Computed in 14.96 milliseconds  
// Computed in 14.791 milliseconds  
// Computed in 13.868 milliseconds  
// Computed in 13.997 milliseconds  
// Computed in 14.462 milliseconds
```

We discover that the time is fairly consistent and it takes on average about 14.4156 milliseconds to execute that code. What if we run the function adding the first 100,000 integers?

```
for (let i = 0; i < 5; i++) {  
    sumOfN(100000);  
}  
// Computed in 152.101 milliseconds  
// Computed in 174.257 milliseconds  
// Computed in 153.963 milliseconds  
// Computed in 158.269 milliseconds  
// Computed in 153.325 milliseconds
```

In this case, the average is 158.3829, about 10 times the previous. For  $n$  equal to 1,000,000 we get:

```
for (let i = 0; i < 5; i++) {  
    sumOfN(1000000);  
}  
// Computed in 1451.723 milliseconds  
// Computed in 1487.598 milliseconds  
// Computed in 1498.227 milliseconds  
// Computed in 1436.73 milliseconds  
// Computed in 1459.882 milliseconds
```

This time we found the average is 1466.8319, again, it's close to 10 times the previous.

Now, consider the code below. It shows a different means of solving the summation problem. This function, takes advantage of a closed equation  $\sum_{i=1}^n i = \frac{(n)(n+1)}{2}$  to compute the sum of the first  $n$  integers without iterating.

```
function sumOfN2(n) {  
  var start = performance.now();  
  
  var result = (n*(n+1))/2;  
  
  var end = performance.now();  
  
  console.log(`Computed in ${end-start} milliseconds`);  
  
  return result;  
}  
  
console.log(sumOfN2(10));
```

If we do the same benchmark measurement for `sumOfN2`, using five different values for `n` (10,000, 100,000, 1,000,000, 10,000,000, and 100,000,000), we get the following results:

```
// 10,000 => 0.4638ms  
// 100,000 => 0.5874ms  
// 1,000,000 => 0.5040ms  
// 10,000,000 => 0.5166ms  
// 100,000,000 => 0.4958ms
```

There are two important things to notice about this output. First, the times recorded above are shorter than any of the previous examples. Second, they are very consistent no matter what the value of `n`. It appears that `sumOfN2` is hardly impacted by the number of integers being added.

But what does this benchmark really tell us? Intuitively, we can see that the iterative solutions seem to be doing more work since some program steps are being repeated. This is likely the reason it is taking longer. Also, the time required for the iterative solution seems to increase as we increase the value of `n`. However, there is a problem. If we ran the same function on a different computer or used a different programming language, we would likely get different results. It could take even longer to perform `sumOfN2` if the computer were older.

We need a better way to characterize these algorithms with respect to execution time. The benchmark technique computes the actual time to execute. It does not really provide us with a useful measurement, because it is dependent on a particular machine, program, time of day, compiler, and programming language. Instead, we would like to have a characterization that is independent of the program or computer being used. This measure would then be useful for judging the algorithm alone and could be used to compare algorithms across implementations.



## 2.2 Big-O Notation

When trying to characterize an algorithm's efficiency in terms of execution time, independent of any particular program or computer, it is important to quantify the number of operations or steps that the algorithm will require. If each of these steps is considered to be a basic unit of computation, then the execution time for an algorithm can be expressed as the number of steps required to solve the problem. Deciding on an appropriate basic unit of computation can be a complicated problem and will depend on how the algorithm is implemented.

A good basic unit of computation for comparing the summation algorithms shown earlier might be to count the number of assignment statements performed to compute the sum. In the function `sumOfN`, the number of assignment statements is 1 (*theSum=0*) plus the value of *n* (the number of times we perform *theSum = theSum + i*). We can denote this by a function, call it *T*, where  $T(n) = 1 + n$ . The parameter *n* is often referred to as the "size of the problem", and we can read this as "*T(n)* is the time it takes to solve a problem of size *n*, namely  $1+n$  steps".

In the summation functions given above, it makes sense to use the number of terms in the summation to denote the size of the problem. We can then say that the sum of the first 100,000 integers is a bigger instance of the summation problem than the sum of the first 1,000. Because of this, it might seem reasonable that the time required to solve the larger case would be greater than for the smaller case. Our goal then is to show how the algorithm's execution time changes with respect to the size of the problem.

Computer scientists prefer to take this analysis technique one step further. It turns out that the exact number of operations is not as important as determining the most dominant part of the  $T(n)$  function. In other words, as the problem gets larger, some portion of the  $T(n)$  function tends to overpower the rest. This dominant term is what, in the end, is used for comparison. The order of magnitude function describes the part of  $T(n)$  that increases the fastest as the value of *n* increases. **Order of magnitude** is often called **Big-O notation** (for "order") and written as  $O(f(n))$ . It provides a useful approximation to the actual number of steps in the computation. The function  $f(n)$  provides a simple representation of the dominant part of the original  $T(n)$ .

In the above example,  $T(n)=1+n$ . As *n* gets large, the constant 1 will become less and less significant to the final result. If we are looking for an approximation for  $T(n)$ , then we can drop the 1 and simply say that the running time is  $O(n)$ . It is important to note that the 1 is certainly significant for  $T(n)$ . However, as *n* gets large, our approximation will be just as accurate without it.

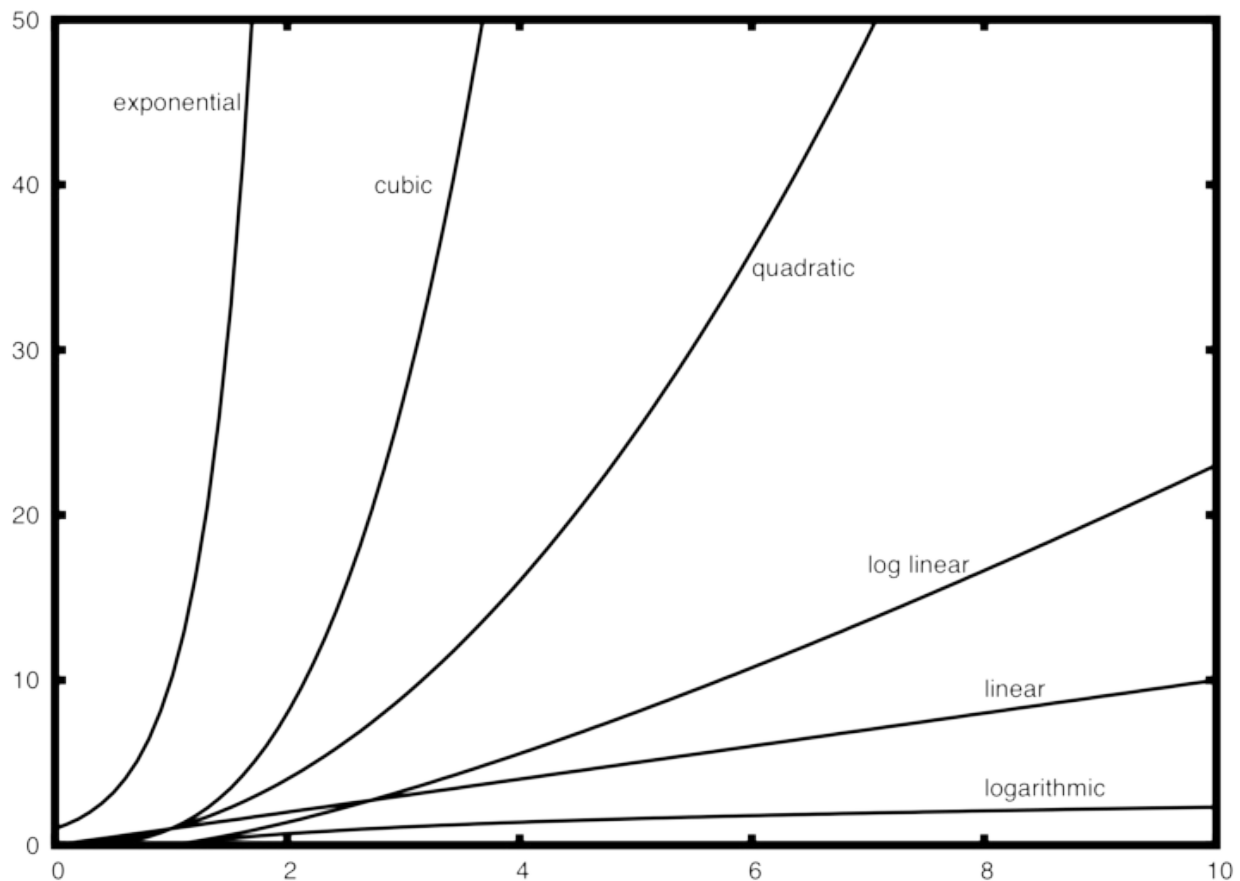
As another example, suppose that for some algorithm, the exact number of steps is  $T(n) = 5n^2 + 27n + 1005$ . When  $n$  is small, say 1 or 2, the constant 1005 seems to be the dominant part of the function. However, as  $n$  gets larger, the  $n^2$  term becomes the most important. In fact, when  $n$  is really large, the other two terms become insignificant in the role that they play in determining the final result. Again, to approximate  $T(n)$  as  $n$  gets large, we can ignore the other terms and focus on  $5n^2$ . In addition, the coefficient 5 becomes insignificant as  $n$  gets large. We would say then that the function  $T(n)$  has an order of magnitude  $f(n) = n^2$ , or simply that it is  $O(n^2)$ .

Although we do not see this in the summation example, sometimes the performance of an algorithm depends on the exact values of the data rather than simply the size of the problem. For these kinds of algorithms we need to characterize their performance in terms of best case, **worst case**, or **average case** performance. The worst case performance refers to a particular data set where the algorithm performs especially poorly. Whereas a different data set for the exact same algorithm might have extraordinarily good performance. However, in most cases the algorithm performs somewhere in between these two extremes (average case). It is important for a computer scientist to understand these distinctions so they are not misled by one particular case.

A number of very common order of magnitude functions will come up over and over as you study algorithms. These are shown in the table below. In order to decide which of these functions is the dominant part of any  $x = y$  function, we must see how they compare with one another as  $n$  gets large.

$f(n)$	name
1	Constant
$\log n$	Logarithmic
$n$	Linear
$n \log n$	Log linear
$n^2$	Quadratic
$n^3$	Cubic
$2^n$	Exponential

The following figure shows graphs of the common functions from the table. Notice that when  $n$  is small, the functions are not very well defined with respect to one another. It is hard to tell which is dominant. However, as  $n$  grows, there is a definite relationship and it is easy to see how they compare with one another.



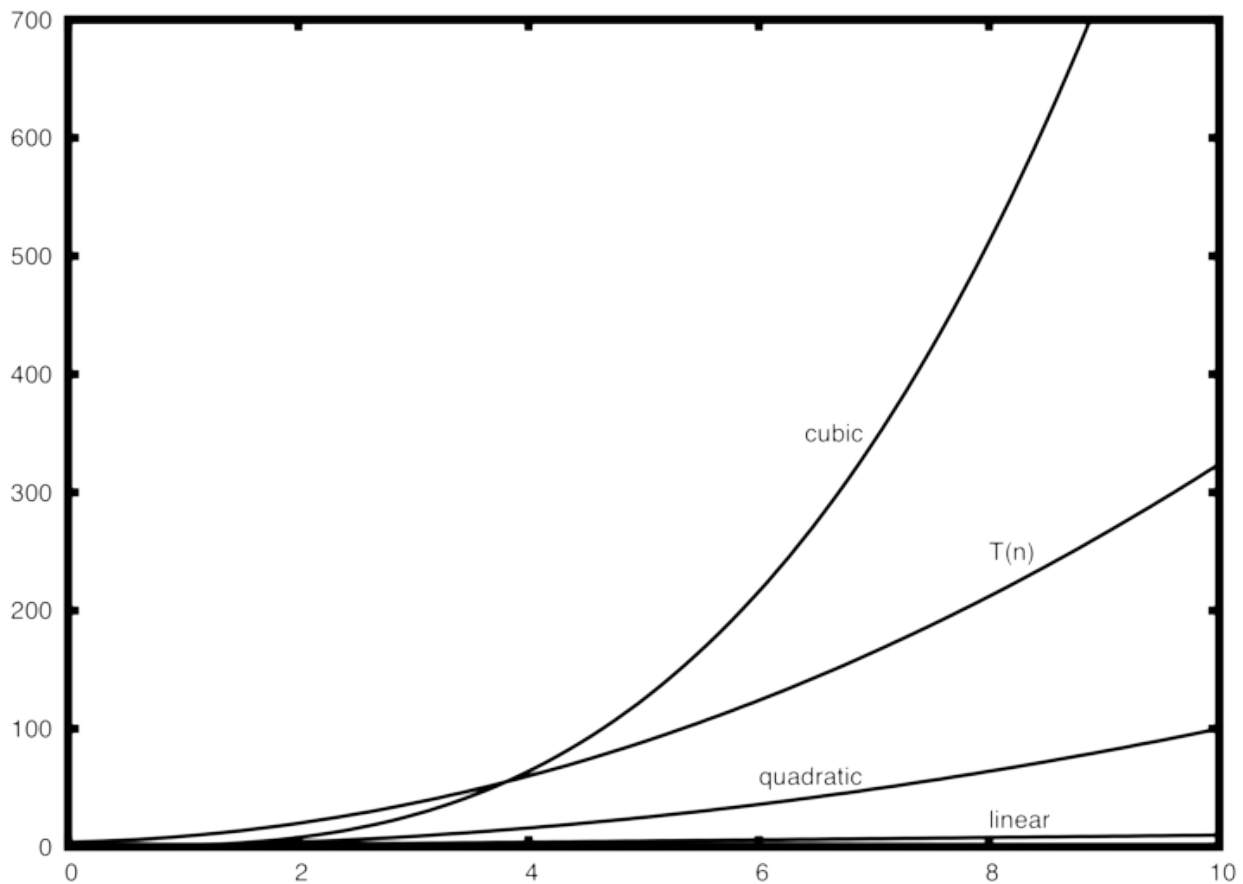
As a final example, suppose that we have the fragment of JavaScript code shown in below. Although this program does not really do anything, it is instructive to see how we can take actual code and analyze performance.

```
var a = 5;
var b = 6;
var c = 10;

for (var i = 0; i < n; i++) {
  for (var j = 0; j < n; j++) {
    var x = i * i;
    var y = j * j;
    var z = i * j;
  }
}
for (var k = 0; k < n; k++) {
  var w = a * k + 45;
  var v = b * b;
}
var d = 33;
```

The number of assignment operations is the sum of four terms. The first term is the constant 3, representing the three assignment statements at the start of the fragment. The second term is  $3n^2$ , since there are three statements that are performed  $n^2$  times due to

the nested iteration. The third term is  $2n$ , two statements iterated  $n$  times. Finally, the fourth term is the constant 1, representing the final assignment statement. This gives us  $T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$ . By looking at the exponents, we can easily see that the  $n^2$  term will be dominant and therefore this fragment of code is  $O(n^2)$ . Note that all of the other terms as well as the coefficient on the dominant term can be ignored as  $n$  grows larger.



It shows a few of the common Big-O functions as they compare with the  $T(n)$  function discussed above. Note that  $T(n)$  is initially larger than the cubic function. However, as  $n$  grows, the cubic function quickly overtakes  $T(n)$ . It is easy to see that  $T(n)$  then follows the quadratic function as  $n$  continues to grow.



## 2.3 An Anagram Detection Example

A good example problem for showing algorithms with different orders of magnitude is the classic anagram detection problem for strings. One string is an anagram of another if the second is simply a rearrangement of the first. For example, 'heart' and 'earth' are anagrams. For the sake of simplicity, we will assume that the two strings in question are of equal length and that they are made up of symbols from the set of 26 lowercase alphabetic characters. Our goal is to write a boolean function that will take two strings and return whether they are anagrams.

### 2.3.1 Solution 1: Checking Off

Our first solution to the anagram problem will check to see that each character in the first string actually occurs in the second. If it is possible to “checkoff” each character, then the two strings must be anagrams. Checking off a character will be accomplished by replacing it with the special JavaScript value `null`. However, since strings in JavaScript are immutable, the first step in the process will be to convert the second string to an array. Each character from the first string can be checked against the characters in the list and if found, checked off by replacement. An implementation of this strategy may look like this:

```
function anagramCheckingOff (s1, s2) {
  if (s1.length !== s2.length) return false

  const s2ToCheckOff = s2.split('')

  for (let i = 0; i < s1.length; i++) {
    let letterFound = false
    for (let j = 0; j < s2ToCheckOff.length; j++) {
      if (s1[i] === s2ToCheckOff[j]) {
        s2ToCheckOff[j] = null
        letterFound = true
        break
      }
    }
    if (!letterFound) return false
  }

  return true
}

anagramCheckingOff('abcd', 'dcba') ; // true
anagramCheckingOff('abcd', 'abcc'); // false
```

To analyze this algorithm, we need to note that each of the  $n$  characters in `s1` will cause an iteration through up to  $n$  characters in the list from `s2`. The number of visits then becomes the sum of the integers from 1 to  $n$ . We stated earlier that this can be written as

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

As  $n$  gets large, the  $n^2$  term will dominate the  $n$  term and the  $\frac{1}{2}$  can be ignored. Therefore, this solution is  $O(n^2)$ .

## 2.3.2 Solution 2: Sort and Compare

Another solution to the anagram problem will make use of the fact that even though `s1` and `s2` are different, they are anagrams only if they consist of exactly the same characters. So, if we begin by sorting each string alphabetically, from a to z, we will end up with the same string if the original two strings are anagrams. The code below shows this solution. First, we convert each string to an array using the string method `split`, and then we use the array method `sort` which lexographically sorts an array in place and then returns the array. Finally, we loop through the first string, checking to make sure that both strings contain the same letter at every index.

```
function anagramSortAndCompare (s1, s2) {
  if (s1.length !== s2.length) return false

  const sortedS1 = s1.split('').sort()
  const sortedS2 = s2.split('').sort()

  for (let i = 0; i < sortedS1.length; i++) {
    if (sortedS1[i] !== sortedS2[i]) return false
  }

  return true
}

anagramSortAndCompare('abcde', 'edcba'); // true
anagramSortAndCompare('abcde', 'abcd'); // false
```

At first glance you may be tempted to think that this algorithm is  $O(n)$ , since there is one simple iteration to compare the  $n$  characters after the sorting process. However, the two calls to the JavaScript array `sort` method are not without their own cost. As we will see in a later chapter, sorting is typically either  $O(n^2)$  or  $O(n \log n)$ , so the sorting operations dominate the iteration. In the end, this algorithm will have the same order of magnitude as that of the sorting process.

### 2.3.3 Solution 3: Brute Force

A brute force technique for solving a problem typically tries to exhaust all possibilities. For the anagram detection problem, we can simply generate a list of all possible strings using the characters from `s1` and then see if `s2` occurs. However, there is a difficulty with this approach. When generating all possible strings from `s1`, there are  $n$  possible first characters,  $n-1$  possible characters for the second position,  $n-2$  for the third, and so on. The total number of candidate strings is  $n*(n-1)*(n-2)*\dots*3*2*1$ , which is  $n!$ . Although some of the strings may be duplicates, the program cannot know this ahead of time and so it will still generate  $n!$  different strings.

It turns out that  $n!$  grows even faster than  $2^n$  as  $n$  gets large. In fact, if `s1` were 20 characters long, there would be  $20!=2,432,902,008,176,640,000$  possible candidate strings. If we processed one possibility every second, it would still take us 77,146,816,596 years to go through the entire list. This is probably not going to be a good solution.

### 2.3.4 Solution 4: Count and Compare

Our final solution to the anagram problem takes advantage of the fact that any two anagrams will have the same number of a's, the same number of b's, the same number of c's, and so on. In order to decide whether two strings are anagrams, we will first count the number of times each character occurs. Since there are 26 possible characters, we can use a list of 26 counters, one for each possible character. Each time we see a particular character, we will increment the counter at that position. In the end, if the two lists of counters are identical, the strings must be anagrams. The code below shows this solution.

```
function getLetterPosition(letter) {
  return letter.charCodeAt() - 'a'.charCodeAt();
}

function anagramCountCompare(s1,s2) {
  const s1LetterCounts = new Array(26).fill(0);
  const s2LetterCounts = new Array(26).fill(0);

  for (let i = 0; i < s1.length; i++) {
    const letterPosition = this.getLetterPosition(s1[i]);
    s1LetterCounts[letterPosition]++;
  }
  for (let i = 0; i < s2.length; i++) {
    const letterPosition = this.getLetterPosition(s2[i]);
    s2LetterCounts[letterPosition]++;
  }
  for (let i = 0; i < s1LetterCounts.length; i++) {
    if (s1LetterCounts[i] !== s2LetterCounts[i]) {
      return false;
    }
  }

  return true;
}

console.log( anagramCountCompare('apple', 'pleap') ); // true
console.log( anagramCountCompare('apple', 'applf') ); // false
```

Again, the solution has a number of iterations. However, unlike the first solution, none of them are nested. The first two iterations used to count the characters are both based on  $n$ . The third iteration, comparing the two lists of counts, always takes 26 steps since there are 26 possible characters in the strings. Adding it all up gives us  $T(n)=2n+26$  steps. That is  $O(n)$ . We have found a linear order of magnitude algorithm for solving this problem.

Before leaving this example, we need to say something about space requirements. Although the last solution was able to run in linear time, it could only do so by using additional storage to keep the two lists of character counts. In other words, this algorithm sacrificed space in order to gain time.

This is a common occurrence. On many occasions you will need to make decisions between time and space trade-offs. In this case, the amount of extra space is not significant. However, if the underlying alphabet had millions of characters, there would be more concern. As a computer scientist, when given a choice of algorithms, it will be up to you to determine the best use of computing resources given a particular problem.



## 2.4 Performance of JavaScript Data Structure

Now that you have a general idea of Big-O notation and the differences between the different functions, our goal in this section is to tell you about the Big-O performance for the operations on JavaScript list and dictionaries. I will then show you some timing experiments that illustrate the costs and benefits of using certain operations on each data structure. It is important for you to understand the efficiency of these JavaScript data structures because they are the building blocks we will use as we implement other data structures in the remainder of the book. In this section we are not going to explain why the performance is what it is. In later chapters you will see some possible implementations of both arrays and objects and how the performance depends on the implementation.

### 2.4.1 List

We have many choices to make when we implemented the list data structure. Each of these choices could have an impact on how fast list operations perform. We can look at the most common use the list data structure and optimize it's implementation so the most common operations are very fast. Of course we also tried to make the less common operations fast, but when a tradeoff had to be made the performance of a less common operation is often sacrificed in favor of the more common operation.

Two common operations are indexing and assigning to an index position. Both of these operations take the same amount of time no matter how large the list becomes. When an operation like this is independent of the size of the list they are  $O(1)$ .

Another very common programming task is to grow a list, this is the `append` method. As we will see later, we use an array as list datastore. There are three ways to create a longer array. You can use the `push()` method the `concat()` method or the manual assignment. Whatever, the append method is  $O(1)$ . However, the manual assignment is fastest option as we can if we benchmark it. This is the kind of thing important to know because it can help you make your own programs more efficient by choosing the right tool for the job.

```
var arr = [];  
  
function fn_push(n) {  
  var start = process.hrtime();  
  
  for (var i = 0; i < (n + 1); i++) {  
    arr.push(n);  
  }  
  
  var end = process.hrtime(start);  
  end = end[0] * 1000000 + end[1] / 1000;  
  console.log(`Computed in ${end} milliseconds`);  
}  
  
function fn_push_manually(n) {  
  var start = process.hrtime();  
  
  for (var i = 0; i < (n + 1); i++) {  
    arr[arr.length] = n;  
  }  
  
  var end = process.hrtime(start);  
  end = end[0] * 1000000 + end[1] / 1000;  
  console.log(`Computed in ${end} milliseconds`);  
}  
  
function fn_concat_end(n) {  
  var start = process.hrtime();  
  
  for (var i = 0; i < (n + 1); i++) {  
    arr.concat([n]);  
  }  
  
  var end = process.hrtime(start);  
  end = end[0] * 1000000 + end[1] / 1000;  
  console.log(`Computed in ${end} milliseconds`);  
}  
  
fn_push(1000); // Computed in 240.77 milliseconds  
arr = [];  
fn_push_manually(1000); // Computed in 47.038 milliseconds  
arr = [];  
fn_concat_end(1000); // Computed in 195.935 milliseconds
```

Now that we have seen how performance can be measured concretely you can look at the table below to see the Big-O efficiency of all the basic list operations.

# Basic Data Structures

## Objectives

## Chapters

- [3.1 What Are Linear Structures?](#)
- [3.2 Stack](#)
- [3.3 Queue](#)
- [3.4 Deque](#)
- [3.5 List](#)



# 3.1. What Are Linear Structures?

We will begin our study of data structures by considering four simple but very powerful concepts. Stacks, queues, deques, and lists are examples of data collections whose items are ordered depending on how they are added or removed. Once an item is added, it stays in that position relative to the other elements that came before and came after it. Collections such as these are often referred to as **linear data structures**.

Linear structures can be thought of as having two ends. Sometimes these ends are referred to as the "left" and the "right" or in some cases the "front" and the "rear". You could also call them the "top" and the "bottom". The names given to the ends are not significant. What distinguishes one linear structure from another is the way in which items are added and removed, in particular the location where these additions and removals occur. For example, a structure might allow new items to be added at only one end. Some structures might allow items to be removed from either end.

These variations give rise to some of the most useful data structures in computer science. They appear in many algorithms and can be used to solve a variety of important problems.

## 3.2 Stack

A stack (sometimes called a "push-down stack") is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the "top". The end opposite the top is known as the "base".

The base of the stack is significant since items stored in the stack that are closer to the base represent those that have been in the stack the longest. The most recently added item is the one that is in position to be removed first. This ordering principle is sometimes called **LIFO**, **last-in first-out**. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base.

Many examples of stacks occur in everyday situations. Almost any cafeteria has a stack of trays or plates where you take the one at the top, uncovering a new tray or plate for the next customer in line. Imagine a stack of books on a desk. The only book whose cover is visible is the one on top. To access others in the stack, we need to remove the ones that are sitting on top of them.

One of the most useful ideas related to stacks comes from the simple observation of items as they are added and then removed. Assume you start out with a clean desktop. Now place books one at a time on top of each other. You are constructing a stack. Consider what happens when you begin removing books. The order that they are removed is exactly the reverse of the order that they were placed. Stacks are fundamentally important, as they can be used to reverse the order of items. The order of insertion is the reverse of the order of removal.

Considering this reversal property, you can perhaps think of examples of stacks that occur as you use your computer. For example, every web browser has a Back button. As you navigate from web page to web page, those pages are placed on a stack (actually it is the URLs that are going on the stack). The current page that you are viewing is on the top and the first page you looked at is at the base. If you click on the Back button, you begin to move in reverse order through the pages.

### 3.2.1 The Stack Abstract Data Type

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top". Stacks are ordered LIFO. The stack operations

are given below.

- `stack()` creates a new stack that is empty. It needs no parameters and returns an empty stack.
- `push(item)` adds a new item to the top of the stack. It needs the item and returns nothing.
- `pop()` removes the top item from the stack and returns that element. It needs no parameters and returns the item. The stack is modified.
- `peek()` returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- `isEmpty()` tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items on the stack. It needs no parameters and returns an integer.

For example, if `s` is a stack that has been created and starts out empty, then the table below shows the results of a sequence of stack operations. Under stack contents, the top item is listed at the far right.

Stack Operation	Stack Contents	Return Value
<code>s.isEmpty()</code>	[ ]	true
<code>s.push(4)</code>	[4]	
<code>s.push('banana')</code>	[4, 'banana']	
<code>s.peek()</code>	[4, 'banana']	'banana'
<code>s.push(true)</code>	[4, 'banana', true]	
<code>s.size()</code>	[4, 'banana', true]	3
<code>s.isEmpty()</code>	[4, 'banana', true]	false
<code>s.push(8.4)</code>	[4, 'banana', true, 8.4]	
<code>s.pop()</code>	[4, 'banana', true]	8.4
<code>s.pop()</code>	[4, 'banana']	true
<code>s.size()</code>	[4, 'banana']	2

## 3.2.2 A Stack Implementation

Now that we have clearly defined the stack as an abstract data type we will turn our attention to using JavaScript to implement the stack. Recall that when we give an abstract data type a physical implementation we refer to the implementation as a data structure.

As we described earlier, in JavaScript, as in any object-oriented programming language, the implementation of choice for an abstract data type such as a stack is the creation of a new class. The stack operations are implemented as methods. Further, to implement a stack, which is a collection of elements, it makes sense to utilize the power and simplicity of the array provided by JavaScript.

Recall that the arrays in JavaScript are list-like objects whose prototype has methods to perform traversal and mutation operations. For example, if we have the list [2,5,3,6,7,4], we can use the array `reverse()` method to change which end of the list will be considered the top of the stack and which will be the base. Once that decision is made, the operations can be implemented using the list methods such as `append` and `pop`.

The following stack implementation assumes that the end of the list will hold the top element of the stack. As the stack grows (as `push` operations occur), new items will be added on the end of the list. `pop` operations will manipulate that same end.

```
class Stack {
  constructor() {
    this.items = [];
  }

  isEmpty() {
    return !Boolean(this.items.length); // equivalent to `!!this.items.length`
  }

  push(item) {
    this.items[this.items.length] = item;
  }

  pop() {
    return this.items.pop();
  }

  peek() {
    return this.items[this.items.length-1];
  }

  size() {
    return this.items.length;
  }
}
```

Now we can create a `Stack` object and use it and perform the sequence of operations from first table. Notice that the definition of the `Stack` class is imported as a JavaScript module.

```
import Stack from './stack.js';

var s = new Stack();
console.log( s.isEmpty() ); // true
s.push(4);
s.push('banana');
console.log( s.peek() ); // banana
s.push(true);
console.log( s.size() ); // 3
console.log( s.isEmpty() ); // false
s.push(8.4);
console.log( s.pop() ); // 8.4
console.log( s.pop() ); // true
console.log( s.size() ); // 2
```

It is important to note that we could have chosen to implement the stack using a list where the top is at the beginning instead of at the end. In this case, the previous `pop` and `append` methods would no longer work and we would have to index position 0 (the first item in the list) explicitly using `pop` and `insert` .

```
class Stack {
  constructor() {
    this.items = [];
  }

  isEmpty() {
    return !Boolean(this.items.length); // equivalent to `!!this.items.length`
  }

  push(item) {
    this.items.splice(0, 0, item);
  }

  pop() {
    return this.items.shift();
  }

  peek() {
    return this.items[0];
  }

  size() {
    return this.items.length;
  }
}

export default Stack;
```

This ability to change the physical implementation of an abstract data type while maintaining the logical characteristics is an example of abstraction at work. However, even though the stack will work either way, if we consider the performance of the two implementations, there is definitely a difference. Recall that the manual assignment and `pop()` operations were both  $O(1)$ . This means that the first implementation will perform `push` and `pop` in constant time no matter how many items are on the stack. The performance of the second implementation suffers in that the `splice` and `shift` operations will both require  $O(n)$  for a stack of size  $n$ . The reason is that `shift()` has to re-index the whole array while `pop()` doesn't. For `splice`, is going to vary a lot between implementations (e.g., vendors). Clearly, even though the implementations are logically equivalent, they would have very different timings when performing benchmark testing.

### 3.2.3 Simple Balanced Parentheses

We now turn our attention to using stacks to solve real computer science problems. You have no doubt written arithmetic expressions such as  $(5+6)*(7+8)/(4+3)$  where parentheses are used to order the performance of operations.

In these example, parentheses must appear in a balanced fashion. **Balanced parentheses** means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. The ability to differentiate between parentheses that are correctly balanced and those that are unbalanced is an important part of recognizing many programming language structures.

The challenge then is to write an algorithm that will read a string of parentheses from left to right and decide whether the symbols are balanced. To solve this problem we need to make an important observation. As you process symbols from left to right, the most recent opening parenthesis must match the next closing symbol. Also, the first opening symbol processed may have to wait until the very last symbol for its match. Closing symbols match opening symbols in the reverse order of their appearance; they match from the inside out. This is a clue that stacks can be used to solve the problem.

Once you agree that a stack is the appropriate data structure for keeping the parentheses, the statement of the algorithm is straightforward. Starting with an empty stack, process the parenthesis strings from left to right. If a symbol is an opening parenthesis, push it on the stack as a signal that a corresponding closing symbol needs to appear later. If, on the other hand, a symbol is a closing parenthesis, pop the stack. As long as it is possible to pop the stack to match every closing symbol, the parentheses remain balanced. If at any time there is no opening symbol on the stack to match a closing symbol, the string is not balanced properly. At the end of the string, when all symbols have been processed, the stack should be empty. The JavaScript code to implement this algorithm is shown below.

```
import Stack from './stack.js';

function parChecker(symbolString) {
  var s = new Stack();
  var balanced = true;
  var index = 0;

  while (index < symbolString.length && balanced) {
    var symbol = symbolString[index];
    if (symbol == '(') {
      s.push(symbol);
    }
    else {
      if (s.isEmpty()) {
        balanced = false;
      }
      else {
        s.pop();
      }
    }
    index = index + 1;
  }

  if (balanced && s.isEmpty()) {
    return true;
  }
  else {
    return false;
  }
}

console.log( parChecker('((()))') ); // true
console.log( parChecker('(()') ); // false
```

This function, `parChecker`, assumes that a `stack` class is available and returns a boolean result as to whether the string of parentheses is balanced. Note that the boolean variable `balanced` is initialized to `true` as there is no reason to assume otherwise at the start. If the current symbol is `(`, then it is pushed on the stack (10). Note also in line 17 that `pop` simply removes a symbol from the stack. The returned value is not used since we know it must be an opening symbol seen earlier. At the end (lines 22–27), as long as the expression is balanced and the stack has been completely cleaned off, the string represents a correctly balanced sequence of parentheses.

## 3.2.4 Balanced Symbols (A General Case)

The balanced parentheses problem shown above is a specific case of a more general situation that arises in many programming languages. The general problem of balancing and nesting different kinds of opening and closing symbols occurs frequently. For example, in JavaScript square brackets, `[` and `]`, are used for arrays; curly braces, `{` and `}`, are used for objects; and parentheses, `(` and `)`, are used for tuples and arithmetic expressions. It is possible to mix symbols as long as each maintains its own open and close relationship. Strings of symbols such as the following are properly balanced in that not only does each opening symbol have a corresponding closing symbol, but the types of symbols match as well.

```
{ { ( [ ] [ ] ) } ( ) }  
[ [ { { ( ( ) ) } } ] ]  
[ ] [ ] [ ] ( ) { }
```

Compare those with the following strings that are not balanced:

```
( [ ] ]  
( ( ( ) ] ) )  
[ { ( ) ]
```

The simple parentheses checker from the previous section can easily be extended to handle these new types of symbols. Recall that each opening symbol is simply pushed on the stack to wait for the matching closing symbol to appear later in the sequence. When a closing symbol does appear, the only difference is that we must check to be sure that it correctly matches the type of the opening symbol on top of the stack. If the two symbols do not match, the string is not balanced. Once again, if the entire string is processed and nothing is left on the stack, the string is correctly balanced.

The JavaScript program to implement this is shown below:



```

import Stack from './stack.js';

function parChecker(symbolString) {
  var s = new Stack();
  var balanced = true;
  var index = 0;
  var openingSymbols = ['(', '[', '{'];

  while (index < symbolString.length && balanced) {
    var symbol = symbolString[index];
    if (openingSymbols.includes(symbol)) {
      s.push(symbol);
    }
    else {
      if (s.isEmpty()) {
        balanced = false;
      }
      else {
        var top = s.pop();
        if (!matches(top, symbol)) {
          balanced = false;
        }
      }
    }
    index = index + 1;
  }

  if (balanced && s.isEmpty()) {
    return true;
  }
  else {
    return false;
  }
}

function matches(open, close) {
  opens = ['(', '[', '{'];
  closers = [')', ']', '}'];
  return opens.indexOf(open) === closers.indexOf(close);
}

console.log( '{ { ( [ ] [ ] ) } ( ) }' ); // true
console.log( '( [ ) ]' ); // false

```

The `includes()` is a new method introduced by the ECMAScript 2016. It determine whether an array includes a certain element, returning `true` or `false` as appropriate.

The only change appears in line 20 where we call a helper function, `matches`, to assist with symbol-matching. Each symbol that is removed from the stack must be checked to see that it matches the current closing symbol. If a mismatch occurs, the boolean variable `balanced`

is set to `false` .

These two examples show that stacks are very important data structures for the processing of language constructs in computer science. Almost any notation you can think of has some type of nested symbol that must be matched in a balanced order. There are a number of other important uses for stacks in computer science. We will continue to explore them in the next sections.

## 3.2.4 Converting Decimal Numbers to Binary Numbers

In your study of computer science, you have probably been exposed in one way or another to the idea of a binary number. Binary representation is important in computer science since all values stored within a computer exist as a string of binary digits, a string of 0s and 1s. Without the ability to convert back and forth between common representations and binary numbers, we would need to interact with computers in very awkward ways.

Integer values are common data items. They are used in computer programs and computation all the time. We learn about them in math class and of course represent them using the decimal number system, or base 10. The decimal number  $233_{10}$  and its corresponding binary equivalent  $11101001_2$  are interpreted respectively as  $2 \times 10^2 + 3 \times 10^1 + 3 \times 10^0$  and  $1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ .

But how can we easily convert integer values into binary numbers? The answer is an algorithm called “Divide by 2” that uses a stack to keep track of the digits for the binary result.

The Divide by 2 algorithm assumes that we start with an integer greater than 0. A simple iteration then continually divides the decimal number by 2 and keeps track of the remainder. The first division by 2 gives information as to whether the value is even or odd. An even value will have a remainder of 0. It will have the digit 0 in the ones place. An odd value will have a remainder of 1 and will have the digit 1 in the ones place. We think about building our binary number as a sequence of digits; the first remainder we compute will actually be the last digit in the sequence. We see here the reversal property that signals that a stack is likely to be the appropriate data structure for solving the problem.

The JavaScript code below implements the Divide by 2 algorithm. The function `divideBy2` takes an argument that is a decimal number and repeatedly divides it by 2.

```

import Stack from './stack.js';

function divideBy2(decNumber) {
  var remstack = new Stack();

  while (decNumber > 0) {
    var rem = decNumber % 2;
    remstack.push(rem);
    decNumber = Math.floor(decNumber / 2);
  }
  var binString = '';
  while (!remstack.isEmpty()) {
    binString = binString + remstack.pop().toString();
  }

  return binString;
}

console.log( divideBy2(42) ); // 101010

```

Line 7 uses the built-in modulo operator, %, to extract the remainder and line 8 then pushes it on the stack. After the division process reaches 0, a binary string is constructed in lines 11-13. Line 11 creates an empty string. The binary digits are popped from the stack one at a time and appended to the right-hand end of the string. The binary string is then returned.

The algorithm for binary conversion can easily be extended to perform the conversion for any base. In computer science it is common to use a number of different encodings. The most common of these are binary, octal (base 8), and hexadecimal (base 16).

The decimal number  $233$  and its corresponding octal and hexadecimal equivalents  $351_8$  and  $E9_{16}$  are interpreted as  $3 \times 8^2 + 5 \times 8^1 + 1 \times 8^0$  and as  $14 \times 16^1 + 9 \times 16^0$ .

The function `divideBy2` can be modified to accept not only a decimal value but also a base for the intended conversion. The "Divide by 2" idea is simply replaced with a more general "Divide by base". Our new function we be called `baseConverter` and takes a decimal number and any base between 2 and 16 as parameters. The remainders are still pushed onto the stack until the value being converted becomes 0. The same left-to-right string construction technique can be used with one slight change. Base 2 through base 10 numbers need a maximum of 10 digits, so the typical digit characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 work fine. The problem comes when we go beyond base 10. We can no longer simply use the remainders, as they are themselves represented as two-digit decimal numbers. Instead we need to create a set of digits that can be used to represent those remainders beyond 9.

A solution to this problem is to extend the digit set to include some alphabet characters. For example, hexadecimal uses the ten decimal digits along with the first six alphabet characters for the 16 digits. To implement this, a digit string is created that stores the digits in their corresponding positions. 0 is at position 0, 1 is at position 1, A is at position 10, B is at position 11, and so on. When a remainder is removed from the stack, it can be used to index into the digit string and the correct resulting digit can be appended to the answer. For example, if the remainder 13 is removed from the stack, the digit D is appended to the resulting string.

```
function baseConverter(decNumber, base) {
    var digits = '0123456789ABCDEF';

    var remstack = new Stack();

    while (decNumber > 0) {
        var rem = decNumber % base;
        remstack.push(rem);
        decNumber = Math.floor(decNumber / base);
    }
    var newString = '';
    while (!remstack.isEmpty()) {
        newString = newString + digits[remstack.pop()];
    }

    return newString;
}

console.log( baseConverter(25, 2) );
console.log( baseConverter(25, 16) );
```

### 3.2.5 Infix, Prefix and Postfix Expressions

When you write an arithmetic expression such as  $B * C$ , the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator  $*$  appears between them in the expression. This type of notation is referred to as **infix** since the operator is in between the two operands that it is working on.

Consider another infix example,  $A + B * C$ . The operators  $+$  and  $*$  still appear between the operands, but there is a problem. Which operands do they work on? Does the  $+$  work on A and B or does the  $*$  take B and C? The expression seems ambiguous.

In fact, you have been reading and writing these types of expressions for a long time and they do not cause you any problem. The reason for this is that you know something about the operators  $+$  and  $*$ . Each operator has a **precedence** level. Operators of higher

precedence are used before operators of lower precedence. The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

Let's interpret the troublesome expression  $A + B * C$  using operator precedence.  $B$  and  $C$  are multiplied first, and  $A$  is then added to that result.  $(A + B) * C$  would force the addition of  $A$  and  $B$  to be done first before the multiplication. In expression  $A + B + C$ , by precedence (via associativity), the leftmost  $+$  would be done first.

Although all this may be obvious to you, remember that computers need to know exactly what operators to perform and in what order. One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a **fully parenthesized** expression. This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations; there is no ambiguity. There is also no need to remember any precedence rules.

The expression  $A + B * C + D$  can be rewritten as  $((A + (B * C)) + D)$  to show that the multiplication happens first, followed by the leftmost addition.  $A + B + C + D$  can be written as  $((((A + B) + C) + D)$  since the addition operations associate from left to right.

These changes to the position of the operator with respect to the operands create two new expression formats, **prefix** and **postfix**. Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands.

$A + B * C$  would be written as  $+ A * B C$  in prefix. The multiplication operator comes immediately before the operands  $B$  and  $C$ , denoting that  $*$  has precedence over  $+$ . The addition operator then appears before the  $A$  and the result of the multiplication.

In postfix, the expression would be  $A B C * +$ . Again, the order of operations is preserved since the  $*$  appears immediately after the  $B$  and the  $C$ , denoting that  $*$  has precedence, with  $+$  coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

A few more examples should help to make this a bit clearer:

Infix Expression	Prefix Expression	Postfix Expression
$A + B$	$+ A B$	$A B +$
$A + B * C$	$+ A * B C$	$A B C * +$

Now consider the infix expression  $(A + B) * C$ . Recall that in this case, infix requires the parentheses to force the performance of the addition before the multiplication. However, when  $A + B$  was written in prefix, the addition operator was simply moved before the operands,  $+ A B$ . The result of this operation becomes the first operand for the multiplication. The multiplication operator is moved in front of the entire expression, giving us  $* + A B C$ . Likewise, in postfix  $A B +$  forces the addition to happen first. The multiplication can be done to that result and the remaining operand  $C$ . The proper postfix expression is then  $A B + C *$ .

Consider again the three expressions in the following table:

Infix Expression	Prefix Expression	Postfix Expression
$(A + B) * C$	$* + A B C$	$A B + C *$

Something very important has happened. Where did the parentheses go? Why don't we need them in prefix and postfix? The answer is that the operators are no longer ambiguous with respect to the operands that they work on. Only infix notation requires the additional symbols. The order of operations within prefix and postfix expressions is completely determined by the position of the operator and nothing else. In many ways, this makes infix the least desirable notation to use.

The next table shows some additional examples of infix expressions and the equivalent prefix and postfix expressions. Be sure that you understand how they are equivalent in terms of the order of the operations being performed.

Infix Expression	Prefix Expression	Postfix Expression
$A + B * C + D$	$+ + A * B C D$	$A B C * + D +$
$(A + B) * (C + D)$	$* + A B + C D$	$A B + C D + *$
$A * B + C * B$	$+ * A B * C B$	$A B * C D * +$
$A + B + C + D$	$+ + + A B C D$	$A B + C + D +$

### 3.2.5.1 Conversion of Infix Expressions to Prefix and Postfix

So far, we have used ad hoc methods to convert between infix expressions and the equivalent prefix and postfix expression notations. As you might expect, there are algorithmic ways to perform the conversion that allow any expression of any complexity to be correctly transformed.

The first technique that we will consider uses the notion of a fully parenthesized expression that was discussed earlier. Recall that  $A + B * C$  can be written as  $(A + (B * C))$  to show explicitly that the multiplication has precedence over the addition. On closer observation,

however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

Consider this expression  $(A + (B * C))$ . In the subexpression  $(B * C)$ , if we move the multiplication symbol to replace the right parenthesis and remove the matching left parenthesis, giving us  $BC*$ , we convert it to postfix notation. If the addition operator is also moved to its corresponding right parenthesis position and the matching left parenthesis removed, the complete postfix expression would result:  $ABC*+$ .

If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation. The position of the parenthesis pair is actually a clue to the final position of the enclosed operator:  $+A*BC$ .

So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix notation.

### 3.2.5.2 General Infix-to-Postfix Conversion

We need to develop an algorithm to convert any infix expression to a postfix expression. To do this we will look closer at the conversion process.

Consider once again the expression  $A + B * C$ . As shown above,  $ABC*+$  is the postfix equivalent. We have already noted that the operands  $A$ ,  $B$ , and  $C$  stay in their relative positions. It is only the operators that change position. Let's look again at the operators in the infix expression. The first operator that appears from left to right is  $+$ . However, in the postfix expression,  $+$  is at the end since the next operator,  $*$ , has precedence over addition. The order of the operators in the original expression is reversed in the resulting postfix expression.

As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their precedence. This is the case with the addition and the multiplication in this example. Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used. Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

What about  $(A + B) * C$ ? Recall that  $AB + C*$  is the postfix equivalent. Again, processing this infix expression from left to right, we see  $+$  first. In this case, when we see  $*$ ,  $+$  has already been placed in the result expression because it has precedence over  $*$  by virtue of the parentheses. We can now start to see how the conversion algorithm will work. When we

see a left parenthesis, we will save it to denote that another operator of high precedence will be coming. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique). When that right parenthesis does appear, the operator can be popped from the stack.

As we scan the infix expression from left to right, we will use a stack to keep the operators. This will provide the reversal that we noted in the first example. The top of the stack will always be the most recently saved operator. Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are `*`, `/`, `+`, and `-`, along with the left and right parentheses, `(` and `)`. The operand tokens are the single-character identifiers `A`, `B`, `C`, and so on. The following steps will produce a string of tokens in postfix order.

1. Create an empty stack called `opstack` for keeping operators. Create an empty array for output.
2. Convert the input infix string to a list by using the string method `split`.
3. Scan the token list from left to right.
  - i. If the token is an operand, append it to the end of the output list.
  - ii. If the token is a left parenthesis, push it on the `opstack`.
  - iii. If the token is a right parenthesis, pop the `opstack` until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
  - iv. If the token is an operator, `*`, `/`, `+`, or `-`, push it on the `opstack`. However, first remove any operators already on the `opstack` that have higher or equal precedence and append them to the output list.
4. When the input expression has been completely processed, check the `opstack`. Any operators still on the stack can be removed and appended to the end of the output list.

In order to code the algorithm in JavaScript, we will use an object as dictionary to hold the precedence values for the operators. This object will map each operator to an integer that can be compared against the precedence levels of other operators (we have arbitrarily used the integers 3, 2, and 1). The left parenthesis will receive the lowest value possible. This way any operator that is compared against it will have higher precedence and will be placed on top of it. Line 15 defines the operands to be any upper-case character or digit.



```

import Stack from './stack.js';

function infixToPostfix(infixexpr) {
  infixexpr = infixexpr.replace(/\s/g, '');
  var prec = {
    '*': 3,
    '/': 3,
    '+': 2,
    '-': 2,
    '(': 1
  };

  var opStack = new Stack();
  var postfixList = [];
  var tokenList = infixexpr.split('');

  for (var i = 0; i < tokenList.length; i++) {
    var token = tokenList[i];
    if ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'.includes(token) || '0123456789'.includes(token)) {
      postfixList.push(token);
    }
    else if (token === '(') {
      opStack.push(token);
    }
    else if (token === ')') {
      var topToken = opStack.pop();
      while (topToken !== '(') {
        postfixList.push(topToken);
        topToken = opStack.pop();
      }
    }
    else {
      while (!opStack.isEmpty() && prec[opStack.peek()] >= prec[token]) {
        postfixList.push(opStack.pop());
      }
      opStack.push(token);
    }
  }

  while (!opStack.isEmpty()) {
    postfixList.push(opStack.pop());
  }
  return postfixList.join(' ');
}

console.log( infixToPostfix('A * B + C * D') );
console.log( infixToPostfix('( A + B ) * C - ( D - E ) * ( F + G )' ) );

```

### 3.2.5.3 Postfix Evaluation

As a final stack example, we will consider the evaluation of an expression that is already in postfix notation. In this case, a stack is again the data structure of choice. However, as you scan the postfix expression, it is the operands that must wait, not the operators as in the conversion algorithm above. Another way to think about the solution is that whenever an operator is seen on the input, the two most recent operands will be used in the evaluation.

To see this in more detail, consider the postfix expression `4 5 6 * +`. As you scan the expression from left to right, you first encounter the operands 4 and 5. At this point, you are still unsure what to do with them until you see the next symbol. Placing each on the stack ensures that they are available if an operator comes next.

In this case, the next symbol is another operand. So, as before, push it and check the next symbol. Now we see an operator, `*`. This means that the two most recent operands need to be used in a multiplication operation. By popping the stack twice, we can get the proper operands and then perform the multiplication (in this case getting the result 30).

We can now handle this result by placing it back on the stack so that it can be used as an operand for the later operators in the expression. When the final operator is processed, there will be only one value left on the stack. Pop and return it as the result of the expression.

Lets take a slightly more complex example, `7 8 + 3 2 + /`. There are two things to note in this example. First, the stack size grows, shrinks, and then grows again as the subexpressions are evaluated. Second, the division operation needs to be handled carefully. Recall that the operands in the postfix expression are in their original order since postfix changes only the placement of operators. When the operands for the division are popped from the stack, they are reversed. Since division is not a commutative operator, in other words  $15/5$  is not the same as  $5/15$ , we must be sure that the order of the operands is not switched.

Assume the postfix expression is a string of tokens delimited by spaces. The operators are `*`, `/`, `+`, and `-` and the operands are assumed to be single-digit integer values. The output will be an integer result.

1. Create an empty stack called `operandStack`.
2. Convert the string to a list by using the string method `split`.
3. Scan the token list from left to right.
  - i. If the token is an operand, convert it from a string to an integer and push the value onto the `operandStack`.
  - ii. If the token is an operator, `*`, `/`, `+` or `-`, it will need two operands. Pop the `operandStack` twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the `operandStack`.
4. When the input expression has been completely processed, the result is on the stack.

Pop the `operandStack` and return the value.

To assist with the arithmetic, a helper function `doMath` is defined that will take two operands and an operator and then perform the proper arithmetic operation. Here is the complete function:

```
import Stack from './stack.js';

function postfixEval(postfixExpr) {
    postfixExpr = postfixExpr.replace(/\s/g, '');
    var operandStack = new Stack();
    var tokenList = postfixExpr.split('');

    for (var i = 0; i < tokenList.length; i++) {
        var token = tokenList[i];
        if ('0123456789'.includes(token)) {
            operandStack.push(parseInt(token));
        }
        else {
            let operand2 = operandStack.pop();
            let operand1 = operandStack.pop();
            let result = doMath(token, operand1, operand2);
            operandStack.push(result);
        }
    }
    return operandStack.pop();
}

function doMath(op, op1, op2) {
    if (op === '*') {
        return op1 * op2;
    }
    else if (op === '/') {
        return op1 / op2;
    }
    else if (op === '+') {
        return op1 + op2;
    }
    else {
        return op1 - op2;
    }
}

console.log( postfixEval('7 8 + 3 2 + /') );
```

It is important to note that in both the postfix conversion and the postfix evaluation programs we assumed that there were no errors in the input expression. Using these programs as a starting point, you can easily see how error detection and reporting can be included. We leave this as an exercise at the end of the chapter.



## 3.3 Queue

A queue is an ordered collection of items where the addition of new items happens at one end, called the "rear", and the removal of existing items occurs at the other end, commonly called the "front". As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called **FIFO, first-in first-out**. It is also known as "first-come first-served".

The simplest example of a queue is the typical line that we all participate in from time to time. We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line (so that we can pop the tray stack). Well-behaved lines, or queues, are very restrictive in that they have only one way in and only one way out. There is no jumping in the middle and no leaving before you have waited the necessary amount of time to get to the front.

Computer science also has common examples of queues. Let's take an office with 30 computers networked with a single printer. When employees want to print, their print tasks "get in line" with all the other printing tasks that are waiting. The first task in is the next to be completed. If you are last in line, you must wait for all the other tasks to print ahead of you. We will explore this interesting example in more detail later.

In addition to printing queues, operating systems use a number of different queues to control processes within a computer. The scheduling of what gets done next is typically based on a queuing algorithm that tries to execute programs as quickly as possible and serve as many users as it can. Also, as we type, sometimes keystrokes get ahead of the characters that appear on the screen. This is due to the computer doing other work at that moment. The keystrokes are being placed in a queue-like buffer so that they can eventually be displayed on the screen in the proper order.

### 3.3.1 The Queue Abstract Data Type

The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear", and removed from the other end, called the "front". Queues maintain a FIFO ordering property. The queue operations are given below.

- `Queue()` creates a new queue that is empty. It needs no parameters and returns an

empty queue.

- `enqueue(item)` adds an item to the rear of the queue. It needs the item and returns nothing.
- `dequeue()` removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- `isEmpty()` tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the queue. It needs no parameters and returns an integer.

As an example, if we assume that `q` is a queue that has been created and is currently empty, then the following table shows the results of a sequence of queue operations. The queue contents are shown such that the front is on the right. 4 was the first item enqueued so it is the first item returned by `dequeue`.

Queue Operation	Queue Contents	Return Value
<code>q.isEmpty()</code>	[ ]	true
<code>q.enqueue(4)</code>	[ 4 ]	
<code>q.enqueue('banana')</code>	[ 'banana' , 4 ]	
<code>q.enqueue(true)</code>	[ true, 'banana', 4 ]	
<code>q.size()</code>	[ true, 'banana', 4 ]	3
<code>q.isEmpty()</code>	[ true, 'banana', 4 ]	false
<code>q.enqueue(8.4)</code>	[ 8.4, true, 'banana', 4 ]	
<code>q.dequeue()</code>	[ 8.4, true, 'banana' ]	4
<code>q.dequeue()</code>	[ 8.4, true ]	'banana'
<code>q.size()</code>	[ 8.4, true ]	2

### 3.3.2 Implementing a Queue in JavaScript

It is again appropriate to create a new class for the implementation of the abstract data type queue. As before, we will use the power and simplicity of the list collection to build the internal representation of the queue.

We need to decide which end of the list to use as the rear and which to use as the front. The implementation I propose assumes that the rear is at position 0 in the list.

```
class Queue {
  constructor() {
    this.items = [];
  }

  isEmpty() {
    return (this.items.length === 0);
  }

  enqueue(item) {
    this.items.unshift(item);
  }

  dequeue() {
    return this.items.pop();
  }

  size() {
    return this.items.length;
  }
}

export default Queue;
```

Manipulation of this queue would give the following results:

```
var q = new Queue();

console.log( q.isEmpty() ); // true
q.enqueue(8.4);
q.enqueue('banana')
console.log( q.size() ); // 2
console.log( q.dequeue() ); // 8.4
console.log( q.isEmpty() ); // false
console.log( q.size() ); // 1
```

### 3.3.3 Simulation: Hot Potato

One of the typical applications for showing a queue in action is to simulate a real situation that requires data to be managed in a FIFO manner. To begin, let's consider the children's game Hot Potato. In this game children line up in a circle and pass an item from neighbor to neighbor as fast as they can. At a certain point in the game, the action is stopped and the child who has the item (the potato) is removed from the circle. Play continues until only one child is left.

We will implement a general simulation of Hot Potato. Our program will input a list of names and a constant, call it "num", to be used for counting. It will return the name of the last person remaining after repetitive counting by `num`. What happens at that point is up to you.

To simulate the circle, we will use a queue. Assume that the child holding the potato will be at the front of the queue. Upon passing the potato, the simulation will simply dequeue and then immediately enqueue that child, putting her at the end of the line. She will then wait until all the others have been at the front before it will be her turn again. After `num` dequeue/enqueue operations, the child at the front will be removed permanently and another cycle will begin. This process will continue until only one name remains (the size of the queue is 1).

A call to the `hotPotato` function using 7 as the counting constant returns `Susan`.

```
import Queue from './queue.js';

function hotPotato(namelist, num) {
  var simqueue = new Queue();

  for (let i = 0; i < namelist.length; i++) {
    simqueue.enqueue(namelist[i]);
  }

  while (simqueue.size() > 1) {
    for (let i = 0; i < num; i++) {
      simqueue.enqueue(simqueue.dequeue());
    }
    simqueue.dequeue();
  }

  return simqueue.dequeue();
}

console.log( hotPotato(['Bill', 'David', 'Susan', 'Jane', 'Kent', 'Brad'],7) );
```

Note that in this example the value of the counting constant is greater than the number of names in the list. This is not a problem since the queue acts like a circle and counting continues back at the beginning until the value is reached. Also, notice that the list is loaded into the queue such that the first name on the list will be at the front of the queue. `Bill` in this case is the first item in the list and therefore moves to the front of the queue.

### 3.3.4 Simulation: Printing Tasks



A more interesting simulation allows us to study the behavior of the printing queue described earlier in this section. Recall that as employees send printing tasks to the shared printer, the tasks are placed in a queue to be processed in a first-come first-served manner. Many questions arise with this configuration. The most important of these might be whether the printer is capable of handling a certain amount of work. If it cannot, employees will be waiting too long for printing and may miss their next meeting.

Consider the following situation in an office. On any average day about 10 employees are working in the office at any given hour. These employees typically print up to twice during that time, and the length of these tasks ranges from 1 to 20 pages. The printer in the lab is older, capable of processing 10 pages per minute of draft quality. The printer could be switched to give better quality, but then it would produce only five pages per minute. The slower printing speed could make employees wait too long. What page rate should be used?

We could decide by building a simulation that models the office. We will need to construct representations for employees, printing tasks, and the printer. As employees submit printing tasks, we will add them to a waiting list, a queue of print tasks attached to the printer. When the printer completes a task, it will look at the queue to see if there are any remaining tasks to process. Of interest for us is the average amount of time employees will wait for their papers to be printed. This is equal to the average amount of time a task waits in the queue.

To model this situation we need to use some probabilities. For example, employees may print a paper from 1 to 20 pages in length. If each length from 1 to 20 is equally likely, the actual length for a print task can be simulated by using a random number between 1 and 20 inclusive. This means that there is equal chance of any length from 1 to 20 appearing.

If there are 10 students in the lab and each prints twice, then there are 20 print tasks per hour on average. What is the chance that at any given second, a print task is going to be created? The way to answer this is to consider the ratio of tasks to time. Twenty tasks per hour means that on average there will be one task every 180 seconds:

$$\frac{20 \text{ tasks}}{1 \text{ hour}} \times \frac{1 \text{ hour}}{60 \text{ minutes}} \times \frac{1 \text{ minute}}{60 \text{ seconds}} = \frac{1 \text{ task}}{180 \text{ seconds}}$$

For every second we can simulate the chance that a print task occurs by generating a random number between 1 and 180 inclusive. If the number is 180, we say a task has been created. Note that it is possible that many tasks could be created in a row or we may wait quite a while for a task to appear. That is the nature of simulation. You want to simulate the real situation as closely as possible given that you know general parameters.

### 3.14.1. Main Simulation Steps

Here is the main simulation.

1. Create a queue of print tasks. Each task will be given a timestamp upon its arrival. The queue is empty to start.
2. For each second (`currentSecond`):
  - i. Does a new print task get created? If so, add it to the queue with the `currentSecond` as the timestamp.
  - ii. If the printer is not busy and if a task is waiting,
    - i. Remove the next task from the print queue and assign it to the printer.
    - ii. Subtract the timestamp from the `currentSecond` to compute the waiting time for that task. Append the waiting time for that task to a list for later processing.
    - iii. Based on the number of pages in the print task, figure out how much time will be required.
  - iii. The printer now does one second of printing if necessary. It also subtracts one second from the time required for that task.
  - iv. If the task has been completed, in other words the time required has reached zero, the printer is no longer busy.
3. After the simulation is complete, compute the average waiting time from the list of waiting times generated.

## 3.14.2 JavaScript implementation

To design this simulation we will create classes for the three real-world objects described above: `Printer`, `Task`, and `PrintQueue`.

The `Printer` class will need to track whether it has a current task. If it does, then it is busy (lines 13–17) and the amount of time needed can be computed from the number of pages in the task. The constructor will also allow the pages-per-minute setting to be initialized. The `tick` method decrements the internal timer and sets the printer to idle (line 11) if the task is completed.

```

class Printer {
    constructor(ppm) {
        this.pageRate = ppm;
        this.currentTask = null;
        this.timeRemaining = 0;
    }

    tick() {
        if (this.currentTask) {
            this.timeRemaining = this.timeRemaining - 1;
            if (this.timeRemaining <= 0) {
                this.currentTask = null;
            }
        }
    }

    busy() {
        if (this.currentTask) {
            return true;
        }
        else {
            return false;
        }
    }

    startNext(newTask) {
        this.currentTask = newTask;
        this.timeRemaining = newTask.getPages() * 60 / this.pageRate;
    }
}

```

The `Task` class will represent a single printing task. When the task is created, a random number generator will provide a length from 1 to 20 pages. We have chosen to use the `Math` methods to generate the random numbers. Here is an example:

```
Math.floor(Math.random() * (42 - 1)) + 1;
```

The logic behind is a simple rule of three:

`Math.random()` returns a `Number` between 0 (inclusive) and 1 (exclusive). So we have an interval like this:

```
[0 .... 1]
```

Now, we'd like a number between `min` (inclusive) and `max` (exclusive):

```
[0 ..... 1]
[min ..... max]
```

We can use the `Math.random` to get the correspondent in the `[min, max]` interval. But, first we should factor a little bit the problem by subtracting `min` from the second interval:

```
[0 ..... 1]
[min - min ..... max - min]
```

This gives:

```
[0 ..... 1]
[0 ..... max - min]
```

We may now apply `Math.random` and then calculate the correspondent. Let's choose a random number:

```

      Math.random()
      |
[0 ..... 1]
[0 ..... max - min]
      |
      x (what we need)

```

So, in order to find `x`, we would do:

```
var x = Math.random() * (max - min);
```

By adding `min` back, so that we get a number in the `[min, max]` interval:

```
var x = Math.random() * (max - min) + min;
```

To get an integer we can use `Math.floor` to have a perfectly even distribution.

```

min.... min+1... min+2 ... max-1... max.... max+1 (is excluded from interval)
|         |         |         |         |         |
|         |         |         |         |         |
|         |         |         |         |         |
min      min+1      ...      max-1      max      ← floor()

```

Now, about our task, each of them will also need to keep a timestamp to be used for computing waiting time. This timestamp will represent the time that the task was created and placed in the printer queue. The `waitTime` method can then be used to retrieve the amount of time spent in the queue before printing begins.

```
class Task {
    constructor(time) {
        this.timestamp = time;
        this.pages = Math.floor(Math.random() * (21 - 1)) + 1;
    }

    getStamp() {
        return this.timestamp;
    }

    getPages() {
        return this.pages;
    }

    waitTime(currentTime) {
        return currentTime - this.timestamp;
    }
}
```

The main simulation implements the algorithm described above. The `printQueue` object is an instance of our existing queue ADT. A boolean helper function, `newPrintTask`, decides whether a new printing task has been created. Print tasks arrive once every 180 seconds. By arbitrarily choosing 180 from the range of random integers (line 32), we can simulate this random event. The simulation function allows us to set the total time and the pages per minute for the printer.

```

import Queue from './queue.js';

function simulation(numSeconds, pagesPerMinute) {
  var printer = new Printer(pagesPerMinute);
  var printQueue = new Queue();
  var waitingTimes = [];

  for (var i = 0; i < numSeconds; i++) {
    var currentSecond = i;
    console.log('currentSecond:', currentSecond);

    if (newPrintTask()) {
      var task = new Task(currentSecond);
      printQueue.enqueue(task);
    }
    if (!printer.busy() && !printQueue.isEmpty()) {
      var nextTask = printQueue.dequeue();
      console.log('nextTask.waitTime(currentSecond):', nextTask.waitTime(current
Second));
      waitingTimes.push(nextTask.waitTime(currentSecond));
      printer.startNext(nextTask);
    }
    printer.tick();
  }

  var sumWaitingTimes = 0;
  for (var i = 0; i < waitingTimes.length; i++) {
    sumWaitingTimes += waitingTimes[i];
  }
  console.log('waitingTimes:', waitingTimes);
  var averageWait = sumWaitingTimes / waitingTimes.length;
  console.log(`Average Wait ${averageWait} secs. ${printQueue.size()} tasks remainin
g`);
}

function newPrintTask() {
  var num = Math.floor(Math.random() * (181 - 1)) + 1;
  if (num === 180) {
    return true;
  }
  else {
    return false;
  }
}

for(var i = 0; i < 10; i++) {
  simulation(3600, 5);
}

```

When we run the simulation, we should not be concerned that the results are different each time. This is due to the probabilistic nature of the random numbers. We are interested in the trends that may be occurring as the parameters to the simulation are adjusted. Here are some results.

First, we will run the simulation for a period of 60 minutes (3,600 seconds) using a page rate of five pages per minute. In addition, we will run 10 independent trials. Remember that because the simulation works with random numbers each run will return different results.

```
for(var i = 0; i < 10; i++) {  
    simulation(3600, 5);  
}  
// Average Wait 63 secs. 0 tasks remaining  
// Average Wait 155.76923076923077 secs. 0 tasks remaining  
// Average Wait 51.23076923076923 secs. 1 tasks remaining  
// Average Wait 95.8 secs. 1 tasks remaining  
// Average Wait 321.29411764705884 secs. 0 tasks remaining  
// Average Wait 18.45 secs. 0 tasks remaining  
// Average Wait 18.785714285714285 secs. 0 tasks remaining  
// Average Wait 141.2 secs. 1 tasks remaining  
// Average Wait 171.41176470588235 secs. 3 tasks remaining  
// Average Wait 240.2173913043478 secs. 0 tasks remaining
```

After running our 10 trials we can see that the mean average wait time is 127.71 seconds. You can also see that there is a large variation in the average weight time with a minimum average of 18.45 seconds and a maximum of 321.29 seconds. You may also notice that in only 6 of the cases were all the tasks completed.

Now, we will adjust the page rate to 10 pages per minute, and run the 10 trials again, with a faster page rate our hope would be that more tasks would be completed in the one hour time frame.

```
for(var i = 0; i < 10; i++) {  
    simulation(3600, 10);  
}  
// Average Wait 57.65384615384615 secs. 0 tasks remaining  
// Average Wait 25 secs. 2 tasks remaining  
// Average Wait 9.45 secs. 0 tasks remaining  
// Average Wait 1 secs. 0 tasks remaining  
// Average Wait 4.894736842105263 secs. 0 tasks remaining  
// Average Wait 10.235294117647058 secs. 0 tasks remaining  
// Average Wait 6.6521739130434785 secs. 0 tasks remaining  
// Average Wait 3.3529411764705883 secs. 0 tasks remaining  
// Average Wait 24.473684210526315 secs. 0 tasks remaining  
// Average Wait 22.41176470588235 secs. 0 tasks remaining
```

### 3.14.3 Discussion

We were trying to answer a question about whether the current printer could handle the task load if it were set to print with a better quality but slower page rate. The approach we took was to write a simulation that modeled the printing tasks as random events of various lengths and arrival times.

The output above shows that with 5 pages per minute printing, the average waiting time varied from a low of 18.45 seconds to a high of 321.29 seconds (about 5 minutes). With a faster printing rate, the low value was 1 second with a high of only 57. In addition, in 4 out of 10 runs at 5 pages per minute there were print tasks still waiting in the queue at the end of the hour.

Therefore, we are perhaps persuaded that slowing the printer down to get better quality may not be a good idea. Employees cannot afford to wait that long for their papers, especially when they need to be getting on to their next meeting. A five-minute wait would simply be too long.

This type of simulation analysis allows us to answer many questions, commonly known as “what if” questions. All we need to do is vary the parameters used by the simulation and we can simulate any number of interesting behaviors. For example:

- What if the enrollment goes up and the average number of employees increase by 20?
- What if the direction decide that the Monday will be meeting-free? Can they afford to wait?
- What if the size of the average print task decreases?

These questions could all be answered by modifying the above simulation. However, it is important to remember that the simulation is only as good as the assumptions that are used to build it. Real data about the number of print tasks per hour and the number of students per hour was necessary to construct a robust simulation.



## 3.4 Deque

A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.

### 3.4.1 The Deque Abstract Data Type

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

- `Deque()` create a new deque that is empty. It need no parameters and returns an empty deque.
- `addFront(item)` adds a new item in the front of the deque. It needs the item and returns nothing.
- `addRear(item)` adds a new item to the rear of the deque. It needs the item and returns nothing.
- `removeFront()` removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- `removeRear()` removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- `isEmpty()` tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the deque. It needs no parameters and returns an integer.

As an example, if we assume that `d` is a deque that has been created and is currently empty, then the following table shows the results of a sequence of deque operations. Note that the contents in front are listed on the right. It is very important to keep track of the front and the rear as you move items in and out of the collection as things can get a bit confusing.

Deque Operation	Deque Contents	Return Value
d.isEmpty()	[ ]	true
d.addRear(4)	[ 4 ]	
d.addRear('banana')	[ 'banana', 4 ]	
d.addRear('apple')	[ 'banana', 4, 'apple' ]	
d.addFront(true)	[ 'banana', 4, 'apple', true ]	
d.size()	[ 'banana', 4, 'apple', true ]	4
d.isEmpty()	[ 'banana', 4, 'apple', true ]	false
d.addRear(8.4)	[ 8.4, 'banana', 4, 'apple', true ]	
d.removeRear()	[ 'banana', 4, 'apple', true ]	8.4
d.removeFront()	[ 'banana', 4, 'apple' ]	true

## 3.4.2 Implementing a Deque in JavaScript

As we have done in previous sections, we will create a new class for the implementation of the abstract data type deque. Again, the JavaScript array will provide a very nice set of methods upon which to build the details of the deque. Our implementation will assume that the rear of the deque is at position 0 in the list.

```

class Deque {
  constructor() {
    this.items = [];
  }

  isEmpty() {
    return !Boolean(this.items.length);
  }

  addFront(item) {
    this.items.unshift(item);
  }

  addRear(item) {
    this.items.push(item);
  }

  removeFront() {
    return this.items.shift();
  }

  removeRear() {
    return this.items.pop();
  }

  size() {
    return this.items.length;
  }
}

d = new Deque();
console.log( d.isEmpty() );
d.addRear(4);
d.addRear('banana');
d.addFront('apple');
d.addFront(true);
console.log( d.size() );
console.log( d.isEmpty() );
d.addRear(8.4);
console.log( d.removeRear() );
console.log( d.removeFront() );

```

In `removeFront` we use the `shift` method to remove the last element from the list. However, in `removeRear`, the `pop` method must remove the first element of the list. Likewise, we need to use the `push` method in `addRear` since the `unshift` method assumes the addition of a new element to the end of the list.

You can see many similarities to JavaScript code already described for stacks and queues. You are also likely to observe that in this implementation adding and removing items from the front is  $\$O(1)\$$  whereas adding and removing from the rear is  $\$O(n)\$$ . This is to be

expected given the common operations that appear for adding and removing items. Again, the important thing is to be certain that we know where the front and rear are assigned in the implementation.

### 3.4.3 Palindrome-Checker

An interesting problem that can be easily solved using the deque data structure is the classic palindrome problem. A **palindrome** is a string that reads the same forward and backward, for example, *radar*, *toot*, and *madam*. We would like to construct an algorithm to input a string of characters and check whether it is a palindrome.

The solution to this problem will use a deque to store the characters of the string. We will process the string from left to right and add each character to the rear of the deque. At this point, the deque will be acting very much like an ordinary queue. However, we can now make use of the dual functionality of the deque. The front of the deque will hold the first character of the string and the rear of the deque will hold the last character.

Since we can remove both of them directly, we can compare them and continue only if they match. If we can keep matching first and the last items, we will eventually either run out of characters or be left with a deque of size 1 depending on whether the length of the original string was even or odd. In either case, the string must be a palindrome. Here is the complete function for palindrome-checking:

```
import Deque from './deque.js';

function palChecker(str) {
  var charDeque = new Deque();

  for (var i = 0; i < str.length; i++) {
    charDeque.addRear(str[i]);
  }

  var stillEqual = true;

  while (charDeque.size() > 1 && stillEqual) {
    var first = charDeque.removeFront();
    var last = charDeque.removeRear();

    if (first !== last) {
      stillEqual = false;
    }
  }

  return stillEqual;
}

console.log( palChecker('lsdkjfskf') );
console.log( palChecker('radar') );
```

## 3.5 List

Throughout the discussion of basic data structures, we have used JavaScript array to implement the abstract data types presented. The array is the implementation of a data structure called **list**. This is a powerful, yet simple, collection mechanism that provides the programmer with a wide variety of operations. However, not all programming languages include a list collection. In these cases, the notion of a list must be implemented by the programmer.

A **list** is a collection of items where each item holds a relative position with respect to the others. More specifically, we will refer to this type of list as an unordered list. We can consider the list as having a first item, a second item, a third item, and so on. We can also refer to the beginning of the list (the first item) or the end of the list (the last item). For simplicity we will assume that lists cannot contain duplicate items.

For example, the collection of integers 54, 26, 93, 17, 77, and 31 might represent a simple unordered list of exam scores. Note that we have written them as comma-delimited values, a common way of showing the list structure. Of course, JavaScript would show this list as [54,26,93,17,77,31].

### 3.5.1 The Unordered List Abstract Data Type

The structure of an unordered list, as described above, is a collection of items where each item holds a relative position with respect to the others. Some possible unordered list operations are given below.

- `List()` create a new list that is empty. It needs no parameters and returns an empty list.
- `add(item)` adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- `remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- `search(item)` searches for the item in the list. It needs the item and returns a boolean value.
- `isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the list. It needs no parameters and returns an integer.
- `append(item)` adds a new item to the end of the list making it the last item in the

collection. It needs the item and returns nothing. Assume the item is not already in the list.

- `index(item)` returns the position of an item in the list. It needs the item and returns the index. Assume the item is in the list.
- `insert(pos, item)` adds a new item to the list at position `pos`. It needs the item and the position and returns nothing. Assume the item is not already in the list and there are enough existing items to have position `pos`.
- `pop()` removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- `pop(pos)` removes and returns the item at position `pos`. It needs the position and returns the item. Assume the item is in the list.

## 3.5.2 Implementing an Unordered List: Linked Lists

In order to implement an unordered list, we will construct what is commonly known as a **linked list**. Recall that we need to be sure that we can maintain the relative positioning of the items. However, there is no requirement that we maintain that positioning in contiguous memory. If we can maintain some explicit information in each item, namely the location of the next item, then the relative position of each item can be expressed by simply following the link from one item to the next.

It is important to note that the location of the first item of the list must be explicitly specified. Once we know where the first item is, the first item can tell us where the second is, and so on. The external reference is often referred to as the **head** of the list. Similarly, the last item needs to know that there is no next item.

### 3.5.2.1 The Node Class

The basic building block for the linked list implementation is the **node**. Each node object must hold at least two pieces of information. First, the node must contain the list item itself. We will call this the **data field** of the node. In addition, each node must hold a reference to the next node. To construct a node, you need to supply the initial data value for the node. Evaluating the assignment statement below will yield a node object containing the value 93. The `Node` class also includes the usual methods to access and modify the data and the next reference.

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }

  getData() {
    return this.data;
  }

  getNext() {
    return this.next;
  }

  setData(newData) {
    this.data = newData;
  }

  setNext(nextNode) {
    this.next = nextNode;
  }
}

var myNode = new Node(93);
console.log( myNode.getData() ); // 93
```

The JavaScript primitive values `null` will play an important role in the `Node` class and later in the linked list itself. A reference to `null` will denote the fact that there is no next node. Note in the constructor that a node is initially created with `next` set to `null`. This is sometimes referred to as "grounding the node". It is always a good idea to explicitly assign `null` to your initial next reference values.

### 3.5.2.2 The Unordered List Class

As we suggested above, the unordered list will be built from a collection of nodes, each linked to the next by explicit references. As long as we know where to find the first node (containing the first item), each item after that can be found by successively following the next links. With this in mind, the `UnorderedList` class must maintain a reference to the first node. Here is the constructor:



```
class UnorderedList {  
    constructor() {  
        this.head = null;  
    }  
}  
  
var mylist = new UnorderedList();
```

As we discussed in the Node class, the special reference `null` will again be used to state that the head of the list does not refer to anything. The head of the list refers to the first node which contains the first item of the list. In turn, that node holds a reference to the next node (the next item) and so on. It is very important to note that the list class itself does not contain any node objects. Instead it contains a single reference to only the first node in the linked structure.

## isEmpty()

The `isEmpty` method simply checks to see if the head of the list is a reference to `null`. The result of the boolean expression `this.head === null` will only be true if there are no nodes in the linked list. Since a new list is empty, the constructor and the check for empty must be consistent with one another. This shows the advantage to using the reference `null` to denote the "end" of the linked structure.

```
isEmpty() {  
    return this.head === null;  
}
```

## add()

So, how do we get items into our list? We need to implement the `add` method. However, before we can do that, we need to address the important question of where in the linked list to place the new item. Since this list is unordered, the specific location of the new item with respect to the other items already in the list is not important. The new item can go anywhere. With that in mind, it makes sense to place the new item in the easiest location possible.

Recall that the linked list structure provides us with only one entry point, the head of the list. All of the other nodes can only be reached by accessing the first node and then following `next` links. This means that the easiest place to add the new node is right at the head, or beginning, of the list. In other words, we will make the new item the first item of the list and the existing items will need to be linked to this new first item so that they follow. Here is the implementation of the `add` method:

```
add(item) {  
    var newNode = new Node(item);  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

Each item of the list must reside in a node object. Line 2 creates a new node and places the item as its data. Now we must complete the process by linking the new node into the existing structure. This requires two steps. Step 1 (line 3) changes the `next` reference of the new node to refer to the old first node of the list. Now that the rest of the list has been properly attached to the new node, we can modify the head of the list to refer to the new node. The assignment statement in line 4 sets the head of the list.

The order of the two steps described above is very important. What happens if the order of line 3 and line 4 is reversed? If the modification of the head of the list happens first, since the head was the only external reference to the list nodes, all of the original nodes are lost and can no longer be accessed.

Let's build a new list by calling the add method a number of times:

```
var myList = new UnorderedList();  
  
myList.add(31);  
myList.add(77);  
myList.add(17);  
myList.add(93);  
myList.add(26);  
myList.add(54);
```

Since 31 is the first item added to the list, it will eventually be the last node on the linked list as every other item is added ahead of it. Also, since 54 is the last item added, it will become the data value in the first node of the linked list.

The next methods that we will implement ( `size` , `search` , and `remove` ) are all based on a technique known as **linked list traversal**. Traversal refers to the process of systematically visiting each node. To do this we use an external reference that starts at the first node in the list. As we visit each node, we move the reference to the next node by "traversing" the next reference.

## size()

To implement the `size` method, we need to traverse the linked list and keep a count of the number of nodes that occurred.

```
size() {  
    var current = this.head;  
    var count = 0;  
    while (current !== null) {  
        count = count + 1;  
        current = current.getNext();  
    }  
  
    return count;  
}
```

The external reference is called `current` and is initialized to the head of the list in line 2. At the start of the process we have not seen any nodes so the count is set to 0. Lines 4–6 actually implement the traversal. As long as the current reference has not seen the end of the list ( `null` ), we move current along to the next node via the assignment statement in line 6. Every time current moves to a new node, we add 1 to `count` . Finally, `count` gets returned after the iteration stops.

## search()

Searching for a value in a linked list implementation of an unordered list also uses the traversal technique. As we visit each node in the linked list we will ask whether the data stored there matches the item we are looking for. In this case, however, we may not have to traverse all the way to the end of the list. In fact, if we do get to the end of the list, that means that the item we are looking for must not be present. So, if we do find the item, there is no need to continue.

```
search(item) {  
    var current = this.head;  
    var found = false;  
    while (current !== null && !found) {  
        if (current.getData() === item) {  
            found = true;  
        }  
        else {  
            current = current.getNext();  
        }  
    }  
  
    return found;  
}
```

As in the `size` method, the traversal is initialized to start at the head of the list (line 2). We also use a boolean variable called `found` to remember whether we have located the item we are searching for. Since we have not found the item at the start of the traversal, `found`

can be set to `false` (line 3). The iteration in line 4 takes into account both conditions discussed above. As long as there are more nodes to visit and we have not found the item we are looking for, we continue to check the next node. The question in line 5 asks whether the data item is present in the current node. If so, `found` can be set to `true`.

## **remove()**

The `remove` method requires two logical steps. Once we find the item (recall that we assume it is present), we must remove it. The first step is very similar to `search`. Starting with an external reference set to the head of the list, we traverse the links until we discover the item we are looking for. Since we assume that item is present, we know that the iteration will stop before `current` gets to `null`. This means that we can simply use the boolean `found` in the condition.

When `found` becomes `true`, `current` will be a reference to the node containing the item to be removed. But how do we remove it? One possibility would be to replace the value of the item with some marker that suggests that the item is no longer present. The problem with this approach is the number of nodes will no longer match the number of items. It would be much better to remove the item by removing the entire node.

In order to remove the node containing the item, we need to modify the link in the previous node so that it refers to the node that comes after `current`. Unfortunately, there is no way to go backward in the linked list. Since `current` refers to the node ahead of the node where we would like to make the change, it is too late to make the necessary modification.

The solution to this dilemma is to use two external references as we traverse down the linked list. `current` will behave just as it did before, marking the current location of the traverse. The new reference, which we will call `previous`, will always travel one node behind `current`. That way, when `current` stops at the node to be removed, `previous` will be referring to the proper place in the linked list for the modification.

```
remove(item) {
    var current = this.head;
    var previous = null;
    var found = false;
    while (!found) {
        if (current.getData() === item) {
            found = true;
        }
        else {
            previous = current;
            current = current.getNext();
        }
    }

    if (previous === null) {
        this.head = current.getNext();
    }
    else {
        previous.setNext(current.getNext());
    }
}
```

Lines 2–3 assign initial values to the two references. Note that `current` starts out at the list head as in the other traversal examples. `previous`, however, is assumed to always travel one node behind `current`. For this reason, `previous` starts out with a value of `null` since there is no node before the head. The boolean variable `found` will again be used to control the iteration.

In lines 6–7 we ask whether the item stored in the current node is the item we wish to remove. If so, `found` can be set to `true`. If we do not find the item, `previous` and `current` must both be moved one node ahead. Again, the order of these two statements is crucial. `previous` must first be moved one node ahead to the location of `current`. At that point, `current` can be moved. This process is often referred to as "inch-worming" as `previous` must catch up to `current` before `current` moves ahead.

Once the searching step of the `remove` has been completed, we need to remove the node from the linked list. However, there is a special case that needs to be addressed. If the item to be removed happens to be the first item in the list, then `current` will reference the first node in the linked list. This also means that `previous` will be `null`. We said earlier that `previous` would be referring to the node whose next reference needs to be modified in order to complete the remove. In this case, it is not `previous` but rather the head of the list that needs to be changed.

Line 15 allows us to check whether we are dealing with the special case described above. If `previous` did not move, it will still have the value `null` when the boolean `found` becomes `true`. In that case (line 16) the head of the list is modified to refer to the node after the

current node, in effect removing the first node from the linked list. However, if `previous` is not `null`, the node to be removed is somewhere down the linked list structure. In this case the `previous` reference is providing us with the node whose next reference must be changed. Line 19 uses the `setNext` method from `previous` to accomplish the removal. Note that in both cases the destination of the reference change is `current.getNext()`. One question that often arises is whether the two cases shown here will also handle the situation where the item to be removed is in the last node of the linked list. We leave that for you to consider.

The remaining methods `append`, `insert`, `index`, and `pop` are left as exercises. Remember that each of these must take into account whether the change is taking place at the head of the list or someplace else. Also, `insert`, `index`, and `pop` require that we name the positions of the list. We will assume that position names are integers starting with 0.

### 3.5.3 The Ordered List Abstract Data Type

We will now consider a type of list known as an ordered list. For example, if the list of integers shown above were an ordered list (ascending order), then it could be written as 17, 26, 31, 54, 77, and 93. Since 17 is the smallest item, it occupies the first position in the list. Likewise, since 93 is the largest, it occupies the last position.

The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item. The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined. Many of the ordered list operations are the same as those of the unordered list.

- `OrderedList()` create a new ordered list that is empty. It needs no parameters and returns an empty list.
- `add(item)` adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
- `remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- `search(item)` searches for the item in the list. It needs the item and returns a boolean value.
- `isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the list. It needs no parameters and returns an integer.
- `index(item)` returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- `pop()` removes and returns the last item in the list. It needs nothing and returns an item.

Assume the list has at least one item.

- `pop(pos)` remove and returns the item at position `pos`. It needs the position and returns the item. Assume the item is in the list.

### 3.5.4 Implementing and ordered list

To implement the `orderedList` class, we will use the same technique as seen previously with unordered lists. Once again, an empty list will be denoted by a head reference to `null`.

```
class OrderedList {  
    constructor() {  
        this.head = null;  
    }  
}
```

As we consider the operations for the ordered list, we should note that the `isEmpty` and `size` methods can be implemented the same as with unordered lists since they deal only with the number of nodes in the list without regard to the actual item values. Likewise, the `remove` method will work just fine since we still need to find the item and then link around the node to remove it. The two remaining methods, `search` and `add`, will require some modification.

#### **search()**

The search of an unordered linked list required that we traverse the nodes one at a time until we either find the item we are looking for or run out of nodes ( `null` ). It turns out that the same approach would actually work with the ordered list and in fact in the case where we find the item it is exactly what we need. However, in the case where the item is not in the list, we can take advantage of the ordering to stop the search as soon as possible.

```

search(item) {
    var current = this.head;
    var found = false;
    var stop = false;
    while (current !== null && !found && !stop) {
        if (current.getData() === item) {
            found = true;
        }
        else {
            if (current.getData() > item) {
                stop = true;
            }
            else {
                current = current.getNext();
            }
        }
    }

    return found
}

```

It is easy to incorporate the new condition discussed above by adding another boolean variable, `stop`, and initializing it to `false` (line 4). While `stop` is `false` we can continue to look forward in the list (line 5). If any node is ever discovered that contains data greater than the item we are looking for, we will set `stop` to `true` (lines 10–11). The remaining lines are identical to the unordered list search.

## add()

The most significant method modification will take place in `add`. Recall that for unordered lists, the `add` method could simply place a new node at the head of the list. It was the easiest point of access. Unfortunately, this will no longer work with ordered lists. It is now necessary that we discover the specific place where a new item belongs in the existing ordered list.

Assume we have the ordered list consisting of 17, 26, 54, 77, and 93 and we want to add the value 31. The `add` method must decide that the new item belongs between 26 and 54. As we explained earlier, we need to traverse the linked list looking for the place where the new node will be added. We know we have found that place when either we run out of nodes ( `current` becomes `null` ) or the value of the current node becomes greater than the item we wish to add.

As we saw with unordered lists, it is necessary to have an additional reference, again called `previous`, since `current` will not provide access to the node that must be modified.



```

add(item) {
    var current = this.head;
    var previous = null;
    var stop = false;
    while (current !== null && !stop) {
        if (current.getData() > item) {
            stop = false;
        }
        else {
            previous = current;
            current = current.getNext();
        }
    }

    var temp = new Node(item);
    if (previous === null) {
        temp.setNext(this.head);
        this.head = temp;
    }
    else {
        temp.setNext(current);
        previous.setNext(temp);
    }
}

```

Lines 2–3 set up the two external references and lines 10–11 again allow `previous` to follow one node behind `current` every time through the iteration. The condition (line 5) allows the iteration to continue as long as there are more nodes and the value in the `current` node is not larger than the item. In either case, when the iteration fails, we have found the location for the new node.

Once a new node has been created for the item, the only remaining question is whether the new node will be added at the beginning of the linked list or some place in the middle. Again, `previous === null` (line 16) can be used to provide the answer.

We leave the remaining methods as exercises. You should carefully consider whether the unordered implementations will work given that the list is now ordered.

### 3.5.5 Analysis of Linked Lists

To analyze the complexity of the linked list operations, we need to consider whether they require traversal. Consider a linked list that has  $n$  nodes. The `isEmpty` method is  $O(1)$  since it requires one step to check the head reference for `null`. `size`, on the other hand, will always require  $n$  steps since there is no way to know how many nodes are in the linked list without traversing from head to end so it is  $O(n)$ . Adding an item to

an unordered list will always be  $O(1)$  since we simply place the new node at the head of the linked list. However, `search` and `remove`, as well as `add` for an ordered list, all require the traversal process. Although on average they may need to traverse only half of the nodes, these methods are all  $O(n)$  since in the worst case each will process every node in the list.

# Recursion

## Objectives

- To understand that complex problems that may otherwise be difficult to solve may have a simple recursive solution.
- To learn how to formulate programs recursively.
- To understand and apply the three laws of recursion.
- To understand recursion as a form of iteration.
- To implement the recursive formulation of a problem.
- To understand how recursion is implemented by a computer system.

## Chapters

-

# 4.1 What is Recursion?

**Recursion** is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

## 4.2 Calculating the Sum of a List of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a list of numbers such as: [1, 3, 5, 7, 9]. We will use an accumulator variable ( `theSum` ) to compute a running total of all the numbers in the list by starting with 00 and adding each number in the list.

```
function listSum(numList) {
  var theSum = 0;
  for (var i = 0; i < numList.length; i++) {
    theSum += numList[i];
  }
  return theSum;
}

console.log( listSum([1, 3, 5, 7, 9]) );
```

Pretend for a minute that you do not have `while` loops or `for` loops. How would you compute the sum of a list of numbers? If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers. To redefine the problem from adding a list to adding pairs of numbers, we could rewrite the list as a fully parenthesized expression. Such an expression looks like this:  $((((1 + 3) + 5) + 7) + 9)$ .

We can also parenthesize the expression the other way around:  $(1 + (3 + (5 + (7 + 9))))$ .

Notice that the innermost set of parentheses,  $(7+9)$ , is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

$$\begin{aligned} \text{total} &= (1 + (3 + (5 + (7 + 9)))) \\ \text{total} &= (1 + (3 + (5 + 16))) \\ \text{total} &= (1 + (3 + 21)) \\ \text{total} &= (1 + 24) \\ \text{total} &= 25 \end{aligned}$$

How can we take this idea and turn it into a JavaScript program? First, let's restate the sum problem in terms of JavaScript arrays. We might say the the sum of the array `numList` is the sum of the first element of the array ( `numList[0]` ), and the sum of the numbers in the rest of the array ( `numList[1:]` ). To state it in a functional form:

$$\text{listSum}(\text{numList}) = \text{first}(\text{numList}) + \text{listSum}(\text{rest}(\text{numList})) \quad \text{\label{eqn:listsun}}$$

In this equation  $\text{first}(\text{numList})$  returns the first element of the list and  $\text{rest}(\text{numList})$  returns a list of everything but the first element. This is easily expressed in JavaScript as shown below:

```
function listSum(numList) {  
  if (numList.length === 1) {  
    return numList[0];  
  }  
  else {  
    numList.shift();  
    return numList[0] + listSum(numList);  
  }  
}  
  
console.log( listSum([1, 3, 5, 7, 9]) );
```

There are a few key ideas in this code to look at. First, on line 2 we are checking to see if the list is one element long. This check is crucial and is our escape clause from the function. The sum of a list of length 1 is trivial; it is just the number in the list. Second, on line 7 our function calls itself! This is the reason that we call the listsum algorithm recursive. A recursive function is a function that calls itself. You should think of this series of calls as a series of simplifications.

Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller. When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved.

## 4.3 The Three Laws of Recursion

Like the robots of Asimov, all recursive algorithms must obey three important laws. Let's look at each one of these laws in more detail and see how it was used in the `listsum` algorithm.

### **A recursive algorithm must have a base case**

A base case is the condition that allows the algorithm to stop recursing. A base case is typically a problem that is small enough to solve directly. In the `listsum` algorithm the base case is a list of length 1.

### **A recursive algorithm must change its state and move toward the base case**

This second law states that we must arrange for a change of state that moves the algorithm toward the base case. A change of state means that some data that the algorithm is using is modified. Usually the data that represents our problem gets smaller in some way. In the `listsum` algorithm our primary data structure is a list, so we must focus our state-changing efforts on the list. Since the base case is a list of length 1, a natural progression toward the base case is to shorten the list. This is exactly what happens on line 7 when we call `listsum` with a shorter list.

### **A recursive algorithm must call itself, recursively.**

The final law is that the algorithm must call itself. This is the very definition of recursion. Recursion is a confusing concept to many beginning programmers. As a novice programmer, you have learned that functions are good because you can take a large problem and break it up into smaller problems. The smaller problems can be solved by writing a function to solve each problem. When we talk about recursion it may seem that we are talking ourselves in circles. We have a problem to solve with a function, but that function solves the problem by calling itself! But the logic is not circular at all; the logic of recursion is an elegant expression of solving a problem by breaking it down into a smaller and easier problems.

## 4.4 Converting an Integer to a String in Any Base

Suppose you want to convert an integer to a string in some base between binary and hexadecimal. For example, convert the integer 10 to its string representation in decimal as "10", or to its string representation in binary as "1010". While there are many algorithms to solve this problem, including the algorithm discussed in the stack section, the recursive formulation of the problem is very elegant.

Let's look at a concrete example with the number 769 to convert using base 10. Suppose we have a sequence of characters corresponding to the first 10 digits, like `var convString = '0123456789'`. It is easy to convert a number less than 10 to its string equivalent by looking it up in the sequence. For example, if the number is 9, then the string is `convString[9]` which gives us '9'. If we can arrange to break up the number 769 into three single-digit numbers, 7, 6, and 9, then converting it to a string is simple. A number less than 10 sounds like a good base case.

Knowing what our base is suggests that the overall algorithm will involve three components:

1. Reduce the original number to a series of single-digit numbers.
2. Convert the single digit-number to a string using a lookup.
3. Concatenate the single-digit strings together to form the final result.

The next step is to figure out how to change state and make progress toward the base case. Since we are working with an integer, let's consider what mathematical operations might reduce a number. The most likely candidates are division and subtraction. While subtraction might work, it is unclear what we should subtract from what. Integer division with remainders gives us a clear direction. Let's look at what happens if we divide a number by the base we are trying to convert to.

Using integer division to divide 769 by 10, we get 76 with a remainder of 9. This gives us two good results. First, the remainder is a number less than our base that can be converted to a string immediately by lookup. Second, we get a number that is smaller than our original and moves us toward the base case of having a single number less than our base. Now our job is to convert 76 to its string representation. Again we will use integer division plus remainder to get results of 7 and 6 respectively. Finally, we have reduced the problem to converting 7, which we can do easily since it satisfies the base case condition of  $n < \text{base}$ , where  $\text{base} = 10$ .



Here is the JavaScript code that implement the algorithm outlined above for any base between 2 and 16:

```
function toStr(n, base) {  
  var convertString = '0123456789ABCDEF';  
  if (n < base) {  
    return convertString[n];  
  }  
  else {  
    return toStr(Math.floor(n/base), base) + convertString[n%base];  
  }  
}  
  
console.log( toStr(1453, 16) );
```

Notice that in line 3 we check for the base case where `n` is less than the base we are converting to. When we detect the base case, we stop recursing and simply return the string from the `convertString` sequence. In line 6 we satisfy both the second and third laws—by making the recursive call and by reducing the problem size—using division.

## 4.5 Stack Frames: Implementing Recursion

Suppose that instead of concatenating the result of the recursive call to `toStr` with the string from `convertString`, we modified our algorithm to push the strings onto a stack prior to making the recursive call.

```
import Stack from './stack.js';

var rStack = new Stack();

function toStr(n, base) {
  var convertString = '0123456789ABCDEF';
  while (n > 0) {
    if (n < base) {
      rStack.push(convertString[n]);
    }
    else {
      rStack.push(convertString[n % base]);
    }
    n = Math.floor(n/base);
  }
  var res = '';
  while (!rStack.isEmpty()) {
    res = res + rStack.pop();
  }
  return res;
}

console.log( toStr(1453, 16) ); // 5AD
```

Each time we make a call to `toStr`, we push a character on the stack. Notice that now we can simply pop the characters off the stack and concatenate them into the final result, "1010".

The stack frames also provide a scope for the variables used by the function. Even though we are calling the same function over and over, each call creates a new scope for the variables that are local to the function.

## 4.6 Introduction: Visualizing Recursion

In the previous section we looked at some problems that were easy to solve using recursion; however, it can still be difficult to find a mental model or a way of visualizing what is happening in a recursive function. This can make recursion difficult for people to grasp. In this section we will look at a couple of examples of using recursion to draw some interesting pictures. As you watch these pictures take shape you will get some new insight into the recursive process that may be helpful in cementing your understanding of recursion.

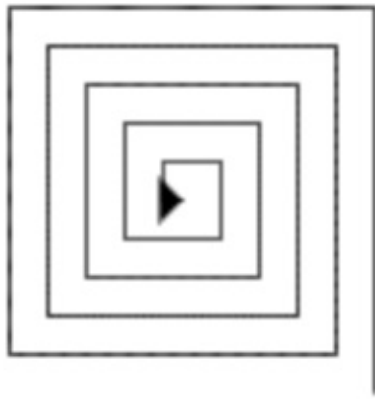
For our illustrations we will use a simple code I made available at <https://github.com/pmary/js-turtle-graphic>. This, allow us to use simple commands similar to the Python's turtle graphics module. The metaphor is quite simple. You can create a turtle and the turtle can move forward, backward, turn left, turn right, etc. The turtle can have its tail up or down. When the turtle's tail is down and the turtle moves it draws a line as it moves.

We will use the turtle class in `turtle.js` to draw a spiral recursively.

```
var canvas = document.getElementById('c');
var arrowCanvas = document.getElementById('a');
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
arrowCanvas.width = window.innerWidth;
arrowCanvas.height = window.innerHeight;

var turtle = new Turtle(canvas, arrowCanvas);

function drawSpiral(turtle, lineLen) {
  if (lineLen > 0) {
    turtle.forward(lineLen);
    turtle.left(90);
    drawSpiral(turtle, lineLen-5);
  }
}
drawSpiral(turtle, 100);
turtle.draw();
```



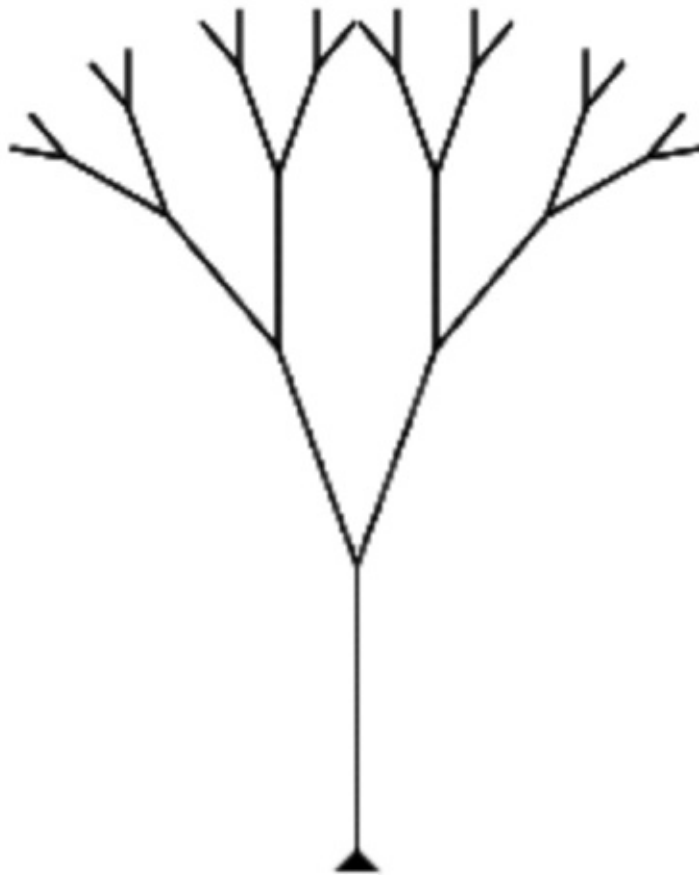
We first set the size of two canvas, one is to draw the lines and the other one is to draw the arrow that will represent our turtle. Then, we create a turtle and define the `drawSpiral` function. The base case for this simple function is when the length of the line we want to draw is reduced to zero or less. If the length of the line is longer than zero we instruct the turtle to go forward by `lineLen` units and then turn right 90 degrees. The recursive step is when we call `drawSpiral` again with a reduced length. At the end you will notice that we call the function `turtle.draw()`. This method start the animated drawing process of the previous instructions.

`forward`, `backward`, `left` and `right` methods is all that you need to know in order to make some pretty impressive drawings. For our next program we are going to draw a fractal tree. Fractals come from a branch of mathematics, and have much in common with recursion. The definition of a fractal is that when you look at it the fractal has the same basic shape no matter how much you magnify it. Some examples from nature are the coastlines of continents, snowflakes, mountains, and even trees or shrubs. The fractal nature of many of these natural phenomenon makes it possible for programmers to generate very realistic looking scenery for computer generated movies. In our next example we will generate a fractal tree.

To understand how this is going to work it is helpful to think of how we might describe a tree using a fractal vocabulary. Remember that we said above that a fractal is something that looks the same at all different levels of magnification. If we translate this to trees and shrubs we might say that even a small twig has the same shape and characteristics as a whole tree. Using this idea we could say that a *tree* is a trunk, with a smaller tree going off to the right and another smaller *tree* going off to the left. If you think of this definition recursively it means that we will apply the recursive definition of a tree to both of the smaller left and right trees.

Let's translate this idea to some JavaScript code:

```
function tree(branchLen, turtle) {  
  if (branchLen > 5) {  
    turtle.forward(branchLen);  
    turtle.right(20);  
    tree(branchLen - 15, turtle);  
    turtle.left(40);  
    tree(branchLen - 15, turtle);  
    turtle.right(20);  
    turtle.backward(branchLen);  
  }  
}
```



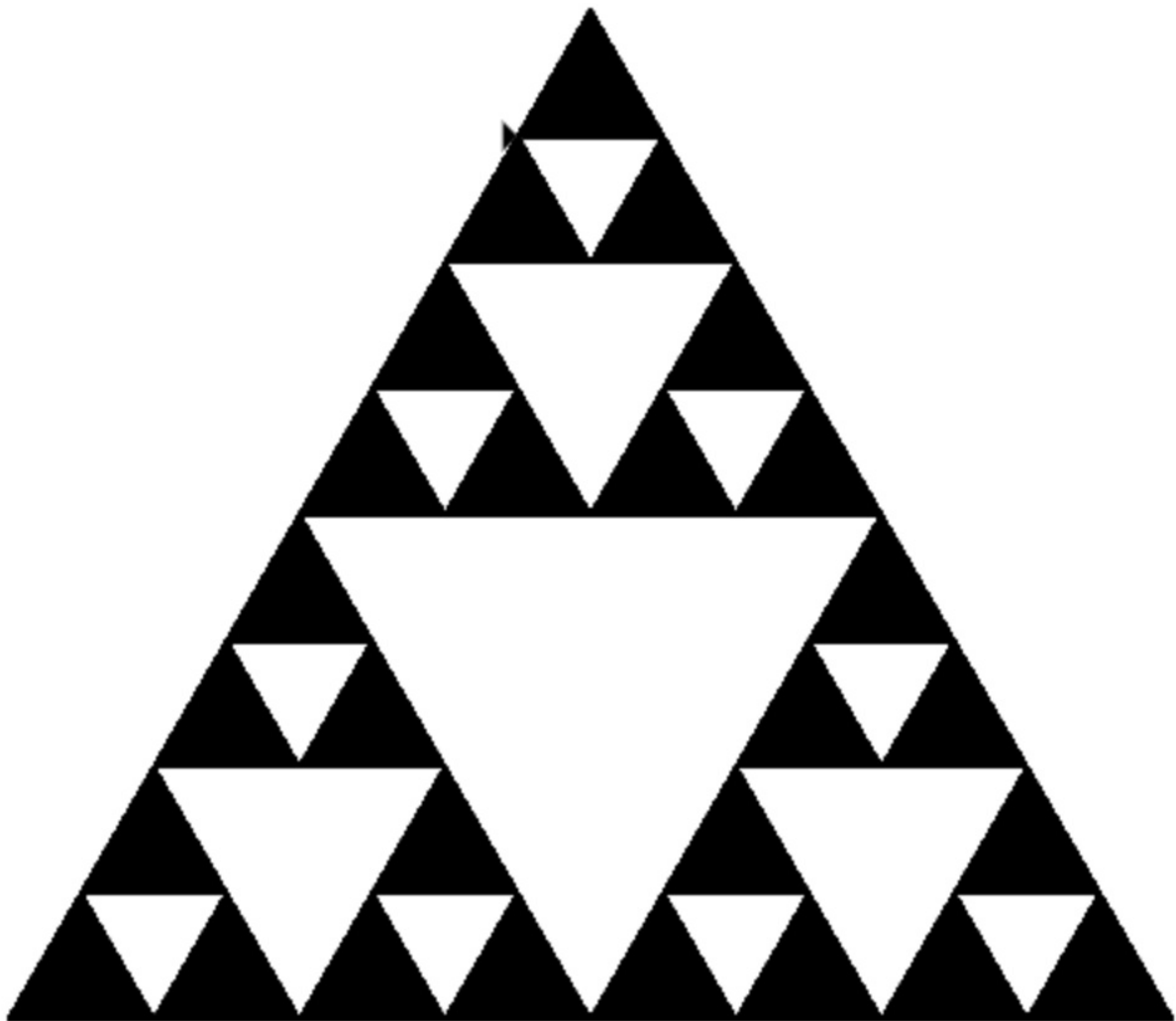
Let's look at the code a bit more closely. You will see that on lines 5 and 7 we are making a recursive call. On line 5 we make the recursive call right after the turtle turns to the right by 20 degrees; this is the right tree mentioned above. Then in line 7 the turtle makes another recursive call, but this time after turning left by 40 degrees. The reason the turtle must turn left by 40 degrees is that it needs to undo the original 20 degree turn to the right and then do an additional 20 degree turn to the left in order to draw the left tree. Also notice that each time we make a recursive call to `tree` we subtract some amount from the `branchLen`

parameter; this is to make sure that the recursive trees get smaller and smaller. You should also recognize the initial `if` statement on line 2 as a check for the base case of `branchLen` getting too small.

By running the code, notice how each branch point on the tree corresponds to a recursive call, and notice how the tree is drawn to the right all the way down to its shortest twig. Now, notice how the program works its way back up the trunk until the entire right side of the tree is drawn. Then the left side of the tree is drawn, but not by going as far out to the left as possible. Rather, once again the entire right side of the left tree is drawn until we finally make our way out to the smallest twig on the left.

## 4.7 Sierpinski Triangle

Another fractal that exhibits the property of self-similarity is the Sierpinski triangle. The Sierpinski triangle illustrates a three-way recursive algorithm. The procedure for drawing a Sierpinski triangle by hand is simple. Start with a single large triangle. Divide this large triangle into four new triangles by connecting the midpoint of each side. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles. You can continue to apply this procedure indefinitely if you have a sharp enough pencil.



Since we can continue to apply the algorithm indefinitely, what is the base case? We will see that the base case is set arbitrarily as the minimal perimeter (64 in this case). Sometimes we call this number the “degree” of the fractal. Each time we make a recursive call, we cut the

perimeter by one half until we reach a number below 64. When we reach a degree below 64, we stop making recursive calls.

```
var canvas = document.getElementById('c');
var arrowCanvas = document.getElementById('a');
var context = canvas.getContext("2d");
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
arrowCanvas.width = window.innerWidth;
arrowCanvas.height = window.innerHeight;
var turtle = new Turtle(canvas, arrowCanvas);

var size = window.innerWidth/2;
var min = 64;
var pf = 0.8660254;    // Pythagoras factor: Math.sqrt(3)/2

function T(l, x, y, myTurtle) {
  // Not done yet?
  if (l > min) {
    // Scale down by 2
    l = l/2;
    // Bottom left triangle
    T(l, x, y, myTurtle);
    // Bottom right triangle
    T(l, x+l, y, myTurtle);
    // Top triangle
    T(l, x+l/2, y-l*pf, myTurtle);
  }
  // Done recursing
  else {
    // Start at (x,y)
    myTurtle.goto(x, y);
    myTurtle.down();
    // Prepare to fill triangle
    myTurtle.beginFill();
    // Triangle base
    myTurtle.forward(l);
    myTurtle.left(120);
    // Triangle right
    myTurtle.forward(l);
    myTurtle.left(120);
    // Triangle left
    myTurtle.forward(l);
    myTurtle.endFill();
    // Face East
    myTurtle.setHeading(-90);
    // Finish at (x,y)
    myTurtle.up();
    myTurtle.goto(x, y);
  }
}
```



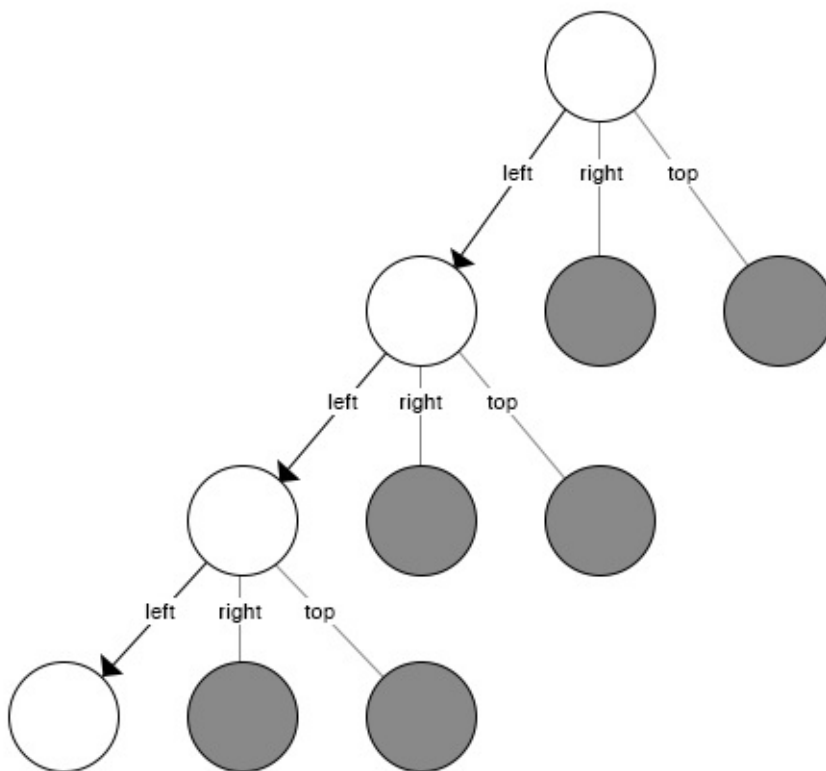
```

turtle.setHeading(-90);
T(size, size/2, window.innerWidth/2, turtle);
turtle.draw();

```

Note that the use of recursion allows the code to reflect the structure of the picture to be drawn. Let  $T$  be the initial triangle (which is invisible at first). When we enter the function  $T$ , we reset the scale by one half, because the internal triangles (that will be created next) have half the perimeter of the original one. Then we create the bottom left triangle with  $T(1, x, y)$ , then the bottom right, then the top one. Then we do that to each of those triangles too – bottom left, bottom right, top – until they're small enough, at which point we actually draw the individual triangles and fill them in with black.

Sometimes it is helpful to think of a recursive algorithm in terms of a diagram of function calls. The following diagram shows that the recursive calls are always made going to the bottom left. The active functions are outlined in black, and the inactive function calls are in gray. The farther you go toward the bottom of the diagram, the smaller the triangles. The function finishes drawing one level at a time; once it is finished with the bottom left it moves to the bottom middle, and so on.



## 4.8 Complex Recursive Problems

In the previous sections we looked at some problems that are relatively easy to solve and some graphically interesting problems that can help us gain a mental model of what is happening in a recursive algorithm. In this section we will look at some problems that are really difficult to solve using an iterative programming style but are very elegant and easy to solve using recursion. We will finish up by looking at a deceptive problem that at first looks like it has an elegant recursive solution but in fact does not.

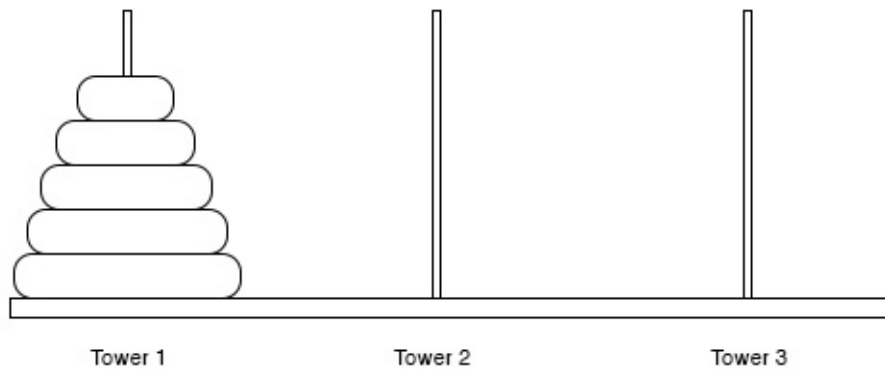
### 4.8.1 Tower of Hanoi

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. He was inspired by a legend that tells of a Hindu temple where the puzzle was presented to young priests. At the beginning of time, the priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it. Their assignment was to transfer all 64 disks from one of the three poles to another, with two important constraints. They could only move one disk at a time, and they could never place a larger disk on top of a smaller one. The priests worked very efficiently, day and night, moving one disk every second. When they finished their work, the legend said, the temple would crumble into dust and the world would vanish.

Although the legend is interesting, you need not worry about the world ending any time soon. The number of moves required to correctly move a tower of 64 disks is  $2^{64} - 1 = 18,446,744,073,709,551,615$ . At a rate of one move per second, that is 584,942,417,355 years! Clearly there is more to this puzzle than meets the eye.

Notice that, as the rules specify, the disks on each peg are stacked so that smaller disks are always on top of the larger disks. If you have not tried to solve this puzzle before, you should try it now. You do not need fancy disks and poles—a pile of books or pieces of paper will work. Here are the three rules to be followed:

- You can only move one disk at a time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.



How do we go about solving this problem recursively? How would you go about solving this problem at all? What is our base case? Let's think about this problem from the bottom up. Suppose you have a tower of five disks, originally on the first peg. Our base case will be to move a single disk to peg three.

Here is a high-level outline of how to move a tower from the starting pole, to the goal pole, using an intermediate pole:

1. Move a tower of height-1 to an intermediate pole, using the final pole.
2. Move the remaining disk to the final pole.
3. Move the tower of height-1 from the intermediate pole to the final pole using the original pole.

As long as we always obey the rule that the larger disks remain on the bottom of the stack, we can use the three steps above recursively, treating any larger disks as though they were not even there. The only thing missing from the outline above is the identification of a base case. The simplest Tower of Hanoi problem is a tower of one disk. In this case, we need move only a single disk to its final destination. A tower of one disk will be our base case. In addition, the steps outlined above move us toward the base case by reducing the height of the tower in steps 1 and 3.

```
function moveTower(height, fromPole, toPole, withPole) {
  if (height >= 1) {
    moveTower(height - 1, fromPole, withPole, toPole);
    moveDisk(fromPole, toPole);
    moveTower(height - 1, withPole, toPole, fromPole);
  }
}

function moveDisk(fp, tp) {
  console.log('moving disk from', fp, 'to', tp);
}

moveTower(3, 'A', 'B', 'C');

// moving disk from A to B
// moving disk from A to C
// moving disk from B to C
// moving disk from A to B
// moving disk from C to A
// moving disk from C to B
// moving disk from A to B
```

## 4.9 Dynamic Programming

In computer science, you may face a kind of problem called **optimization problem**. We can define it as a problem that require to select the best element with regard to some criterion from some set of available alternatives. For example, find the shortest path between two points. In this book, we will explore several different problem solving strategies for the kind of problems. **Dynamic programming** is one of this strategy.

Suppose you are programming a vending machine. You have to find a way to make change using minimum number of coins. If a customer puts in an euro bill and purchases an item for 37 cents, what is the smallest number of coins you can use to make change? The answer is 4 coins: one of 50 cents, one of 10, one of 2 and one cent pieces. To arrive to this result, start with the largest coin available (50 cents) and use as many of those as possible. Then we go to the next lowest coin value and use as many of those as possible. This approach is called a **greedy method** because we treats the solution as some sequence of steps and picks the locally optimal choice at each step. But a greedy algorithm does not guarantee an optimal solution. Picking locally optimal choices may result in a bad global solution.

What if our vending machines is used in Lower Elbonia where they have a 21 cent coin? In the previous exemple, with the greedy method we would still find the solution to be four coins. However, the optimal answer is three coins of 21 cent.

The dynamic programming (or dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space.

This kind of algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not guarantee an optimal solution, because picking locally optimal choices may result in a bad global solution, but it is often faster to calculate. Note that some greedy algorithms (such as Kruskal's or Prim's for minimum spanning trees) are proven to lead to the optimal solution.

```
function giveBackChange(coinValueList, change) {  
    var minCoins = change;  
    if (coinValueList.includes(change)) {  
        return 1;  
    }  
    else {  
        for (var i = 0; i < array.length; i++) {  
            }  
        }  
    }  
    return minCoins;  
}  
  
console.log( giveBackChange([1, 5, 10, 25], 63) );
```

# Arrays

Let's start with the most common data structure: The array. Even if you didn't study computer science in school, you should be familiar with it. Arrays are among the oldest and most important data structures. They are used by almost every program and built-in in every programming language, which makes them usually very efficient. As we will see later, they are also used to implement many other data structures, such as lists and strings. In this chapter we explore how it work in JavaScript and when to use them.

## Arrays in JavaScript

An array consist of a collection of elements (values or variables), each identified by an array index or key. In other words, its a linear list-like object.

If you are more familiar with Java, C++ or C#, JavaScript can seem to be a strange language. For example, every object is an associative array. In fact, the associative array is the fundamental data structure in JavaScript. This is so powerful it can be used to implement object or just about any other data structure. When JavaScript was first released it didn't have a dedicated Array data structure. Why? Because it didn't need one (and in many ways it still doesn't). But to make JavaScript easier to use from the perspective of another language it does have an Array object. Here is why looking at the Array object can be confusing. Try to not forget that every data structure in JavaScript is an object. Or more accurately, an associative array. Because they are objects, they are not as efficient as in other programming languages.

## Using Arrays

The array object prototype has methods to perform traversal and mutation operations.

## Create an array

The easiest and most efficient way to create an array is to assign the `[]` operator to a variable:

```
var fruits = [];
```

This gives you an Array instance inheriting from `Array.prototype`. This new array is empty, we can verify it by calling the length property:

```
console.log(fruits.length);  
> 0
```

But we can also create a non-empty array by declaring our variables inside the `[]` operator:

```
var fruits = ['banana', 'cherry', 'strawberry'];  
console.log(fruits.length);  
> 3
```

Note that in JavaScript, array elements can be of different type:

```
var myArray = ['whale', 42, null, false];
```

You can also use the Array constructor with the same logic:

```
var fruits = new Array();  
console.log(fruits.length);  
> 0  
  
var fruits = new Array('banana', 'cherry', 'strawberry');  
console.log(fruits.length);  
> 3
```

But I suggest you to not use this approach, especially since it doesn't work properly if there is a single value that is a number:

```
new Array(4);  
> [undefined × 4]
```

## ES6 - New methods to create an Array instance

### Array.from()

In ES6, you can now create a new Array instance from an array-like or iterable object by using the `Array.from()` method.

For example, to create an array from a string:



```
var fruits = Array.from("banana");
console.log(fruits);
> ["b", "a", "n", "a", "n", "a"]
```

## Array.of()

With this method, you can create an array of elements, regardless of their number or type.

```
var fruits = Array.of('banana', 'cherry', 'strawberry', 42);
console.log(fruits);
> ["banana", "cherry", "strawberry", 42]
```

## Operations on array elements

You can access to an array element using the `[ ]` operator by its index:

```
var fruits = ['banana', 'cherry', 'strawberry'];
console.log( fruits[1] );
> cherry
```

You can also access sequentially to all the elements by using the `for` loop:

```
var fruits = ['banana', 'cherry', 'strawberry'];
for (var i = 0; i < fruits.length; ++i) {
  console.log( fruits[i], i );
}
> banana 0
> cherry 1
> strawberry 2
```

You can use as well the `forEach` method:

```
var fruits = ['banana', 'cherry', 'strawberry'];
fruits.forEach(function (item, index, array) {
  console.log(item, index);
});
> banana 0
> cherry 1
> strawberry 2
```

An important difference between the `for` loop and the `forEach` method is that, with the former, you may break out of the loop, using the `break` statement.

```
var fruits = ['banana', 'cherry', 'strawberry'];
for (var i = 0; i < fruits.length; ++i) {
  console.log( fruits[i], i )
  if (fruits[i] === 'cherry' ) {
    // Stop the loop execution
    break;
  }
}
> banana 0
> cherry 1
```

*If you try to use a `break` within a `forEach` loop, you will get the following error `Uncaught SyntaxError: Illegal break statement` .*

Furthermore, the `forEach` method is much more expressive, about 95% slower, than the `for` statement. You can check it yourself right here: [www.jsperf.com/fast-array-foreach](http://www.jsperf.com/fast-array-foreach).