DATA STRUCTURES & ALGORITHMS IN JAVASCRIPT

by Adrian Mejia

Level up your software development career!

Data Structures & Algorithms in JavaScript

Adrian Mejia

Version 2.7.4, 2021-02-11

Data Structures & Algorithms in JavaScript

For online information and ordering this and other books, please visit https://adrianmejia.com. The publisher offers discounts on this publication when ordered in quantity. For more information, contact sales@adrianmejia.com.

Version 2.7.4, 2021-02-11.

Table of Contents

Dedication

To my wife Nathalie, who supported me in my long hours of writing, and my baby girl Abigail.

Preface

What is in this book?

Data Structures & Algorithms in JavaScript is a book that can be read from cover to cover. Each section builds on top of the previous one. Also, you can use it as a reference manual. Developers can refresh specific topics before an interview or look for ideas to solve a problem optimally. (Check out the Time Complexity Cheatsheet and topical index)

This publication is designed to be concise, intending to serve software developers looking to get a firm conceptual understanding of data structures in a quick yet in-depth fashion. After reading this book, the reader should have a fundamental knowledge of algorithms, including when and where to apply it, what are the trade-offs of using one data structure over the other. The reader will then be able to make intelligent decisions about algorithms and data structures in their projects.

Who this book is for

This book is for software developers familiar with JavaScript looking to improve their problem-solving skills or preparing for a job interview.

You can apply the concepts in this book to any programming language. However, instead of doing examples in pseudo-code, we are going to use JavaScript to implement the code examples.

What you need for this book

You will need Node.js. The book code was tested against Node.js v14.8, but newer versions should also work.

All the code examples used in this book can be found on: https://github.com/amejiarosario/dsa.js

Conventions

We use some typographical conventions within this book that distinguish between different kinds of information.

The code in the text, including commands, variables, file names, and property names are shown as follows:

> Repeat pair comparison until the last element that has been bubbled up to the right side `array.length - i`.

A block of code is set out as follows. It may be colored, depending on the format in which you're reading this book.

```
function* dummyIdMaker() {
  yield 0;
  yield 1;
  yield 2;
}

const generator = dummyIdMaker()

// getting values
console.log(generator.next()); // ↪ {value: 0, done: false}
```

When we want to draw your attention to specific code lines, those lines are annotated using numbers accompanied by brief descriptions.

Quicksort implementation in JavaScript (QuickSort)

```
/**
 * QuickSort - Efficient in-place recursive sorting algorithm.
 * Avg. Runtime: O(n log n) | Worst: O(n^2)
 * @param {Number[]} array
 * @param {Number} low
 * @param {Number} high
 */
function quickSort(array, low = 0, high = array.length - 1) {
  if (low < high) { ④
    const partitionIndex = partition(array, low, high); ①
    quickSort(array, low, partitionIndex - 1); ②
    quickSort(array, partitionIndex + 1, high); ③
  }
  return array;
}
```

① Partition: picks a pivot and find the index where the pivot will be when the array is sorted.

② Do the partition of the sub-array at the left of the pivot.

③ Do the partition of the sub-array at the right of the pivot.

④ Only do the partition when there's something to divide.

The following admonitions are used to highlight content.

Reword essential concepts. Useful for memorizing, tweeting, and sharing.

Tips are shown using callouts like this.

Warnings are shown using callouts like this.

This is a side note

Sidebar

Additional information about a certain topic may be displayed in a sidebar like this one.

Finally, this text shows what a quote looks like:

> Measurement is the first step that leads to control and eventually to improvement. If you can't measure something, you can't understand it. If you can't understand it, you can't control it. If you can't manage it, you can't improve it.
>
> — H. J. Harrington

Reader feedback

Your feedback is very welcome and valuable. Let us know your thoughts about this book — what you like or ideas to make it better.

To send us feedback, e-mail us at hello+dsajs@adrianmejia.com, send a tweet to @iAmAdrianMejia, or using the hash tag `#dsaJS`.

Introduction

You are about to become a better programmer and grasp the fundamentals of Algorithms and Data Structures. Let's take a moment to explain how we are going to do that.

This book is divided into four main parts:

In **Part 1**, we will cover the framework to compare and analyze algorithms: Big O notation. When you have multiple solutions to a problem, this framework comes in handy to know which solution will scale better.

In **Part 2**, we will go over linear data structures and trade-offs about using one over another. After reading this part, you will know how to trade space for speed using Maps, when to use a linked list over an array, or what problems can be solved using a stack over a queue.

Part 3 is about graphs and trees and its algorithms. Here you'll learn how to translate real-world problems into graphs and different algorithms to solve them.

Part 4 will cover tools and techniques to solve algorithmic problems. This section is for those who want to get better at recognizing patterns and improving problem-solving skills. We cover sorting algorithms and standard practices like dynamic programming, greedy algorithms, divide and conquer, and more.

Finally, in **Appendix A**, we summarize all the topics covered in this book in a cheatsheet. **Appendix B and C** covers self-balancing binary search tree algorithms. **Appendix D** cover the solutions to the problems presented at the end of each chapter.

Let's get started.

PART ONE

1. Algorithms Analysis

In this part, we are going to cover the basics of algorithms analysis. We will also discuss the most common runtimes of algorithms and provide a code example for each one.

1.1. Fundamentals of Algorithms Analysis

You are probably reading this book because you want to write better and faster code. How can you do that? Can you time how long it takes to run a program? Of course, you can! ⏱ However, if you run the same program on a computer, cellphone, or even a smartwatch, it will take different times.



Wouldn't it be great if we can compare algorithms regardless of the hardware where we run them? That's what **time complexity** is for! But, why stop with the running time? We could also compare the memory "used" by different algorithms, and we call that **space complexity**.

In this chapter, you will learn:

- What's the best way to measure your code's performance regardless of what hardware you use.
- Learn how to use Big O notation to compare algorithms.
- How to use algorithms analysis to improve your programs speed.

Before going deeper into space and time complexity, let's cover the basics real quick.

1.1.1. What are Algorithms?

Algorithms (as you might know) are steps of how to do some tasks. When you cook, you follow a recipe (or an algorithm) to prepare a dish. Let's say you want to make a pizza.

Example of an algorithm to make pizza

```
import { rollOut, applyToppings, Oven } from '../pizza-utils';

function makePizza(dough, toppings = ['cheese']) {
    const oven = new Oven(450);
    const rolledDough = rollOut(dough);
    const rawPizza = applyToppings(rolledDough, toppings);
    const pizzaPromise = oven.bake(rawPizza, { minutes: 20 });
    return pizzaPromise;
}
```

If you play a game, you'll devise strategies (or algorithms) to win. Likewise, algorithms in

computers are a set of instructions used to solve a problem.

Algorithms are the steps on how to perform a task.

1.1.2. Comparing Algorithms

Not all algorithms are created equal. There are “good” and “bad” algorithms. The good ones are fast; the bad ones are slow. Slow algorithms cost more money to run. Inefficient algorithms could make some calculations impossible in our lifespan!

Let’s say you want to compute the shortest path from Boston to San Francisco. Slow algorithms can take hours or crash before finishing. On the other hand, a "good" algorithm might compute in a few seconds.

Usually, algorithms time grows as the size of the input increases. For instance, calculating the shortest distance from your house to the local supermarket will take less time than other destination thousands of miles away.

Another example is sorting an array. A good sorting algorithm is Merge Sort, and an inefficient algorithm for large inputs is Selection Sort. Organizing 1 million elements with merge sort could take 20 seconds, for instance, while selection sort takes 12 days, ouch! The fantastic thing is that both programs solve the same problem with comparable data and hardware; yet, there’s a big difference in time! Bad algorithms would perform poorly, even on a supercomputer.

To give you a clearer picture of how different algorithms perform as the input size grows, look at the following problems and how their relative execution time changes as the input size increases.

Table 1. Relationship between algorithm input size and time taken to complete

Input size →	**10**	**100**	**10k**	**100k**	**1M**
Finding if a number is odd	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
Sorting array with merge sort	< 1 sec.	< 1 sec.	< 1 sec.	few sec.	20 sec.
Sorting array with Selection Sort	< 1 sec.	< 1 sec.	2 minutes	3 hours	12 days
Finding all subsets	< 1 sec.	40,170 trillion years	> centillion years	∞	∞
Finding string permutations	4 sec.	> vigintillion years	> centillion years	∞	∞

As you can see in the table, most algorithms on the table are affected by the input size. But not all and not at the same rate. Finding out if a number is odd will take the same if it is 1 or 1 million. We say then that the growth rate is constant. Others grow very fast. Finding all the permutations on a string of length 10 takes a few seconds, while if the string has a size of 100, it won’t even finish!

After completing this book, you are going to *think algorithmically*. You will be able to tell the growth rate of your programs and scale them. You’ll find bottlenecks of existing software and have an Algorithmic Toolbox.

1.1.3. Increasing your code performance

The first step to improve your code performance is to learn how to measure it. As somebody said:

> Measurement is the first step that leads to control and eventually to improvement. If you can't measure something, you can't understand it. If you can't understand it, you can't control it. If you can't control it, you can't improve it.
>
> — H. J. Harrington

This section will learn the basics of measuring our current code performance and compare it with other algorithms.

Calculating Time Complexity

In computer science, time complexity describes the number of operations a program will execute given the size of the input `n`.

How do you get a function that gives you the rough number of operations that the CPU will execute?

One idea is to analyze your code line by line and mind code inside loops. Let's do an example to explain this point. For instance, we have a function to find the minimum value on an array called `getMin`.

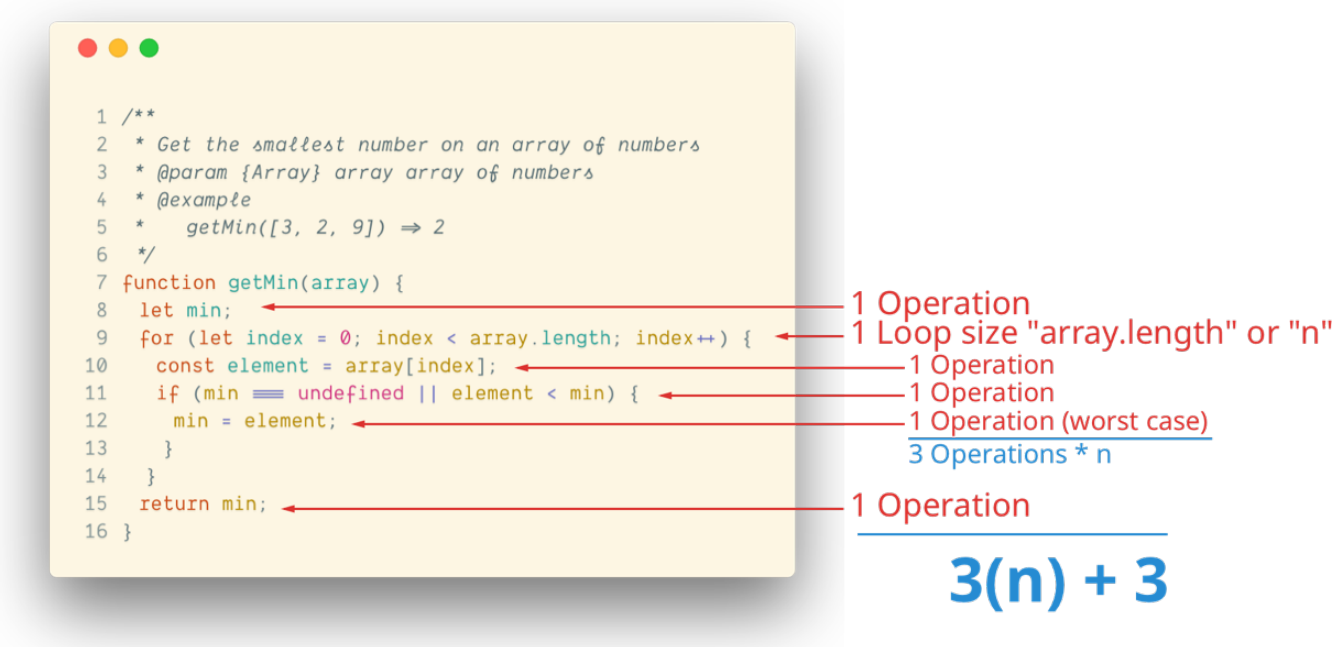


Figure 1. Translating lines of code to an approximate number of operations

Assuming that each line of code is an operation, we get the following:

3n + 3

`n` = input size.

That means that if you have an array of 3 elements, e.g. `getMin([3, 2, 9])`, then it will execute around *3(3)+3 = 12* operations. Of course, this is not for every case. For instance, Line 12 is only executed if the condition on line 11 is met. As you might learn in the next section, we want to get the big picture and get rid of smaller terms to compare algorithms easier.

1.1.4. Space Complexity

Space complexity is similar to time complexity. However, instead of the count of operations executed, it will account for the amount of memory used additionally to the input.

For calculating the **space complexity**, we keep track of the "variables" and memory used. In the `getMin` example, we create a variable called `min`, which only holds one value at a time. So, the space complexity is `1`. On other algorithms, If we have to use an auxiliary array that holds the same number of elements as the input, then the space complexity would be `n`.

Simplifying Complexity with Asymptotic Analysis

When we compare algorithms, we about the growth rate when the input gets huge (towards infinity). Then you have a function like `20*n^3 + 100`. If `n` is one million. The term `+ 100` makes a tiny contribution to the result (less than 0.000001%). Here is when the asymptotic analysis comes to the rescue.

Asymptotic analysis describes the behavior of functions as their inputs approach to infinity.

In the previous example, we analyzed `getMin` with an array of size 3; what happens if the size is 10, 10k, or 10 million?

Table 2. Operations performed by an algorithm with a time complexity of `3n + 3`

n (size)	Operations	total
10	3(10) + 3	33
10k	3(10k)+3	30,003
1M	3(1M)+3	3,000,003

As the input size `n` grows bigger and bigger, then the expression *3n + 3* is closer and closer to *3n*. Dropping terms might look like a stretch at first, but you will see that what matters the most is the higher-order terms of the function rather than lesser terms and constants.

What is Big O Notation?

There's a notation called **Big O**, where `O` refers to the **order of a function** in the worst-case scenario.

Big O = Big Order (rate of growth) of a function.

If you have a program that has a runtime of:

$7n^3 + 3n^2 + 5$

You can express it in Big O notation as $O(n^3)$. The other terms ($3n^2 + 5$) will become less significant as the input grows bigger.

Big O notation only cares about the "biggest" terms in the time/space complexity. It combines what we learn about time and space complexity, asymptotic analysis and adds a worst-case scenario.

All algorithms have three scenarios:

- Best-case scenario: the most favorable input arrangement where the program will take the least amount of operations to complete. E.g., a sorted array is beneficial for some sorting algorithms.
- Average-case scenario: this is the most common case. E.g., array items in random order for a sorting algorithm.
- Worst-case scenario: the inputs are arranged in such a way that causes the program to take the longest to complete. E.g., array items in reversed order for some sorting algorithm will take the longest to run.

To sum up:

Big O only cares about the run time function's highest order in the worst-case scenario.

Don't drop terms that are multiplying other terms. $O(n \log n)$ is not equivalent to $O(n)$. However, $O(n + \log n)$ is.

There are many common notations like polynomial, $O(n^2)$ as we saw in the `getMin` example, constant $O(1)$, and many more that we are going to explore in the next chapter.

Again,

the time complexity is not a direct measure of how long a program takes to execute, but rather how many operations it performs given the input size.

Nevertheless, there's a relationship between time complexity and clock time, as shown in the following table.

Table 3. How long an algorithm takes to run based on their time complexity and input size

Input Size	O(1)	O(n)	O(n log n)	$O(n^2)$	$O(2^n)$	O(n!)
1	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
10	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	4 seconds
10k	< 1 sec.	< 1 sec.	< 1 sec.	2 minutes	∞	∞
100k	< 1 sec.	< 1 sec.	1 second	3 hours	∞	∞
1M	< 1 sec.	1 second	20 seconds	12 days	∞	∞

This is just an illustration since, in different hardware, the times will be distinct. These times are under the assumption of running on 1 GHz CPU, and it can execute on average one instruction in 1 nanosecond (usually takes more time). Also, keep in mind that each line might be translated into dozens of CPU instructions depending on the programming language.

1.1.5. Summary

In this chapter, we learned how you could measure your algorithm performance using time complexity. Rather than timing how long your program takes to run, you can approximate the number of operations it will perform based on the input size.

We learned about time and space complexity and how they can be translated to Big O notation. Big O refers to the **order** of the function.

In the next section, we will go deeper into how to analyze time complexity and provide examples!

1.2. How to determine time complexity from code?

In general, you can determine the time complexity by analyzing the program's statements. However, you have to be mindful of how are the statements arranged. Suppose they are inside a loop or have function calls or even recursion. All these factors affect the runtime of your code. Let's see how to deal with these cases.

Sequential Statements

If we have statements with basic operations like conditionals, assignments, reading a variable. We can assume they take constant time.

```
statement 1;
statement 2;
...
statement n;
```

If we calculate the total time complexity, it would be something like this:

```
total = time(statement 1) + time(statement 2) + ... time (statement n)
```

Let's use `T(n)` as the total time in function of the input size `n`, and `t` as the time complexity taken by a statement or group of statements.

```
T(n) = t(statement 1) + t(statement 2) + ... + t(statement n);
```

If each statement executes a basic operation, we can say it takes constant time `O(1)`. As long as you have a fixed number of operations, it will be constant time, even if we have 1 or 100 of these statements.

be careful with function calls. You will have to go to the implementation and check their run time. More on that later.

Conditional Statements

Very rarely, you have a code without any conditional statement. How do you calculate the time complexity? Remember that we care about the worst-case with Big O so that we will take the maximum possible runtime.

```
if (isValid) {
  statement 1;
  statement 2;
} else {
  statement 3;
}
```

Since, we are after the worst-case we take the whichever is larger of the two possibilities:

```
T(n) = Math.max([t(statement 1) + t(statement 2)], [time(statement 3)])
```

Loop Statements

Another prevalent scenario is loops like for-loops or while-loops. For any loop, we find out the runtime of the block inside them and multiply it by the number of times the program will repeat the loop.

```
for (let i = 0; i < array.length; i++) {
  statement 1;
  statement 2;
}
```

For this example, the loop is executed `array.length`, assuming `n` is length of the array we get the following:

```
T(n) = n * [ t(statement 1) + t(statement 2) ]
```

All loops that grow proportionally to the input size have a linear time complexity `O(n)`. If you loop through only half of the array, that's still `O(n)`. Remember that we drop the constants so `1/2 n => O(n)`.

However, if a constant number bounds the loop, let's say 4 (or even 400). Then, the runtime is constant `O(4) -> O(1)`. See the following example.

```
for (let i = 0; i < 4; i++) {
  statement 1;
  statement 2;
}
```

That code is `O(1)` because it no longer depends on the input size.

Nested loops statements

Sometimes you might need to visit all the elements on a 2D array (grid/table). For such cases, you might find yourself with two nested loops.

```
for (let i = 0; i < n; i++) {
  statement 1;

  for (let j = 0; j < m; j++) {
    statement 2;
    statement 3;
  }
}
```

For this case you would have something like this:

```
T(n) = n * [t(statement 1) + m * t(statement 2...3)]
```

Assuming the statements from 1 to 3 are `O(1)`, we would have a runtime of `O(n * m)`. If instead of `m`, you had to iterate on `n` again, then it would be `O(n^2)`. Another typical case is having a function inside a loop. Let's see how to deal with that next.

Function call statements

When you calculate your programs' time complexity and invoke a function, you need to be aware of its runtime. If you created the function, that might be a simple inspection of the implementation. However, you might infer it from the language/library documentation if you use a 3rd party function.

Let's say you have the following program:

```
for (let i = 0; i < n; i++) {
  fn1();
  for (let j = 0; j < n; j++) {
    fn2();
    for (let k = 0; k < n; k++) {
      fn3();
    }
  }
}
```

Depending on the runtime of fn1, fn2, and fn3, you would have different runtimes.

- If they all are constant `O(1)`, then the final runtime would be `O(n^3)`.
- However, if only `fn1` and `fn2` are constant and `fn3` has a runtime of `O(n^2)`, this program will have a runtime of `O(n^5)`. Another way to look at it is, if `fn3` has two nested and you replace

the invocation with the actual implementation, you would have five nested loops.

In general, you will have something like this:

```
T(n) = n * [ t(fn1()) + n * [ t(fn2()) + n * [ t(fn3()) ] ] ]
```

Recursive Functions Statements

Analyzing the runtime of recursive functions might get a little tricky. There are different ways to do it. One intuitive way is to explore the recursion tree.

Let's say that we have the following program:

```
function fn(n) {
  if (n < 0) return 0;
  if (n < 2) return n;

  return fn(n - 1) + fn(n - 2);
}
```

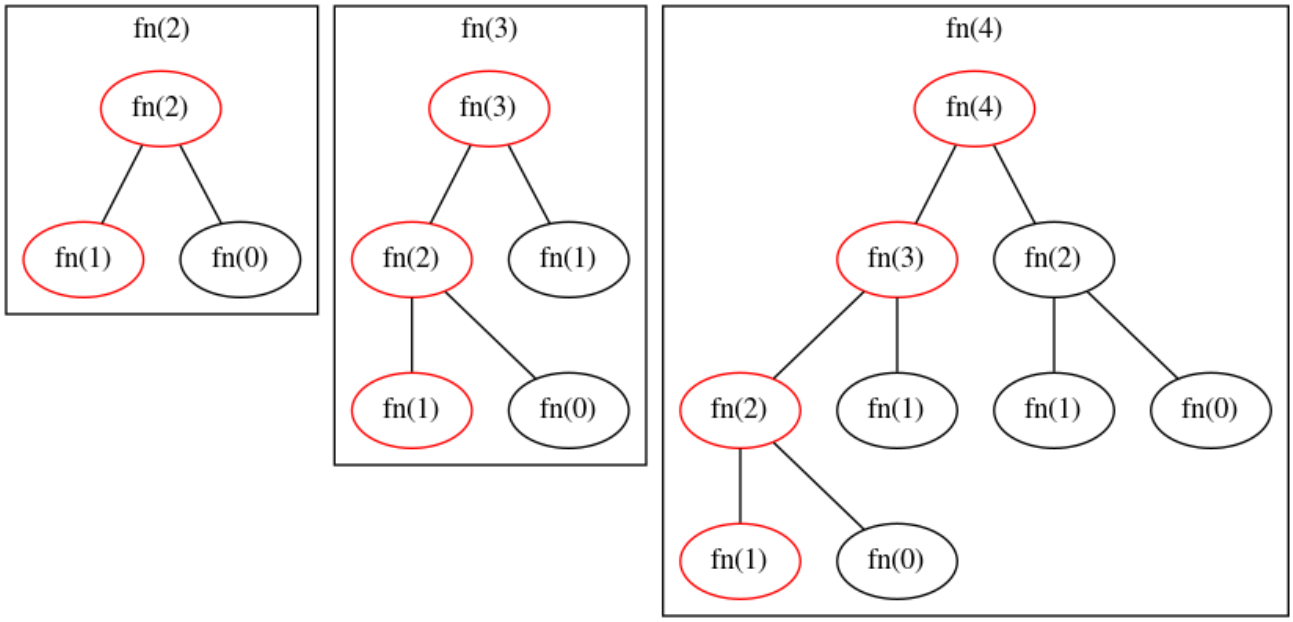
You can represent each function invocation as a bubble (or node).

Let's do some examples:

- When you n = 2, you have 3 function calls. First `fn(2)` which in turn calls `fn(1)` and `fn(0)`.
- For `n = 3`, you have 5 function calls. First `fn(3)`, which in turn calls `fn(2)` and `fn(1)` and so on.
- For `n = 4`, you have 9 function calls. First `fn(4)`, which in turn calls `fn(3)` and `fn(2)` and so on.

Since it's a binary tree, we can sense that every time `n` increases by one, we would have to perform at most the double of operations.

Here's the graphical representation of the 3 examples:



If you take a look at the generated tree calls, the leftmost nodes go down in descending order: `fn(4)`, `fn(3)`, `fn(2)`, `fn(1)`, which means that the height of the tree (or the number of levels) on the tree will be `n`.

The total number of calls in a complete binary tree is `2^n - 1`. As you can see in `fn(4)`, the tree is not complete. The last level will only have two nodes, `fn(1)` and `fn(0)`, while a full tree would have eight nodes. But still, we can say the runtime would be exponential `O(2^n)`. It won't get any worst because `2^n` is the upper bound.

1.2.1. Summary

In this chapter, we learned how to calculate the time complexity of our code when we have the following elements:

- Basic operations like assignments, bit, and math operators.
- Loops and nested loops
- Function invocations and recursions.

In the next section, we are going to the most common time complexities and real code examples.

1.3. Big O examples

There are many kinds of algorithms. Most of them fall into one of the eight-time complexities that we will explore in this chapter.

Eight Running Time Complexities You Should Know

- Constant time: *O(1)*
- Logarithmic time: *O(log n)*
- Linear time: *O(n)*
- Linearithmic time: *O(n log n)*
- Quadratic time: $O(n^2)$
- Cubic time: $O(n^3)$
- Exponential time: $O(2^n)$
- Factorial time: *O(n!)*

We are going to provide examples for each one of them.

Before we dive in, here's a plot with all of them.

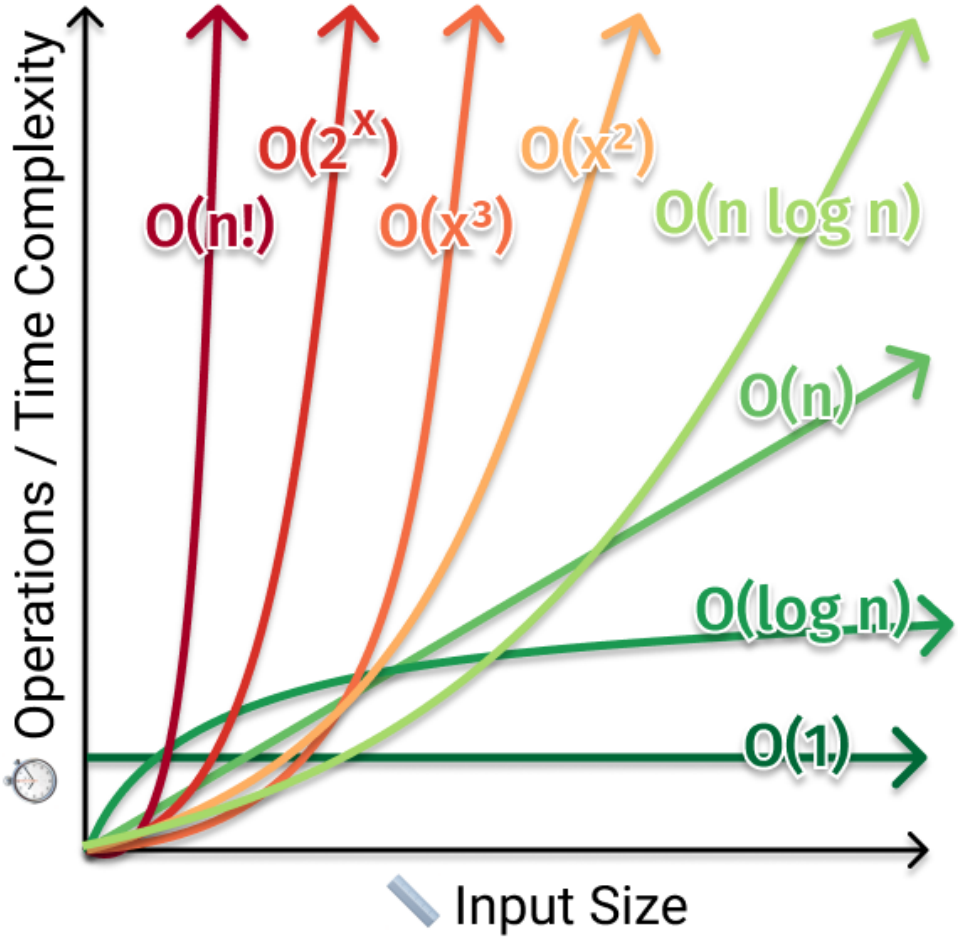


Figure 2. CPU operations vs. Algorithm runtime as the input size grows

The above chart shows how the algorithm's running time is related to the CPU's work. As you can see, O(1) and O(log n) is very scalable. However, $O(n^2)$ and worst can convert your CPU into a furnace 🔥 for massive inputs.

1.3.1. Constant

Represented as **O(1)**, it means that regardless of the input size, the number of operations executed is always the same. Let's see an example:

Finding if an array is empty

Let's implement a function that finds out if an array is empty or not.

```
/**
 * Return true if an array is empty and false otherwise
 * @param {array|string} thing
 * @example
 *      isEmpty() // => true
 *      isEmpty([]) // => true
 *      isEmpty('') // => true
 *      isEmpty([8, 9, 6]) // => false
 *      isEmpty('text') // => false
 */
function isEmpty(thing) {
  return !thing || thing.length < 1;
}
```

Another more real life example is adding an element to the begining of a Linked List. You can check out the implementation here.

As you can see in both examples (array and linked list), if the input is a collection of 10 elements or 10M, it would take the same amount of time to execute. You can't get any more performant than this!

1.3.2. Logarithmic

Represented in Big O notation as **O(log n)**, when an algorithm has this running time, it means that as the input size grows, the number of operations grows very slowly. Logarithmic algorithms are very scalable. One example is the **binary search**.

Searching on a sorted array

The binary search only works for sorted lists. It starts searching for an element in the middle of the array, and then it moves to the right or left depending on if the value you are looking for is bigger or smaller.

```
/**
 * Recursive Binary Search
 * Runtime: O(log n)
 *
 * @example
 *  binarySearch([1, 2, 3], 2); //↪ 1
 *  binarySearch([1, 2, 3], 31); //↪ -1
 * @param {array} array collection of sorted elements
 * @param {string|number} search value to search for
 * @param {number} offset keep track of array's original index
 * @return index of the found element or -1 if not found
 */
function binarySearchRecursive(array, search, offset = 0) {
  // split array in half
  const half = parseInt(array.length / 2, 10);
  const current = array[half];

  if (current === search) {
    return offset + half;
  } if (array.length === 1) {
    return -1;
  } if (search > current) {
    const right = array.slice(half);
    return binarySearchRecursive(right, search, offset + half);
  }

  const left = array.slice(0, half);
  return binarySearchRecursive(left, search, offset);
}
```

This binary search implementation is a recursive algorithm, which means that the function `binarySearchRecursive` calls itself multiple times until the program finds a solution. The binary search splits the array in half every time.

Finding the runtime of recursive algorithms is not very obvious sometimes. It requires some approaches like recursion trees or the Master Theorem.

Since the `binarySearch` divides the input in half each time. As a rule of thumb, when you have an algorithm that divides the data in half on each call, you are most likely in front of a logarithmic runtime: *O(log n)*.

1.3.3. Linear

Linear algorithms are one of the most common runtimes. Their Big O representation is **O(n)**. Usually, an algorithm has a linear running time when it visits every input element a fixed number of times.

Finding duplicates in an array using a map

Let's say that we want to find duplicate elements in an array. What's the first implementation that comes to mind? Check out this implementation:

```
/**
 * Finds out if an array has duplicates
 * Runtime: O(n)
 * @example
 *    hasDuplicates([]); //↪ false
 *    hasDuplicates([1, 1]); //↪ true
 *    hasDuplicates([1, 2]); //↪ false
 * @param {Array} array
 * @returns {boolean} true if has duplicates, false otherwise
 */
function hasDuplicates(array) {
  const words = new Map();
  for (let index = 0; index < array.length; index++) {
    const word = array[index];
    if (words.has(word)) {
      return true;
    }
    words.set(word, true);
  }
  return false;
}
```

`hasDuplicates` *has multiple scenarios:*

- **Best-case scenario**: the first two elements are duplicates. It only has to visit two elements and return.
- **Worst-case scenario**: no duplicates or duplicates are the last two. In either case, it has to visit every item in the array.
- **Average-case scenario**: duplicates are somewhere in the middle of the collection.

As we learned before, the big O cares about the worst-case scenario, where we would have to visit every element on the array. So, we have an **O(n)** runtime.

Space complexity is also **O(n)** since we are using an auxiliary data structure. We have a map that, in the worst case (no duplicates), it will hold every word.

1.3.4. Linearithmic

You can represent linearithmic algorithms as *O(n log n)*. This one is important because it is the best runtime for sorting! Let's see the merge-sort.

Sorting elements in an array

The Merge Sort, like its name indicates, has two functions merge and sort. Let's start with the sort function:

Sort part of the mergeSort

```
/**
 * Split array in half recursively until two or less elements are left.
 * Sort these two elements and combine them back using the merge
function.
 * @param {Array} array
 * @example
 *    splitSort([3, 2, 1]) => [1, 2, 3]
 *    splitSort([3]) => [3]
 *    splitSort([3, 2]) => [2, 3]
 */
function splitSort(array) {
  const size = array.length;
  // base case
  if (size < 2) {
    return array;
  } if (size === 2) {
    return array[0] < array[1] ? array : [array[1], array[0]]; ①
  }

  // recursive split in half and merge back
  const half = Math.ceil(size / 2);
  return merge( ③
    splitSort(array.slice(0, half)), ②
    splitSort(array.slice(half)), ②
  );
}
```

① If the array only has two elements, we can sort them manually.

② We divide the array into two halves.

③ Merge the two parts recursively with the `merge` function explained below

Merge part of the mergeSort

```
/**
 * Merge two arrays in ascending order
 *
 * @param {array} array1 sorted array 1
 * @param {array} array2 sorted array 2
 * @returns {array} merged arrays in asc order
 *
 * @example
 *    merge([2,5,9], [1,6,7]) => [1, 2, 5, 6, 7, 9]
 */
function merge(array1, array2 = []) {
  const mergedLength = array1.length + array2.length;
  const mergedArray = Array(mergedLength);

  // merge elements on a and b in asc order. Run-time O(a + b)
  for (let index = 0, i1 = 0, i2 = 0;
    index < mergedLength; index++) { ①
    if (i2 >= array2.length
      || (i1 < array1.length && array1[i1] <= array2[i2])) {
      mergedArray[index] = array1[i1]; ②
      i1 += 1;
    } else {
      mergedArray[index] = array2[i2]; ②
      i2 += 1;
    }
  }

  return mergedArray; ③
}
```

The merge function combines two sorted arrays in ascending order. Let's say that we want to sort the array `[9, 2, 5, 1, 7, 6]`. In the following illustration, you can see what each function does.

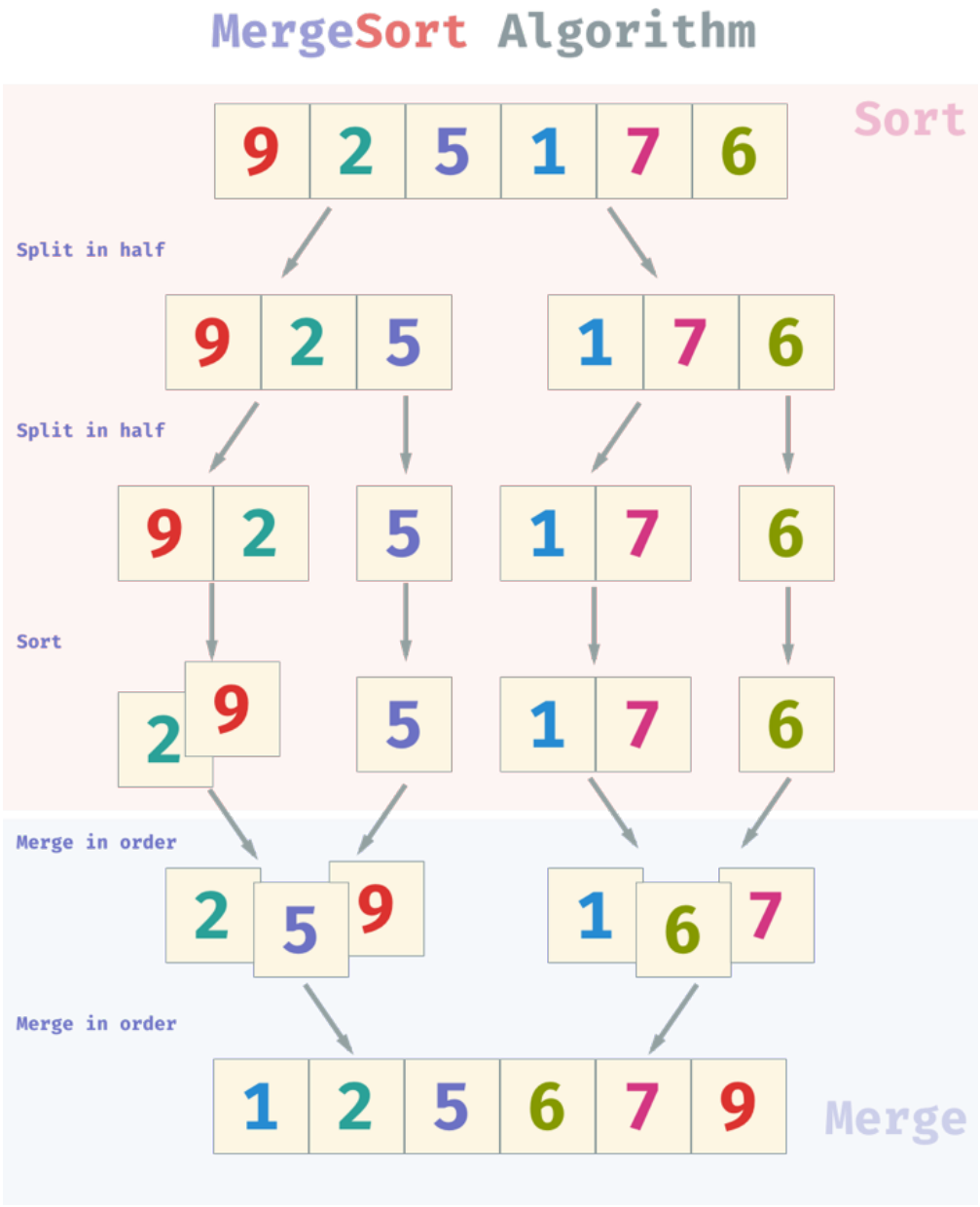


Figure 3. Mergesort visualization. Shows the split, sort and merge steps

How do we obtain the running time of the merge sort algorithm? The merge-sort divides the array in half each time in the split phase, *log n*, and the merge function join each splits, *n*. The total work is **O(n log n)**. There are more formal ways to reach this runtime, like using the Master Method and recursion trees.

1.3.5. Quadratic

Quadratic running times, $O(n^2)$, are the ones to watch out for. They usually don't scale well when they have a large amount of data to process.

Usually, they have double-nested loops, where each one visits all or most elements in the input. One example of this is a naïve implementation to find duplicate words on an array.

Finding duplicates in an array (naïve approach)

If you remember, we have solved this problem more efficiently in the Linear section. We solved this

problem before using an *O(n)*, let's solve it this time with an $O(n^2)$:

Naïve implementation of hasDuplicates function

```
/**
 * Finds out if an array has duplicates
 * Runtime: O(n^2)
 * @example
 *    hasDuplicates([]); //↪ false
 *    hasDuplicates([1, 1]); //↪ true
 *    hasDuplicates([1, 2]); //↪ false
 * @param {Array} array
 * @returns {boolean} true if has duplicates, false otherwise
 */
function hasDuplicates(array) {
  for (let outter = 0; outter < array.length; outter++) {
    for (let inner = outter + 1; inner < array.length; inner++) {
      if (array[outter] === array[inner]) {
        return true;
      }
    }
  }

  return false;
}
```

As you can see, we have two nested loops causing the running time to be quadratic. How much difference is there between a linear vs. quadratic algorithm?

Let's say you want to find a duplicated middle name in a phone directory book of a city of ~1 million people. If you use this quadratic solution, you would have to wait for ~12 days to get an answer 🐢; while if you use the linear solution, you will get the answer in seconds! 🚀

1.3.6. Cubic

Cubic $\mathbf{O(n^3)}$ and higher polynomial functions usually involve many nested loops. An example of a cubic algorithm is a multi-variable equation solver (using brute force) or finding three elements on an array that add up to a given number.

3 Sum

Let's say you want to find 3 items in an array that add up to a target number. One brute force solution would be to visit every possible combination of 3 elements and add them to see if they are equal to the target.

```
function threeSum(nums, target = 0) {
  const ans = [];

  for(let i = 0; i < nums.length -2; i++)
    for(let j = i + 1; j < nums.length - 1; j++)
      for(let k = j + 1; k < nums.length; k++)
        if (nums[i] + nums[j] + nums[k] === target)
          ans.push([nums[i], nums[j], nums[k]]);

  return ans;
}
```

As you can see, three nested loops usually translate to $O(n^3)$. If we had four nested loops (4sum), it would be $O(n^4)$ and so on. A runtime in the form of $O(n^c)$, where $c > 1$, we refer to this as a **polynomial runtime**.

You can improve the runtime of 3sum from $O(n^3)$ to $O(n^2)$, if we sort items first and then use one loop and two pointers to find the solutions.

1.3.7. Exponential

Exponential runtimes, $O(2^n)$, means that every time the input grows by one, the number of operations doubles. Exponential programs are only usable for a tiny number of elements (<100); otherwise, it might not finish in your lifetime. 💀

Let's do an example.

Finding subsets of a set

Finding all distinct subsets of a given set can be implemented as follows:

Subsets in a Set

```
/**
 * Finds all distinct subsets of a given set
 * Runtime: O(2^n)
 *
 * @example
 *    findSubsets('a') //↪ ['', 'a']
 *    findSubsets([1, 'b']) //↪ ['', '1', 'b', '1b']
 *    findSubsets('abc') //↪ ['', 'a', 'b', 'ab', 'c', 'ac', 'bc',
'abc']
 *
 * @param {string|array} n
 * @returns {array} all the subsets (including empty and set itself).
 */
function findSubsets(n = '') {
  const array = Array.from(n);
  const base = ['']; ①

  const results = array.reduce((previous, element) => {
    const previousPlusElement = previous.map((el) => `${el}${element}`);
②
    return previous.concat(previousPlusElement); ③
  }, base);

  return results;
}
```

① Base case is empty element.

② For each element from the input, append it to the results array.

③ The new results array will be what it was before + the duplicated with the appended element.

Every time the input grows by one, the resulting array doubles. That's why it has an **O(2^n)**.

1.3.8. Factorial

The factorial runtime, `O(n!)`, is not scalable at all. Even with input sizes of ~10 elements, it will take a couple of seconds to compute. It's that slow! 🐌

Factorial

A factorial is the multiplication of all the numbers less than itself down to 1.

For instance:

- 3! = 3 x 2 x 1 = 6
- 5! = 5 x 4 x 3 x 2 x 1 = 120
- 10! = 3,628,800
- 11! = 39,916,800

Getting all permutations of a word

One classic example of an *O(n!)* algorithm is finding all the different words formed with a given set of letters.

Word's permutations

```
/**
 * Find all the different permutations a word can have
 * Runtime: O(n!)
 * @example
 *    getPermutations('a') => ['a']
 *    getPermutations('ab') => ['ab', 'ba']
 *    getPermutations('mad') => ['mad', 'mda', 'amd', 'adm', 'dma',
'dam']
 * @param {string} word string or array of chars to find permutations
 * @param {string} prefix used internally for recursion
 * @returns {array} collection of all the ways the letters can be
arranged
 */
function getPermutations(word = '', prefix = '') {
  if (word.length <= 1) {
    return [prefix + word];
  }
  return Array.from(word).reduce((result, char, index) => {
    const reminder = word.slice(0, index) + word.slice(index + 1);
    return result.concat(getPermutations(reminder, prefix + char));
  }, []);
}
```

As you can see in the `getPermutations` function, the resulting array is the factorial of the word length.

Factorial starts very slow and quickly becomes unmanageable. A word size of just 11 characters

would take a couple of hours in most computers! ☃

1.3.9. Summary

We went through 8 of the most common time complexities and provided examples for each of them. Hopefully, this will give you a toolbox to analyze algorithms.

Table 4. Most common algorithmic running times and their examples

Big O Notation	Name	example (s)
O(1)	Constant	Finding if an array is empty
O(log n)	Logarithmic	Searching on a sorted array
O(n)	Linear	Finding duplicates in an array using a map
O(n log n)	Linearithmic	Sorting elements in an array
$O(n^2)$	Quadratic	Finding duplicates in an array (naïve approach)
$O(n^3)$	Cubic	3 Sum
$O(2^n)$	Exponential	Finding subsets of a set
O(n!)	Factorial	Getting all permutations of a word

PART TWO

2. Linear Data Structures

Data Structures come in many flavors. There's no one to rule them all. You have to know the tradeoffs so you can choose the right one for the job.

In your day-to-day work, you might not need to re-implement basic data structures. However, knowing how they work internally can help you understand their time complexity better (Remember the chapter How to determine Big-O from code?).

When you are aware of the data structures implementations, you spot when to use one over the other or even extend them to create a new one. We are going to explore the most common data structures' time and space complexity.

In this part we are going to learn about the following linear data structures:

- Array
- Map
- Set
- Linked List
- Stack
- Queue

Later, in the next part, we are going to explore non-linear data structures like Graphs and Trees.

2.1. Array

Arrays are one of the most used data structures. You probably have used it a lot already. But, are you aware of the runtimes of `push`, `splice`, `shift`, `indexOf`, and other operations? In this chapter, we are going deeper into the most common operations and their runtimes.

2.1.1. Array Basics

An array is a collection of things (strings, characters, numbers, objects, etc.). They can be many or zero.

Strings are a collection of characters. Most of the array methods apply to strings as well.

Arrays look like this:

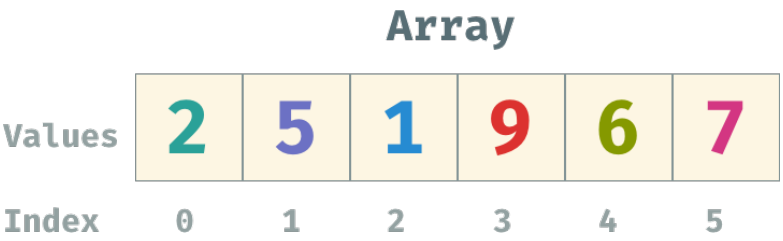


Figure 4. Array representation: You can access each value in constant time through its index.

Read and Update

Arrays are a contiguous collection of elements that can be accessed randomly using an index. This access by index operation takes `O(1)` time. Let's take a look at the different functions that we can do with arrays.

Reading elements from an array and string

```
const array = [2, 5, 1, 9, 6, 7];
const string = "hello";
console.log(array[2]); // 1
console.log(string[1]); // "e"
```

As you can see, you can access the string's characters using the same operator as arrays.

You can update arrays in the same way, using the `[]` operator. However, you can't modify strings. They are immutable!

Reading elements from an array and string

```
const array = [2, 5, 1, 9, 6, 7];
const string = "hello";
array[2] = 117;
console.log(array[2]); // 117
string[1] = "z"; // doesn't change the string.
console.log(string[1]); // "e"
```

When you try to modify and string, you won't get an error or anything. It just gets ignored! Your only option is to create a new string with the adjusted value.

Insertion

Insertions on an array have different times complexities. O(1): constant time (on average) to append a value at the end of the array. O(n): linear time to insert a value at the beginning or middle.

Inserting at the beginning of the array

What if you want to insert a new element at the beginning of the array? You would have to push every item to the right. We can use the following method:

Syntax

```
const newArrLength = arr.unshift(element1[, ...[, elementN]]);
```

Here's an example:

Insert to head

```
const array = [2, 5, 1];
array.unshift(0); // ↪ 8
console.log(array); // [ 0, 2, 5, 1 ]
array.unshift(-2, -1); // ↪ 6
console.log(array); // [ -2, -1, 0, 2, 5, 1 ]
```

As you can see, `2` was at index 0, now was pushed to index 1, and everything else is on a different index. `unshift` takes **O(n)** since it affects **all** the elements of the array.

JavaScript built-in `array.unshift`

The `unshift()` method adds one or more elements to the beginning of an array and returns its new length.

Runtime: `O(n)`.

Inserting at the middle of the array

Inserting a new element in the middle involves moving part of the array but not all of the items. We can use `splice` for that:

Syntax

```
const arrDeletedItems = arr.splice(start[, deleteCount[, item1[, item2[,
...]]]]);
```

Based on the parameters it takes, you can see that we can add and delete items. Here's an example of inserting in the middle.

Inserting element in the middle

```
const array = [2, 5, 1, 9, 6, 7];
array.splice(1, 0, 111); // ↪ [] ①
// array: [2, 111, 5, 1, 9, 6, 7]
```

① at position `1`, delete `0` elements and insert `111`.

The Big O for this operation would be **O(n)** since, in the worst case, it would move most of the elements to the right.

JavaScript built-in `array.splice`

The `splice()` method changes an array's contents by removing existing elements or adding new items. Splice returns an array containing the deleted items.

Runtime: O(n).

Inserting at the end of the array

For inserting items at the end of the array, we can use: push.

Syntax

```
const newArrLength = arr.push([element1[, ...[, elementN]]]);
```

We can push new values to the end of the array like this:

Insert to tail

```
const array = [2, 5, 1, 9, 6, 7];
array.push(4); // ↪ 7 ①
// array: [2, 5, 1, 9, 6, 7, 4]
```

① The `4` element would be pushed to the end of the array. Notice that `push` returns the new length of the array.

Adding to the tail of the array doesn't change other indexes. E.g., element 2 is still at index 0. So, this is a constant time operation **O(1)**.

JavaScript built-in `array.push`

The `push()` method adds one or more elements to the end of an array and returns its new length.

Runtime: O(1).

Searching by value and index

As we saw before, searching by the index is very easy using the `[]` operator:

Search by index

```
const array = [2, 5, 1, 9, 6, 7];
array[4]; // ↪ 6
```

Searching by index takes constant time - **O(1)** - to retrieve values out of the array.

Searching by value can be done using `indexOf`.

Syntax

```
const index = arr.indexOf(searchElement[, fromIndex]);
```

If the value is there, we will get the index, otherwise `-1`.

Search by value

```
const array = [2, 5, 1, 9, 6, 7];
console.log(array.indexOf(9)); // ↪ 3
console.log(array.indexOf(90)); // ↪ -1
```

Internally, `indexOf` has to loop through the whole array (worst case) or until we find the first occurrence. Time complexity is **O(n)**.

Deletion

There are three possible deletion scenarios (similar to insertion): removing at the beginning, middle, or end.

Deleting element from the beginning

Deleting from the beginning can be done using the `splice` function and the `shift`. For simplicity, we will use the latter.

Syntax

```
const removedElement = arr.shift();
let arrDeletedItems = arr.splice(start[, deleteCount[, item1[, item2[,
...]]]]);
```

Deleting from the beginning of the array.

```
const array = [2, 111, 5, 1, 9, 6, 7];
// Deleting from the beginning of the array.
array.shift(); // ↪ 2
array.shift(); // ↪ 111
console.log(array); // [5, 1, 9, 6, 7]
array.splice(0, 1); // ↪ [ 5 ]
console.log(array); // [ 1, 9, 6, 7 ]
```

As expected, this will change every index on the array, so this takes linear time: **O(n)**.

JavaScript built-in array.shift

The `shift()` method shift all elements to the left. In turn, it removes the first element from an array and returns that removed element. This method changes the length of the array.

Runtime: O(n).

Deleting element from the middle

We can use the `splice` method for deleting an item from the middle of an array.

You can delete multiple items at once:

Deleting from the middle

```
const array = [0, 1, 2, 3, 4];
// Deleting from the middle
array.splice(2, 3); // ↪ [ 2, 3, 4 ] ①
console.log(array); // [0, 1]
```

① delete 3 elements starting on position 2

Deleting from the middle might cause most of the array elements to move up one position to fill in

for the eliminated item. Thus, runtime: O(n).

Deleting element from the end

Removing the last element is very straightforward using pop:

Syntax

```
const removedItem = arr.pop();
```

Here's an example:

Deleting last element from the array

```
const array = [2, 5, 1, 9, 111];
array.pop();  // ↪111
// array: [2, 5, 1, 9]
```

While deleting the last element, no other item was touched, so that's an *O(1)* runtime.

JavaScript built-in `array.pop`

The `pop()` method removes the last element from an array and returns that element. This method changes the length of the array.

Runtime: O(1).

Array Complexity

To sum up, the time complexity of an array is:

Table 5. Time/Space complexity for the array operations

Data Structure	Searching By		Inserting at the			Deleting from			Space
	Index/Key	*Value*	*beginning*	*middle*	*end*	*beginning*	*middle*	*end*	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)

Table 6. Array Operations time complexity

Operation	Time Complexity	Usage
`push`	O(1)	Insert element on the right side.
`pop`	O(1)	Remove the rightmost element.
`[]`	O(1)	Search for element by index.
`indexOf`	O(n)	Search for element by value.
`unshift`	O(n)	Insert element on the left side.

`shift`	O(n)	Remove leftmost element.
`splice`	O(n)	Insert and remove from anywhere.
`slice`	O(n)	Returns a shallow copy of the array.

2.1.2. Array Patterns for Solving Interview Questions

Many programming problems involve manipulating arrays. Here are some patterns that can help you improve your problem-solving skills.

Two Pointers Pattern

Usually, we use one pointer to navigate each element in an array. However, there are times when having two pointers (left/right, low/high) comes in handy. Let's do some examples.

AR-A) *Given a sorted* `array` *of integers, find two numbers that add up to a* `target` *and return their values.*

Function Signature

```
/**
 * Find two numbers that add up target.
 * @param arr - The array of integers
 * @param target - The target
 * @returns {number[]} - array with the values that add up to target.
 */
function twoSum(arr, target) {
  // give it a try on your own ...
}
```

Examples

```
twoSum([ -5, -3, 1, 10 ], 7); // [-3, 10] // (10 - 3 = 7)
twoSum([ -5, -3, -1, 1, 2 ], 30); // [] // no 2 numbers add up to 30
twoSum([ -3, -2, -1, 1, 1,  3,  4], -4); // [-3, -1] // (-3 -1 = -4)
```

Solutions:

One naive solution would be to use two pointers in a nested loop:

Solution 1: Brute force

```
function twoSum(arr, target) {
  for (let i = 0; i < arr.length - 1; i++)
    for (let j = i + 1; j < arr.length; j++)
      if (arr[i] + arr[j] === target) return [arr[i], arr[j]];
  return [];
}
```

The runtime of this solution would be `O(n^2)`. Because of the nested loops. Can we do better? We are not using the fact that the array is SORTED!

We can use two pointers: one pointer starting from the left side and the other from the right side.

Depending on whether the sum is bigger or smaller than the target, we move right or left. If the sum is equal to the target, we return the current left and right pointer's values.

Solution 2: Two Pointers

```
function twoSum(arr, target) {
  let left = 0, right = arr.length -1;
  while (left < right) {
    const sum = arr[left] + arr[right];
    if (sum === target) return [arr[left], arr[right]];
    else if (sum > target) right--;
    else left++;
  }
  return [];
}
```

These two pointers have a runtime of `O(n)`.

This technique only works for sorted arrays. If the array was not sorted, you would have to sort it first or choose another approach.

Sliding Window Pattern

The sliding window pattern is similar to the two pointers. The difference is that the distance between the left and right pointer is always the same. Also, the numbers don't need to be sorted. Let's do an example!

AR-B) *Find the max sum of an array of integers, only taking* `k` *items from the right and left side sequentially.* **Constraints**: `k` won't exceed the number of elements in the array: `1 <= k <= n`.

Function Signature

```
/**
 * Find the max sum of an array of integers,
 * only taking `k` items from the right and left side.
 *
 * @param {number[]} arr - The array of integers
 * @param {number} k - The number of elements to sum up.
 * @returns {number}
 */
function maxSum(arr, k) {
  // Give it a try
};
```

Examples

```
maxSum([1,2,3], 3); // 6 // (1 + 2 + 3 = 6)
maxSum([1,1,1,1,200,1], 3); // 202 // (1 + 200 + 1 = 202)
maxSum([3, 10, 12, 4, 7, 2, 100, 1], 3); // 104 // (3 + 1 + 100 = 104)
maxSum([1,200,1], 1); // 6 // (1 + 2 + 3 = 6)
```

There are multiple ways to solve this problem. Before applying the sliding window, let's consider this other algorithm:

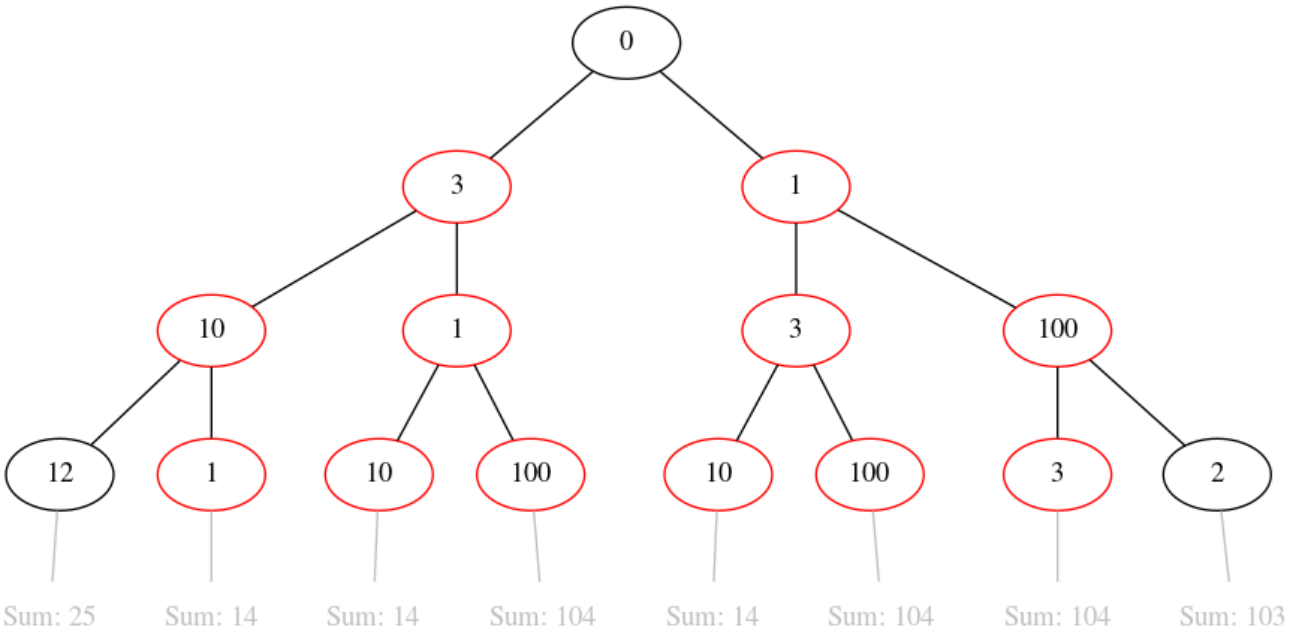
Backtracking algorithm

Let's take `[3, 10, 12, 4, 7, 2, 100, 1], k = 3` as an example.

- We have two initial choices: going left with `3` or right with `1`.
- We can take the first element from the left side `3`; from there, we can keep going left with `10` or right `1`.
- If we go right with `1` on the right side, next, we have two options from the right side `100` or `10`.
- If we go with `100`, then we compute the sum `3 + 1 + 100 = 104`.
- Repeat with other combinations and keep track of the max sum.

How many combinations can we form? `2^k`, since in the worst-case `k` is equal to `n`, then we have a runtime of `O(2^n)`!

We can also visualize all the options as follows. If you add up the numbers from top to bottom, you get the result for all combinations:

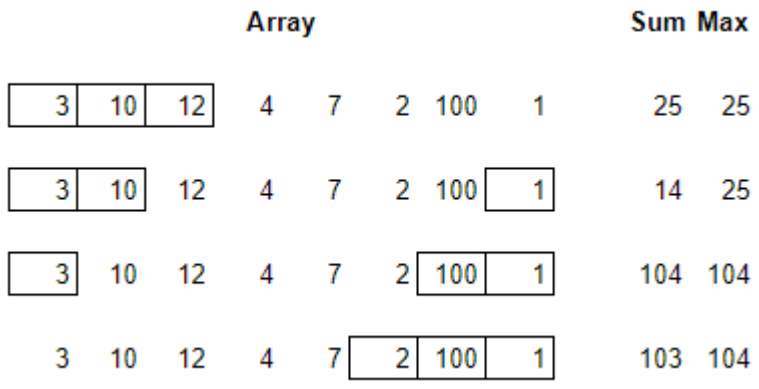


Notice that many middle branches (in red color) have the same numbers, but in a different order, so their sums oscillate between 104 and 14. That's why this algorithm is not very optimal for this problem.

Sliding window algorithm

Another approach is using sliding windows. Since the sum always has k elements, we can compute the cumulative sum for the k first elements from the left. Then, we slide the "window" to the right and remove one from the left until we cover all the right items. In the end, we would have all the possible combinations without duplicated work.

Check out the following illustration:



Here's the implementation:

Solution: using sliding window pointers

```
function maxSum(arr, k) {
  let left = k - 1;
  let right = arr.length -1;
  let sum = 0;
  for (let i = 0; i < k; i++) sum += arr[i];
  let max = sum;

  for (let i = 0; i < k; i++) {
    sum += arr[right--] - arr[left--];
    max = Math.max(max, sum);
  }

  return max;
};
```

The difference between the two pointers pattern and the sliding windows, it's that we move both pointers at the same time to keep the length of the window the same.

The runtime for this code is: `k`. As we move the window k times. Since `k <= n`, the final runtime is `O(n)`.

2.1.3. Practice Questions

Max Subarray

AR-1) *Given an array of integers, find the maximum sum of consecutive elements (subarray).*

Examples:

```
maxSubArray([1, -3, 10, -5]); // 10 (taking only 10)
maxSubArray([-3, 4,-1, 2, 1, -5]); // 6 (sum [4,-1, 2, 1])
maxSubArray([-2, 1, -3, 4, -1, 3, 1]); // 7 (sum [4,-1, 3, 1])
```

Common in interviews at: FAANG, Microsoft

```
/**
 * Find the maximun sum of contiguous elements in an array.
 *
 * @examples
 *  maxSubArray([1, -3, 10, -5]); // => 10
 *  maxSubArray([-3,4,-1,2,1,-5]); // => 6
 *
 * @param {number[]} a - Array
 * @returns {number} - max sum
 */
function maxSubArray(a) {
  // write you code here
}
```

Solution: Max Subarray

Best Time to Buy and Sell a Stock

AR-2) *You have an array of integers. Each value represents the closing value of the stock on that day. You have only one chance to buy and then sell. What's the maximum profit you can obtain? (Note: you have to buy first and then sell)*

Examples:

```
maxProfit([1, 2, 3]) // 2 (buying at 1 and selling at 3)
maxProfit([3, 2, 1]) // 2 (no buys)
maxProfit([5, 10, 5, 10]) // 5 (buying at 5 and selling at 10)
```

Common in interviews at: Amazon, Facebook, Bloomberg

```
/**
 * Find the max profit from buying and selling a stock given their daily
prices.
 * @examples
 *    maxProfit([5, 10, 5, 10]); // => 5
 *    maxProfit([1, 2, 3]); // => 2
 *    maxProfit([3, 2, 1]); // => 0
 * @param {number[]} prices - Array with daily stock prices
 * @returns {number} - Max profit
 */
function maxProfit(prices) {
  // write you code here
}
```

Solution: *Best Time to Buy and Sell a Stock*

2.2. Map

A Map is a data structure where a `key` is mapped to a `value`. It's used for a fast lookup of values based on the given key. Only one key can map to a value (no key duplicates are possible).

Map has many terms depending on the programming language. Here are some other names: Hash Map, Hash Table, Associative Array, Unordered Map, Dictionary.

2.2.1. Map Applications

Maps are one of the most popular data structures because of their fast lookup time.

Holding key/value pairs have many applications like:

- **Caching**: keys are URLs, and values are website content.
- **Indexing**: keys are words, and values are the list of documents where they are found.
- **Spell checking**: keys are English words.
- **Networks**: the key is an IP address/port number, while the value is the corresponding application.

There are many other use cases. We will explore some techniques that we can use to speed up your code with it. But first, let's get the fundamentals out of the way.

2.2.2. Map vs Array

A map shares some similarities with an array. In an array, the key/index is always a number, while the value in a Map can be anything!

Both an Array and Map are very fast for getting values by key in constant time O(1) on average.

A Map uses an array internally. It translates the key into an array's index using a hash function. That's why it is also called "Hash Map" or "Hash Table".

2.2.3. Map vs Objects

JavaScript has two ways to use Maps: one uses objects (`{}`), and the other is using the built-in `Map`.

Using Objects as a HashMap.

```
const objMap = {};
// mapping values to keys
objMap['str'] = 'foo'; // string as key
objMap[1] = 'bar'; // number as key
objMap[{}] = 'test1'; // object as key (not recommended)
const obj1 = {};
objMap[obj1] = 'test2'; // object as key (not recommended)

// searching values by key
console.log(objMap[1]); //↪ bar
console.log(objMap['str']); //↪ foo
console.log(objMap[{}]); //↪ test2 👀
console.log(objMap[obj1]); //↪ test2 👀

console.log(objMap); // {1: "bar", str: "foo", [object Object]: "test"}
```

Notice that the `objMap[{}]` and `objMap[obj1]` return the same value! They both were converted to `[object Object]` as a key.

Let's now use the built-in Map

JavaScript Built-in Map Usage

```
const myMap = new Map();
// mapping values to keys
myMap.set('str', 'foo'); // string as key
myMap.set(1, 'bar'); // number as key
myMap.set({}, 'test1'); // object as key
const obj1 = {};
myMap.set(obj1, 'test2');

// searching values by key
console.log(myMap.get(1)); //↪ bar
console.log(myMap.get('str')); //↪ foo
console.log(myMap.get({})); //↪ undefined 👀
console.log(myMap.get(obj1)); //↪ test2

console.log(myMap);
// Map(4) {"str" => "foo", 1 => "bar", {…} => "test1", {…} => "test2"}
```

As you can see, `Map` handled other objects as a key much better.

Objects are one of the oldest data structures in JavaScript. Maps were introduced as part of the ES2015 enhancements to solve the shortcomings of using Object as a Hashmap. Having two

methods can be confusing. We are going to make it clear when to use one or the other.

Map vs. Object main differences:

- **Object**'s keys should be strings, numbers, or symbols. **Map**'s keys can be anything! Strings, numbers, symbols, arrays, objects, and even other maps!
- **Objects** are not guaranteed to be in insertion order. **Maps** guarantee insertion order.
- When using **Objects** as HashMaps, they might be polluted with other keys defined at the prototype chain. You need to use `hasOwnProperty` or `Object.keys` to avoid these issues. **Maps** doesn't get polluted by the prototype chain.
- **Maps** has been optimized for cases of frequent additions and removals. **Objects** are not optimized.

When do you use an Object over a Map then? When you need to use JSON since it doesn't support Maps yet.

You can convert from Map to Object and vice-versa:

```
const objMap = Object.fromEntries(myMap.entries());   // map -> obj
const map = new Map(objMap.entries());                // obj -> map
```

For completeness, here are some of the most basic operations side-by-side.

Object vs Map Side-by-Side

```
//
// Initialization
//
const obj1 = {};                              // Empty
const obj2 = { adrian: 33, nathalie: 32 };  // w/values

const map1 = new Map();                                    // Empty
const map2 = new Map([['adrian', 33], ['nathalie', 32]]); // w/values

//
// Access
//
assert.equal(obj1.adrian, undefined);
assert.equal(obj2['adrian'], 33); // also "obj2.adrian"

assert.equal(map1.get('adrian'), undefined);
assert.equal(map2.get('adrian'), 33);

//
// Check if the key exists
//
```

```
assert.equal(obj1.adrian !== undefined, false);
assert.equal(obj2['adrian'] !== undefined, true);

assert.equal(map1.has('adrian'), false);
assert.equal(map2.has('adrian'), true);

//
// Adding new elements
//
obj2['Abi'] = 2;
obj2['Dudu'] = 2;

map2.set('Abi', 2).set('Dudu', 2);

//
// Deleting
//
delete obj2.Dudu;

map2.delete('Dudu');

//
// Iterating key/value pairs with for loops
//
for (var k in obj2){
   console.log(`key: ${k}, value: ${obj2[k]}`);
}

for (const [k, v] of map2){
  console.log(`key: ${k}, value: ${v}`);
}

//
// Iterating key/value pairs with
//
Object.keys(obj2)
  .forEach(k => console.log(`key: ${k}, value: ${obj2[k]}`));

map2
  .forEach((v, k) => console.log(`key: ${k}, value: ${v}`));

//
// Getting the size
//
assert.equal(Object.keys(obj2).length, 3);
assert.equal(map2.size, 3);
```

```
//
// Representation
//
console.log(obj2);
// { adrian: 33, nathalie: 32, Abi: 2 }
console.log(map2);
// Map { 'adrian' => 33, 'nathalie' => 32, 'Abi' => 2 }
```

From this point on, we will use built-in Maps (and not objects).

2.2.4. Key by Reference vs. by Value

There's a catch when you use objects/arrays/classes as keys on a `Map`. JavaScript will match the key only if it has the same reference in memory.

Look at the following example:

Array as a Map's key

```
const map = new Map();

map.set([1, 2, 3], 'value');
console.log(map.get([1, 2, 3])); // undefined 👀
```

Trying to access a Map's value using a complex type is a common gotcha. If you want array as a key to work, you need to hold on to a reference, like the following example.

Array reference as a Map's key

```
const map = new Map();
const arr = [1, 2, 3];

map.set(arr, 'value');
console.log(map.get(arr)); // 'value'
```

The same applies to any key that is not a number, string, or symbol.

Map Inner Workings

There are two popular ways to implement Maps, key/value pair data structures:

- Array + Hash Function: Hash Map
- Balanced Binary Search Tree: TreeMap.

In this chapter, we will focus on the Hash Map implementation, which is the one that JavaScript has built-in. In the next parts, we will cover TreeMap.

A map uses an array to store the values and a hash function that translate the key into an array index behind the scenes.

Let's say we have the following key/value pairs.

```
const map = new Map();

map.set('cat', 2);
map.set('dog', 1);
map.set('rat', 7);
map.set('art', 8);
```

There are multiple algorithms for hashing keys. One of them is using modulo division:

1. Convert the key into a number (a.k.a hash code or pre-hashing).
2. Convert the number from step 1 into an array index using modulo. (`hashCode % arrayLength`).

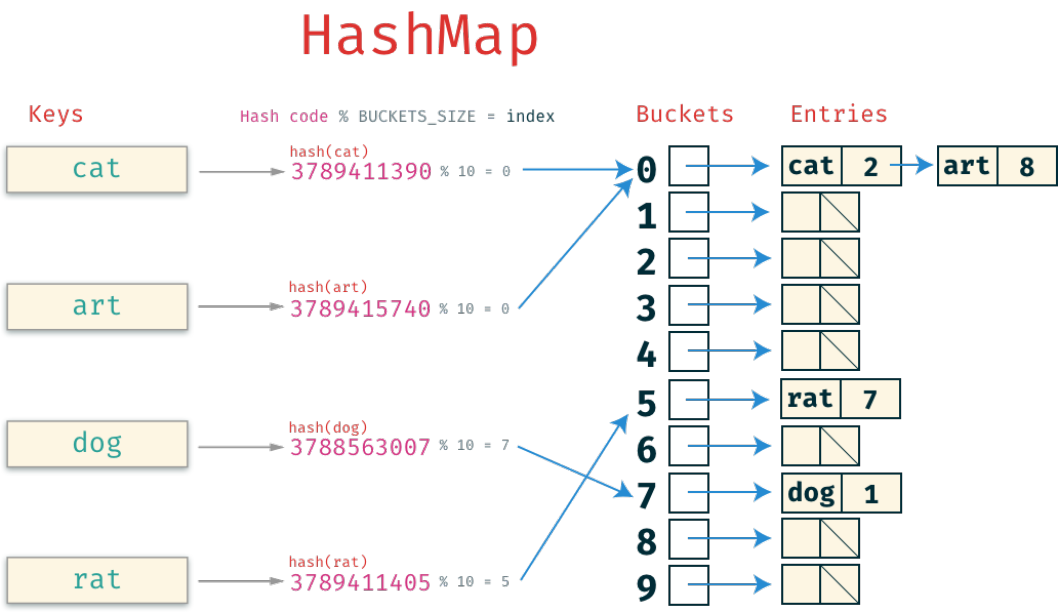


Figure 5. Internal HashMap representation

No hash function is perfect, so it will map two different keys to the same value for some cases. That's what we called a **collision**. When that happens, we chain the results on the same bucket. If we have too many collisions, it could degrade the lookup time from `O(1)` to `O(n)`.

The Map doubles the size of its internal array to minimize collisions when it reaches a certain threshold. This restructuring is called a **rehash**. This **rehash** operation takes `O(n)`, since we have to visit every old key/value pair and remap it to the new internal array. Rehash doesn't happen very often, so statistically speaking, Maps can insert/read/search in constant time `O(1)`.

collisions and rehashes are handled automatically. But it's still good to know the trade-offs. We will go into more detail when we compare it with TreeMaps.

HashMap time complexity

Hash Map is optimal for searching values by key in constant time **O(1)**. However, searching by value is not any better than an array since we have to visit every value **O(n)**.

Table 7. Time complexity for a Hash Map

Data Structure	Searching By		Insert	Delete	Space Complexity
	Index/Key	*Value*			
Hash Map	O(1)	O(n)	O(1)*	O(1)	O(n)

* = Amortized run time. E.g. rehashing might affect run time.

As you can notice, we have amortized times since it will take O(n) while it resizes in the unfortunate case of a rehash. After that, it will be **O(1)**.

2.2.5. HashMap Patterns for Solving Interview Questions

HashMaps are one of the most versatile data structures. You can speed up many programs by using them correctly. In this section, we are going to explore some common patterns.

Smart Caching

One everyday use case for key/value data structures is caching. If you are working on a trendy website, you can save scale better if you cache the results instead of hitting the database and other expensive services every time. That way, you can server many more users requesting the same data.

A common issue with cache you want to expire data you don't often use to make room for hot data. This next exercise is going to help you do that.

HM-A) *Design a Least Recently Used (LRU) cache. This cache has a limit on the number of items it can store. Once the limit it's reached, it will discard the least recently used item. Design a class that takes a limit value, and the methods put and get, to insert and get data.*

Signature

```
/**
 * Least Recently Used (LRU) cache.
 * Key/Value storage with fixed max number of items.
 * Least recently used items are discarded once the limit is reached.
 * Reading and updating the values mark the items as recently used.
 */
class LRUCache {
  /**
   * @param {number} capacity - The max number of items on the cache
   */
  constructor(capacity) {

  }

  /**
   * Get the value associated with the key. Mark keys as recently used.
   * @param {number} key
   * @returns {number} value or if not found -1
   */
  get(key: number): number {

  }

  /**
   * Upsert key/value pair. Updates mark keys are recently used.
   * @param {number} key
   * @param {number} value
   * @returns {void}
   */
  put(key, value) {

  }
}
```

Examples:

```
const c = new LRUCache(2); // capacity: 2
c.put(2, 1); // Cache is [2:1]
c.put(1, 1); // Cache is [2:1, 1:1]
c.put(2, 3); // Cache is [1:1, 2:3]
c.put(4, 1); // Removed 1. Cache is [2:3, 4:1]
c.get(1); // Returns -1 (key 1 not found)
c.get(2); // Returns 3. Cache is [4:1, 2:3]
c.put(5, 5); // Removed key 4. Cache is [2:3, 5:5]
c.get(4); // Returns -1 (key 4 not found)
```

Try it on your own before reading the solution on the next page!

Solution

The LRU cache behavior is almost identical to the Map.

The differences are:

- LRU cache has a limited size, while Map grows until you run out of memory.
- LRU cache removes the least used items once the limit is reached.

We can extend the Map functionality. Also, the Map implementation on JavaScript already keeps the items by insertion order. Every time we read or update a value, we can remove it from where it was and add it back. That way, the oldest (least used) it's the first element on the Map.

Solution: extending Map

```
class LRUCache extends Map {
  constructor(capacity) {
    super();
    this.capacity = capacity;
  }

  get(key) {
    if (!super.has(key)) return -1;
    const value = super.get(key);
    this.put(key, value); // re-insert at the top (most recent).
    return value;
  }

  put(key, value) {
    if (super.has(key)) super.delete(key);
    super.set(key, value);
    if (super.size > this.capacity) {
      const oldestKey = super.keys().next().value;
      super.delete(oldestKey);
    }
  }
}
```

Notice that we call `put` within `get`. This is to rotate the keys to the top (most recent place).

Complexity Analysis

- Time Complexity: `O(1)`. All operations read, write, update, and delete takes `O(1)`.
- Space complexity: `O(k)`. In this case, k, is the capacity of the cache. Even if n has 1 million items, we would only hold to the k most recent ones.

Trading Speed for Space

Maps have a `O(1)` runtime for lookups and `O(n)` space complexity. It can improve the speed of programs in exchange for using a little bit more of memory. Let's do an example.

Let's say you are working on a webcrawler, and for each site, you want to find out the most common words used. How would you do it?

HM-B) *Given a text, return the most common words in descending order. You should sanitize the input by removing punctuation `!?',;.` and converting all letters to lowercase. Return the most common words in descending order.*

Signature

```
/**
 * Given text and banned words,
 * return the most common words in descending order.
 * @param {string} text - The text to parse.
 * @param {number} n - The number of results.
 * @return {string[]}
 */
function mostCommonWords(text, n = 1) {
  // you code goes here
}
```

Examples:

```
mostCommonWords(
  'The map, maps keys to values; Keys can be anything.',
  1); // ['keys']
mostCommonWords(
  'Look at it! What is it? It does look like my code from 1 year ago',
  2); // ['it', 'look']
mostCommonWords(
  'a; a,b, a\'s c A!; b,B,    c.',
  4); // ['a', 'b', 'c', 's']
```

Try it on your own before reading the solution on the next page!

Solutions

This is a problem that has multiple steps:

1. Split the text into lowercased words and remove whitespaces and punctuation.
2. Count the frequency of words.
3. Sort words by frequency and return the top n words.

Possible implementations for each of the steps:

1. We can use regex (regular expressions) and split on non-words `\W+`. The runtime of this will be `O(n)`.
2. Let's discuss this later.
3. We have an array of the word → frequency pairs. We can sort by the frequency using the built-in sort function and return the subarray with the top n results. The time complexity would be `O(n log n)`.

For step 2, we can do it in multiple ways. A brute force solution is using 2 for loops to count the number of times each word appear:

Solution 1: Brute Force

```
function mostCommonWordsBrute(text, n = 1) {
  const words = text.toLowerCase().split(/\W+/);
  const entries = []; // array of [word, count] pairs

  for (let i = 0; i < words.length; i++) {
    if (!words[i]) continue;
    let count = 1;
    for (let j = i + 1; j < words.length; j++) {
      if (words[i] === words[j]) {
        count++;
        words[j] = null; // removed letter once it's counted.
      }
    }
    entries.push([words[i], count]);
  }

  return entries
    .sort((a, b) => b[1] - a[1])
    .slice(0, n)
    .map((w) => w[0]);
}
```

Notice that we null out the counted words. That's to avoid counting the phrase more than once.

Complexity Analysis:

- Time complexity: `O(n^2)`. We have three steps and each one has its time complexity: O(n) + O(n^2) + O(n log n). Remember that with Big O notation, we only care about the term with the highest order: `n^2`.
- Space complexity: `O(n)`. We use multiple O(n) auxiliary spaces for these variables: `words`, `entries`, and the solution is also n space.

Another alternative is to use a Map to count.

Solution 2: Map counter

```
function mostCommonWords(text, n = 1) {
  const words = text.toLowerCase().split(/\W+/);

  const map = words
    .reduce((m, w) => m.set(w, 1 + (m.get(w) || 0)), new Map());

  return Array.from(map.entries())
    .sort((a, b) => b[1] - a[1])
    .slice(0, n)
    .map((w) => w[0]);
}
```

With this solution, we iterate over the words only once. We first get the current count and add one. If the word didn't exist, we would default to a count of 0. Steps 1 and 3 are almost identical to solution #1.

Complexity Analysis

- Time Complexity: `O(n log n)`. We have 3 steps: O(n) + O(n) + O(n log n). The most significant term is `n log n`.
- Space complexity: `O(n)`. We used the same O(n) auxiliary space as solution #1 for `words`, `entries`, and the solution. Additionally, we added one more O(n) space for the Map.

Sliding Window

We saw sliding windows with arrays before. We are now going to use them with strings. The idea is very similar, we still use the two pointers, and the solution is the "window" between the pointers. We can increase or decrease the window as long as it keeps the constraints of the problem. Let's do an example for better understanding.

HM-C) *Return the length of the longest substring without repeating characters.*

Signature

```
/**
 * Return the length of the longest substring without repeating
characters.
 * @param {string} s
 * @return {number}
 */
function longestSubstring(s) {
  // your code goes here!
};
```

Examples

```
longestSubstring('abcdaefg'); // 7 ('bcdaefg')
longestSubstring('abbaa'); // 2 ('ab')
longestSubstring('abbadvdf') // 4 ('badv')
```

Try it on your own before reading the solution on the next page!

Solutions

We are going to solve this problem by using a sliding window approach. We have two pointers called `lo` and `hi`. They start both at zero, and we increase the window as long as they don't have duplicates. If we found a duplicate, we reopen a new window past the duplicated value.

Take a look at this illustration doing an example for the string `abbadvdf`:

Sliding Window for "abbadvdf"

lo	hi		0	1	2	3	4	5	6	7		length	max
0	0		a	b	b	a	d	v	d	f		1	1
0	1		a	b	b	a	d	v	d	f		2	2
2	2		a	b	b	a	d	v	d	f		1	2
2	3		a	b	b	a	d	v	d	f		2	2
2	4		a	b	b	a	d	v	d	f		3	3
2	5		a	b	b	a	d	v	d	f		4	4
5	6		a	b	b	a	d	v	d	f		2	4
5	7		a	b	b	a	d	v	d	f		3	4

As you can see, we calculate the length of the string on each iteration and keep track of the maximum value.

What would this look like in code? Let's try a couple of solutions. Let's go first with the brute force and then how we can improve it.

We can have two pointers, `lo` and `hi`, to define a window. We can use two for-loops for that. Later, within `lo` to `hi` window, we want to know if there's a duplicate value. A simple and naive approach is to use another two for-loops to check for duplicates (4 nested for-loop)! We need labeled breaks to skip updating the max if there's a duplicate.

The following code can hurt your eyes. Don't try this in production; for better solutions, keep reading.

Solution 1: Super Brute Force

```
/**
 * Return the length of the longest substring without repeating
characters.
 * @param {string} s
 * @return {number}
 */
function longestSubstring(s) {
  let max = 0;

  for (let lo = 0; lo < s.length; lo++) {
    repeatedFound:
    for (let hi = lo; hi < s.length; hi++) {
      // check if it's unique withing [lo,hi] range
      for (let i = lo; i < hi; i++) {
        for (let j = lo + 1; j <= hi; j++) {
          if (i !== j && s[i] === s[j]) break repeatedFound;
        }
      }
      // all are unique between [lo,hi] range
      max = Math.max(max, hi - lo + 1);
    }
  }

  return max;
};
```

This function gets the job done. But how efficient is it?

Complexity Analysis

- Time Complexity: `O(n^4)`. In the worst-case, when the string has all unique characters, we have n^4!
- Space complexity: `O(1)`. The only variable we are using is integers.

Solution 1 has a horrible runtime, but the space complexity is constant. Can we trade space for a better speed performance? Absolutely!

Instead of having two loops for finding duplicates, we can do one pass and use a Map to detect duplicates.

Solution 2: Brute force with Map

```
/**
 * Return the length of the longest substring without repeating
characters.
 * @param {string} s
 * @return {number}
 */
function longestSubstring(s) {
  let max = 0;

  for (let lo = 0; lo < s.length; lo++) {
    repeatedFound:
    for (let hi = lo; hi < s.length; hi++) {
      // check if it's unique withing [lo,hi] range
      const map = new Map();
      for (let i = lo; i <= hi; i++) {
        if (map.has(s[i])) break repeatedFound;
        map.set(s[i], true);
      }
      // all are unique between [lo,hi] range
      max = Math.max(max, hi - lo + 1);
    }
  }

  return max;
}
```

We are using the Map to detect duplicates, where the characters are the keys.

Complexity Analysis

- Time Complexity: `O(n^3)`. We have three nested loops that, in the worst-case, each will visit `n` items.
- Space complexity: `O(n)`. We have a map that might grow as big as size `n`.

One optimization that we can do the solution 2 is to store the index where we last saw a character. We can map each character to its index. That way, when we find a duplicate, we can update the `lo` pointer with it, shrinking the window.

Solution 3: Optimized Sliding Window

```
/**
 * Return the length of the longest substring without repeating
characters.
 * @param {string} s
 * @return {number}
 */
function longestSubstring(s) {
  const map = new Map();
  let max = 0;

  for (let hi = 0, lo = 0; hi < s.length; hi++) {
    if (map.has(s[hi])) lo = Math.max(lo, map.get(s[hi]) + 1);
    map.set(s[hi], hi);
    max = Math.max(max, hi - lo + 1);
  }

  return max;
};
```

This solution has the least amount of code, and it's also the most efficient!

Something that might look unnecessary is the `Math.max` when updating the `lo` pointer. You can remove it and try running it with the string "abba", what would happen?

Complexity Analysis

- Time Complexity: `O(n)`. We do one pass and visit each character once.
- Space complexity: `O(n)`. We store everything on the Map so that the max size would be `n`.

2.2.6. Practice Questions

Fit two movies in a flight

HM-1) *You are working in an entertainment recommendation system for an airline. Given a flight duration (target) and an array of movies length, you need to recommend two movies that fit exactly the length of the flight. Return an array with the indices of the two numbers that add up to the target. No duplicates are allowed. If it's not possible to return empty* `[]`.

Common in interviews at: FAANG.

Examples:

```
twoSum([113, 248, 80, 200, 91, 201, 68], 316); // [1, 6] (248 + 68 =
316)
twoSum([150, 100, 200], 300); // [2, 3] (100 + 200 = 300)
twoSum([150, 100, 200], 150); // [] (No two numbers add up to 150)
```

Starter code:

```
/**
 * Find two numbers that add up to the target value.
 * Return empty array if not found.
 * @example twoSum([19, 7, 3], 10) // => [1, 2]
 * @param {number[]} nums - Array of integers
 * @param {number} target - The target sum.
 * @returns {[number, number]} - Array with index 1 and index 2
 */
function twoSum(nums, target) {
  // write your code here...
}
```

Solution: Fit two movies in a flight

Subarray Sum that Equals K

HM-2) *Given an array of integers, find all the possible subarrays to add up to k. Return the count.*

Common in interviews at: FAANG

Examples:

```
subarraySum([1], 1); // 1 (1 equals to 1 :)
subarraySum([1, 1, 1], 1); // 3 ([1], [1], [1] equals 1)
subarraySum([1, -1, 1], 0); // 2 (sum([1, -1]), sum([-1, 1]) equals 0)
subaraySum([1, 2, 3, 0, 1, 4, 0, 5], 5) // 8
// All of these 8 sub arrays add up to 5:
// [2, 30], [2,3,0], [0,1,4], [0,1,4,0], [1,4], [1,4,0], [0,5], [5]
```

Starter code:

```
/**
 * Find the number of subarrays that add up to k.
 * @example subarraySum([1, -1, 1], 0); // 3 ([1,-1,1], [1], [1])
 * @param {number[]} nums - Array of integers.
 * @param {number} k - The target sum.
 * @returns {number} - The number of solutions.
 */
function subarraySum(nums, k) {
  // write your code here...
}
```

Solution: Subarray Sum that Equals K

2.3. Set

Set is a data structure that allows you to store unique values. If you try to add the same value multiple times, the Set will only add it once and ignore all other requests. Also, you can check very quickly if a value exists or not. Searching by value on arrays takes `O(n)`. However, searching by value on a Set takes `O(1)` on average.

A Set can be implemented in different ways. One way it's using a Hash Map, and other is using a Tree Map. JavaScript has a built-in Hash Set, so that' the one we are going to focus on.

We will go more in details with Tree Map after we cover the Binary Search Tree.

2.3.1. Set vs Array

An array allows you to search a value by index in constant time `O(1)`; however, if you don't know the index, searching a value would take you linear time `O(n)`. A Set has doesn't allow you to search value by index, but you can search by value in constant time. The `Set.add` and `Set.has` method both are `O(1)` in average.

Take a look at the following examples:

Set usage example (using JavaScript built-in Set)

```
const set = new Set();

set.add(1); //↪ Set [ 1 ]
set.add(1); //↪ Set [ 1 ]
set.add(2); //↪ Set [ 1, 2 ]
set.add(3); //↪ Set [ 1, 2, 3 ]
set.has(1); //↪ true
set.delete(1); //↪ removes 1 from the set
set.has(1);    //↪ false, 1 has been removed
set.size; //↪ 2, we just removed one value
console.log(set); //↪ Set(2) {2, 3}
```

As you can see, even if we insert the same value multiple times, it only gets added once.

Like a map, you can also insert objects, arrays, maps, and even other sets. However, be careful because anything that is not a number, string, or symbol would be matched by reference. Let's do some examples.

Using a Set with objects

```
const set = new Set();

// matching by value
set.add({a: 1, b: 2});
set.has({a: 1, b: 2}); // ↪ false
set.add({a: 1, b: 2}); // not ignored

// matching by reference
const a = {a: 1, b: 2};
set.add(a);
set.has(a); // ↪ true
set.add(a); // this requests will be ignore.

// Set has 3 arrays with the same value, but since they all have
different memory address it's allowed.
console.log(set); // Set { {a: 1, b: 2}, {a: 1, b: 2}, {a: 1, b: 2} }
```

As you can see, you can't find an object using a new object (e.g. `{a: 1, b: 2}`); you need the reference to find it. If you need to match by value, you would need to convert it to a string using `JSON.stringify`.

Workaround to find objects by value.

```
const set = new Set();

set.add(JSON.stringify({a: 1, b: 2}));
set.add(JSON.stringify({a: 1, b: 2})); // ignored

set.has(JSON.stringify({a: 1, b: 2})); // ↪ true

// Only one object, since strings are matched by value and not by
reference.
console.log(set); // Set { '{"a":1,"b":2}' }
```

2.3.2. Removing duplicates from an array.

One typical case for a Set is to eliminate duplicates from an array.

Removing duplicates from an array

```
const arr = [1, 2, 2, 1, 3, 2];

// convert array to set
const set = new Set(arr);
// convert set to array
const uniqueValues = Array.from(set);
// check array
console.log(uniqueValues); // [ 1, 2, 3 ]
```

You can also do it all in one line.

One-liner to remove duplicates from array.

```
const arr = [1, 2, 2, 1, 3, 2];
console.log([...new Set(arr)]); // [ 1, 2, 3 ]
```

2.3.3. Time Complexity of a Hash Set

All operations on Hash Set are constant time on average: `O(1)`. Like the Hash Map, there are cases when the the Set is getting full, and it would do a rehash taking `O(n)` for that one insertion.

Table 8. Time complexity HashSet

Data Structure	Searching By		Insert	Delete	Space Complexity
	Index/Key	*Value*			
Hash Set	O(1)	-	O(1)*	O(1)	O(n)

* = Amortized run time. E.g. rehashing might affect run time to **O(n)**.

2.3.4. Practice Questions

Most common word

ST-1) *Given a text and a list of banned words. Find the most common word that is not on the banned list. You might need to sanitize the text and strip out punctuation '?!,'.* `

Common in interviews at: Amazon.

Examples:

```
mostCommonWord(
  `How much wood, would a Woodchuck chuck,
  if a woodchuck could chuck?`,
  ['a'],
); // woodchuck or chuck (both show up twice)

mostCommonWord(
`It's a blue ball and its shade... Very BLUE!`,
['and']); // blue (it show up twice, "it" and "its" once)
```

Starter code:

```
/**
 * Find the most common word that is not banned.
 * @example mostCommonWord("It's blue and it's round", ['and']) // it
 * @param {string} paragraph - The text to sanitize and search on.
 * @param {string[]} banned - List of banned words (lowercase)
 * @returns {string} - The first word that is the most repeated.
 */
function mostCommonWord(paragraph, banned) {
  // write your code here...
}
```

Solution: Most common word

Longest Without Repeating

ST-2) *Find the length of the longest substring without repeating characters.*

Common in interviews at: Amazon, Facebook, Bloomberg.

Examples:

```
lenLongestSubstring('aaaaa'); // 1 ('a')
lenLongestSubstring('abccdefg'); // 5 ('cdefg')
lenLongestSubstring('abc'); // 3 ('abc')
```

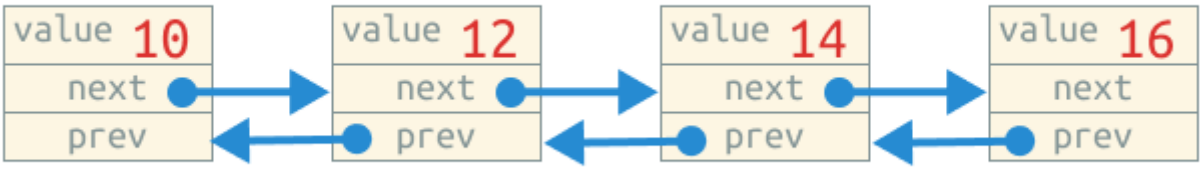
Starter code:

```
/**
 * Find the length of the longest substring without duplicates.
 * @example lenLongestSubstring('abccxyz'); // => 4 (cxyz)
 * @param {string} s - The string.
 * @returns {number} - The length of the longest unique substring.
 */
function lenLongestSubstring(s) {
  // write your code here...
}
```

Solution: Longest Without Repeating

2.4. Linked List

A list (or Linked List) is a linear data structure where each object has a pointer to the next one creating a chain. You can also have a back reference to the previous node.



The data doesn't have to be a number. It can be anything that you need (e.g., images, songs, menu items).

Some features powered by linked lists:

- *Image viewer* – The previous and next images are linked in an image viewer so that the user can navigate them.
- *Previous and next page in web browser* – We can access the previous and next URL searched in a web browser by pressing the back and next button since they are linked.
- *Music Player* - Queue of songs in a music player connects them so you can move to the next song or previous one.

Other Applications:

- Build Stack and Queue data structures, which are useful for Graph Traversal and other things.
- Linked Lists are used on Map to handle collisions.
- Linked Lists can be used when representing a Graph as an adjacency list.
- Operate arbitrary big numbers (think hundreds of digits). Each digit is a node of a linked list.
- Manipulation of polynomials by storing constants in the node of a linked list.
- Representing sparse matrices (an array representation will waste a lot of memory when most of the cells are empty). The linked list will represent only the non-zero values saving significant space.

Hopefully, this will get you excited about learning Linked Lists since it's the base of many interesting applications. Let's learn more about the different types of linked lists.

2.4.1. Types of Linked List

Linked Lists can be:

- **Singly**: every item has a pointer to the next.
- **Doubly**: every item has a reference to the next and the previous.

- **Circular**: the last element points to the first one, forming an infinite loop.

JavaScript doesn't have a built-in List. However, it's straightforward to create.

Linked List Node Implementation

```
/**
 * Linked List Node
 */
class Node {
  constructor(value = null) {
    this.value = value;
    this.next = null;
    this.previous = null; // if doubly linked list
  }
}
```

Let's go one by one!

2.4.2. Singly Linked List

In a singly linked list, each element or node is **connected** to the next one by a reference.

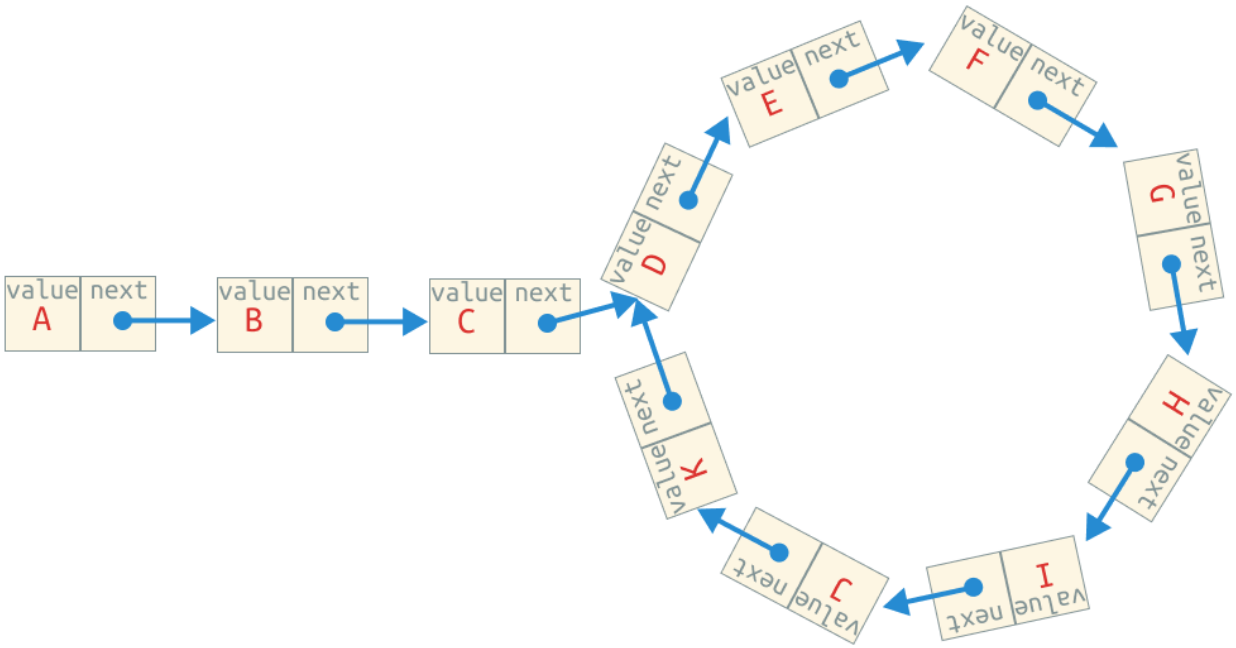
Usually, a Linked List is referenced by the first element called **head** (or **root** node). Let's say that we have a list of strings with the following values: `"art" -> "dog" -> "cat"`. It would look something like the following image.

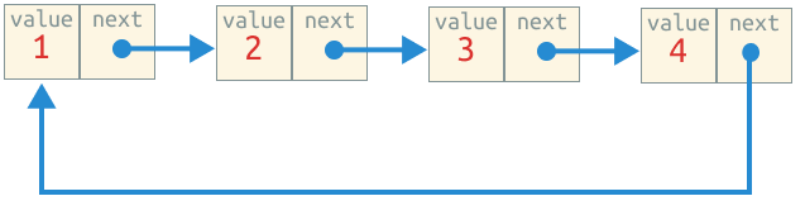
Figure 6. Singly Linked List Representation: each node has a reference (blue arrow) to the next one.

If you want to get the `cat` element from the example above, then the only way to get there is by using the `next` field on the head node. You would get `art` first, then use the next field recursively until you eventually get the `cat` element.

2.4.3. Circular Linked Lists

Circular linked lists happen when the last node points to any node on the list, creating a loop. In the following illustration, you can see two circular linked lists.





One circular linked list happens when the last element points to the first element. Another kind of circular linked list is when the last node points to any node in the middle. There are some efficient algorithms to detect when the list has a loop or not. More on that later in this chapter.

2.4.4. Doubly Linked List

Doubly Linked List has two references to the `next` and `previous` node.

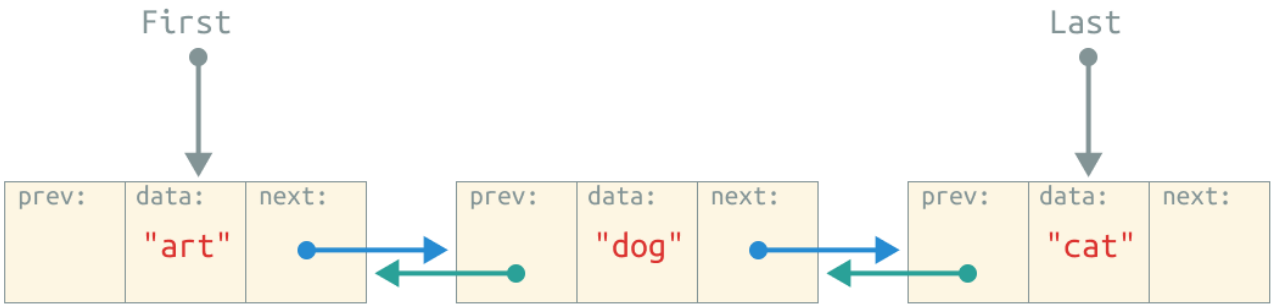


Figure 7. Doubly Linked List: each node has a reference to the next and previous element.

With a doubly-linked list, you can move not only forward but also backward. If you keep a pointer to the `last` element (`cat`), you can step back recursively.

Finding an item on the linked list takes O(n) time. Because in the worst-case, you will have to iterate over the whole list.

2.4.5. Implementing a Linked List

We are going to implement a doubly linked list. First, let's start with the constructor.

if you want to implement a singly linked list instead, it's the same in most parts, but without the setting the `previous` pointers.

The only must-have field on the constructor is the `first` or head reference. If you want to insert data to the back of the list in constant time, then the `last` pointer is needed. Everything else is complimentary.

Linked List's constructor

```
/**
 * Doubly linked list that keeps track of
 * the last and first element
 */
class LinkedList {
  constructor(
    iterable = [],
  ) {
    this.first = null; // head/root element
    this.last = null; // last element of the list
    this.size = 0; // total number of elements in the list

    Array.from(iterable, (i) => this.addLast(i));
  }

  // ... methods go here ...
}
```

The iterable parameter is a nice to have. That will allow you to convert an array of items into a linked list. E.g. `const list = new LinkedList([1, 2, 3]);`

2.4.6. Searching by value or index

There's no other way to find an element by value than iterating through the list. So, the runtime is `O(n)`.

There are two prominent use cases for search: find an element by value, or find them by their index/position.

We can use a for-loop to keep track of the index and the current node simultaneously. Whichever fulfill first, we return that one.

Linked List's searching by values or index

```
/**
 * Find by index or by value, whichever happens first.
 * Runtime: O(n)
 * @example
 *  this.findBy({ index: 10 }).node; // node at index 10.
 *  this.findBy({ value: 10 }).node; // node with value 10.
 *  this.findBy({ value: 10 }).index; // node's index with value 10.
 *
 * @param {Object} params - The search params
 * @param {number} params.index - The index/position to search for.
 * @param {any} params.value - The value to search for.
 * @returns {{node: any, index: number}}
 */
findBy({ value, index = Infinity } = {}) {
  for (let current = this.first, position = 0; ①
    current && position <= index; ②
    position += 1, current = current.next) { ③
    if (position === index || value === current.value) { ④
      return { node: current, index: position }; ⑤
    }
  }
  return {}; // not found
}
```

① We initialize two variables `current` to the first node and `position` to 0 to keep track of the ordinal number.

② While the `current` node is not null, we keep going.

③ On each loop, we move to the next node and increment the index.

④ We check if the index is the one provided or if the node has the expected value.

⑤ Returns the index and the current node if found.

2.4.7. Insertion

You can add elements at the beginning, end, or anywhere in the middle of the list in a linked list. So, let's implement each case.

Inserting elements at the beginning of the list

We will use the Node class to create a new element and stick it at the beginning of the list, as shown below.

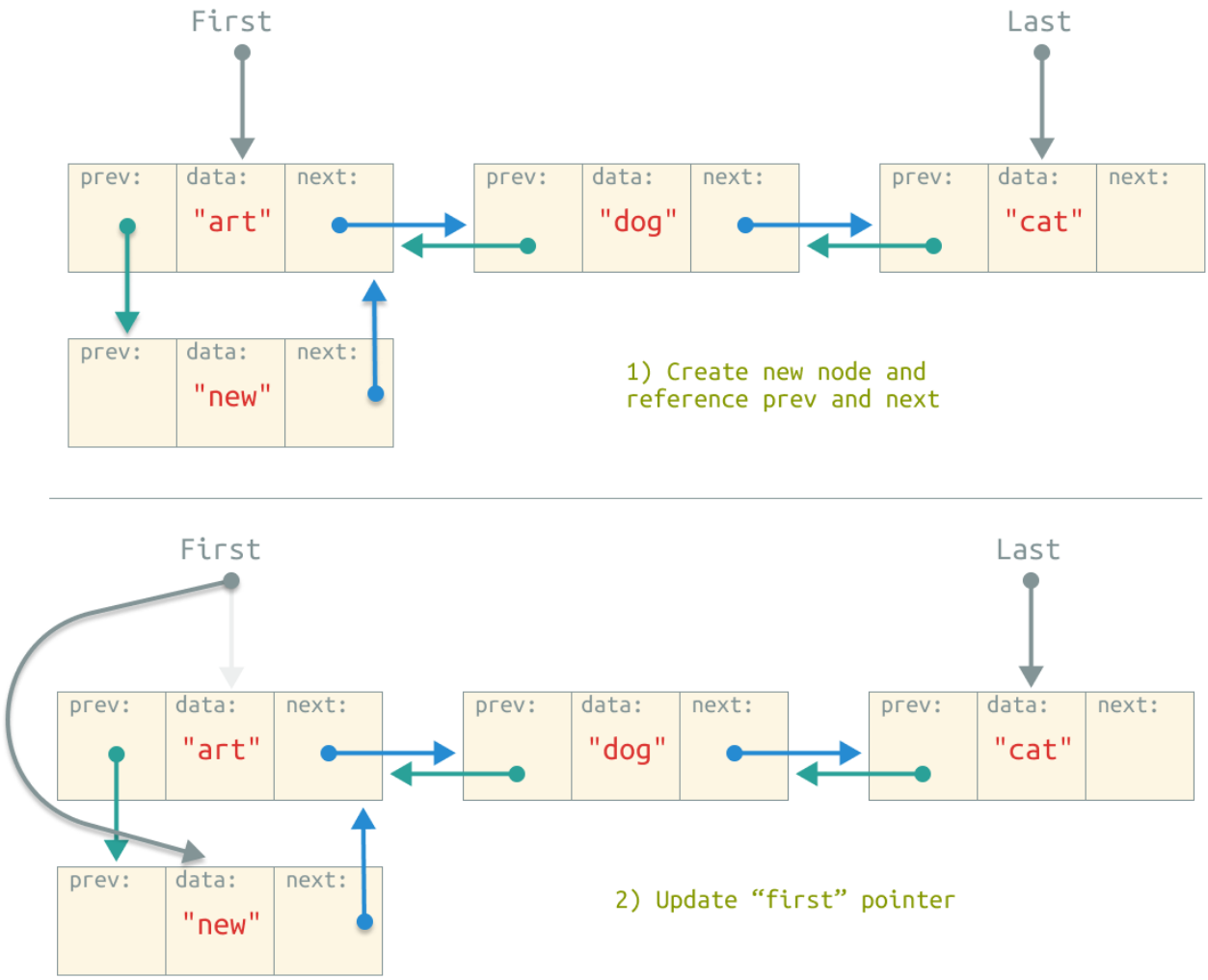


Figure 8. Insert at the beginning by linking the new node with the current first node.

To insert at the beginning, we create a new node with the next reference to the current first node. Then we update the pointer `first` to the new node. In code, it would look something like this:

Add item to the beginning of a Linked List

```
/**
 * Adds element to the begining of the list. Similar to Array.unshift
 * Runtime: O(1)
 * @param {Node} value
 */
addFirst(value) {
  const newNode = new this.ListNode(value);

  newNode.next = this.first;

  if (this.first) { // check if first node exists (list not empty)
    this.first.previous = newNode; ①
  } else { // if list is empty, first & last will point to newNode.
    this.last = newNode;
  }

  this.first = newNode; // update head
  this.size += 1;

  return newNode;
}
```

① It might be confusing seen `this.first.previous`. It means that we are updating the `previous` pointer of the `art` node to point to `new`.

Inserting element at the end of the list

Appending an element at the end of the list can be done very effectively if we have a pointer to the `last` item. Otherwise, you would have to iterate through the whole list.

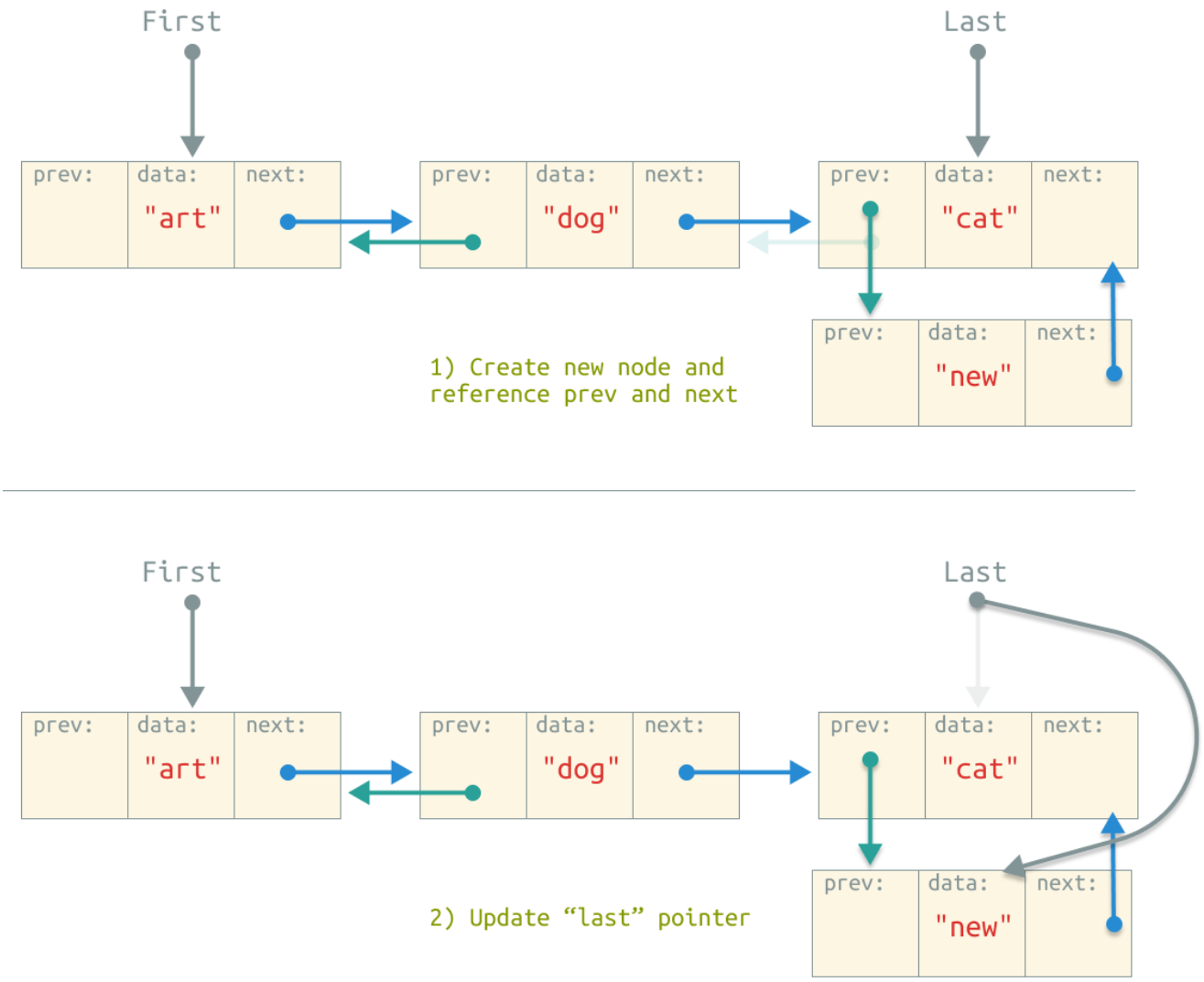


Figure 9. Add element to the end of the linked list

Linked List's add to the end of the list implementation

```
/**
 * Adds element to the end of the list (tail). Similar to Array.push
 * Using the element last reference instead of navigating through the
list,
 * we can reduced from linear to a constant runtime.
 * Runtime: O(1)
 * @param {any} value node's value
 * @returns {Node} newly created node
 */
addLast(value) {
  const newNode = new Node(value);

  if (this.first) { // check if first node exists (list not empty)
    newNode.previous = this.last;
    this.last.next = newNode;
    this.last = newNode;
  } else { // if list is empty, first & last will point to newNode.
    this.first = newNode;
    this.last = newNode;
  }

  this.size += 1;

  return newNode;
}
```

If there's no element in the list yet, the first and last node would be the same. If there's something, we go to the `last` item and add the reference `next` to the new node. That's it! We got a constant time for inserting at the beginning and the end of the list: **O(1)**.

Inserting element at the middle of the list

For inserting an element in the middle of the list, you would need to specify the position (index) in the list. Then, you create the new node and update the references around it.

There are four references to update:

1. New node's `next`.
2. New node's `previous`.
3. New node's previous `next`.
4. New node's next `previous`.

Let's do an example with the following doubly linked list:

```
art <-> dog <-> cat
```

We want to insert the `new` node in the 2nd position (index 1). For that, we first create the "new" node and update the references around it.

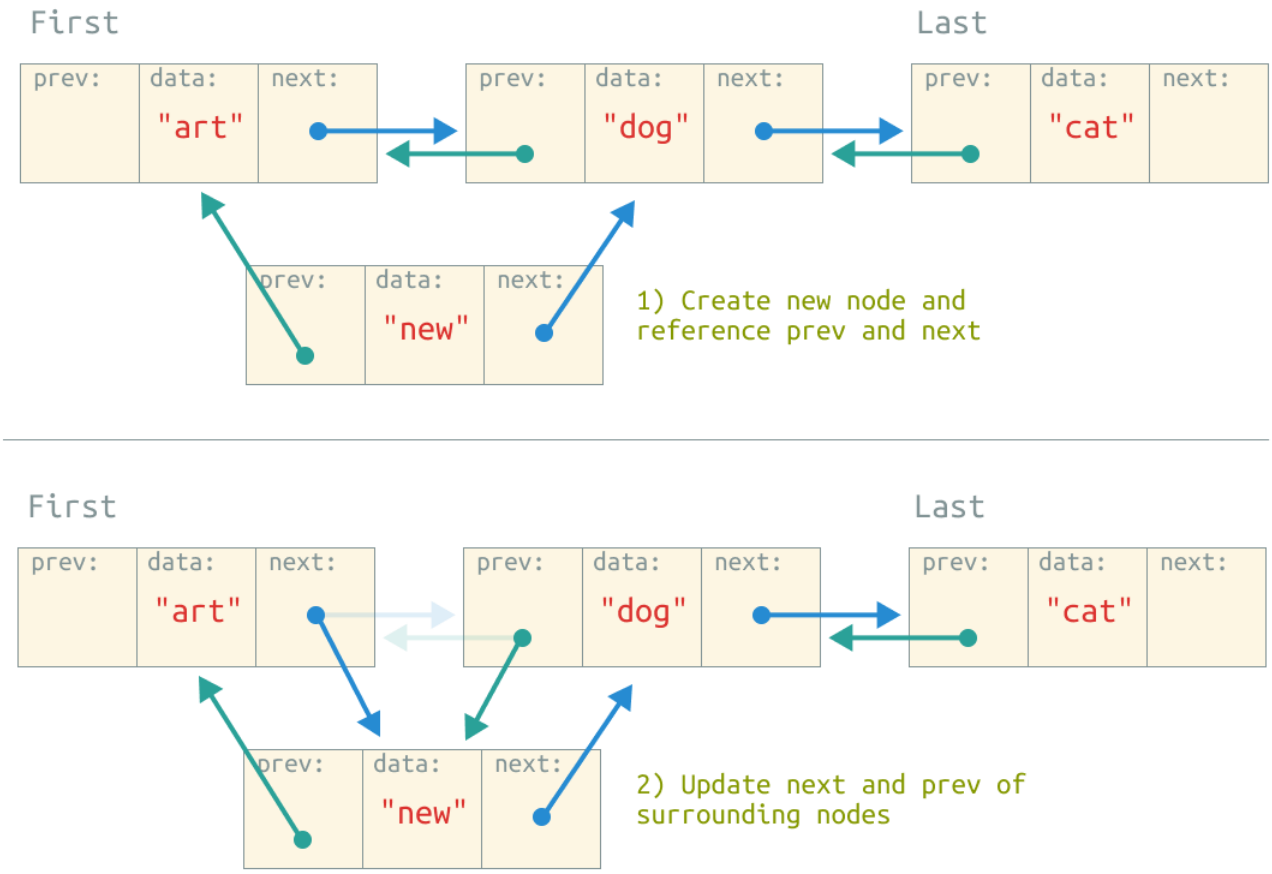


Figure 10. Inserting node in the middle.

Take a look into the implementation of `LinkedList.add`:

Linked List's add to the middle of the list

```
/**
 * Insert new element at the given position (index)
 *
 * Runtime: O(n)
 *
 * @param {any} value new node's value
 * @param {Number} position position to insert element
 * @returns {Node|undefined} new node or 'undefined' if the index is out
of bound.
 */
addAt(value, position = 0) {
  if (position === 0) return this.addFirst(value); ①
  if (position === this.size) return this.addLast(value); ②

  // Adding element in the middle
  const current = this.findBy({ index: position }).node;
  if (!current) return undefined; // out of bound index

  const newNode = new Node(value); ③
  newNode.previous = current.previous; ④
  newNode.next = current; ⑤
  current.previous.next = newNode; ⑥
  current.previous = newNode; ⑦
  this.size += 1;
  return newNode;
}
```

① If the new item goes to position 0, then we reuse the `addFirst` method, and we are done!

② However, if we add to the last position, we reuse the `addLast` method and done!

③ Adding `newNode` to the middle: First, create the `new` node only if it exists. Take a look at Linked List's searching by values or index to see `findBy` implementation again.

④ Set newNode `previous` reference.

⑤ Set newNode `next` link.

⑥ So far, no other node in the list points to `newNode`, so we the `art` node's next point to `new` (refer to the illustration).

⑦ Make `dog` node's previous point to `new`.

Take notice that we reused `addFirst` and `addLast` methods. For all the other cases, the insertion is in the middle. We use `current.previous.next` and `current.next` to update the surrounding elements and point to the new node. Inserting in the middle takes **O(n)** because we have to iterate through the list using the `findBy` method.

2.4.8. Deletion

Deleting is an interesting one. We don't delete an element; we remove all references to that node. The garbage collector will remove it when no one points to it. Let's go case by case to explore what happens.

Deleting element from the head

Deleting the first element (or head) is a matter of removing all references to it.

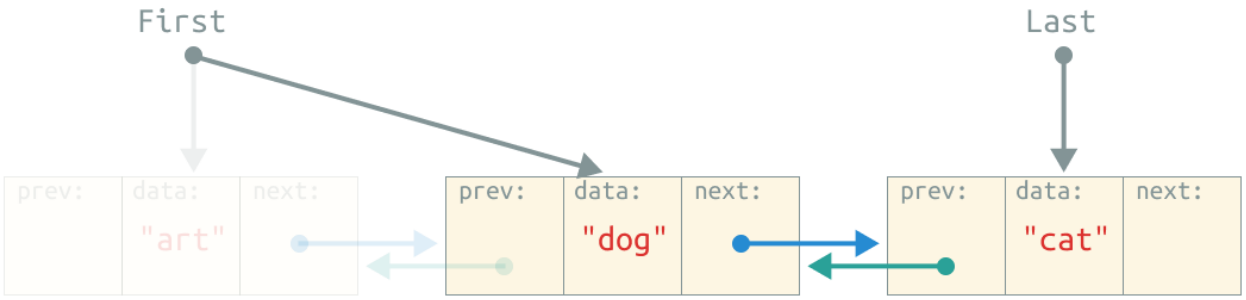


Figure 11. Deleting an element from the head of the list

For instance, to remove the head ("art") node, we change the variable `first` to point to the second node, "dog". We also remove the variable `previous` from the "dog" node, so it doesn't reference the "art" node anymore. The garbage collector will get rid of the "art" node when it sees nothing is using it anymore.

Linked List's remove from the beginning of the list

```
/**
 * Removes element from the start of the list (head/root).
 * Similar to Array.shift().
 * Runtime: O(1)
 * @returns {any} the first element's value which was removed.
 */
removeFirst() {
  if (!this.first) return null; // Check if list is already empty.
  const head = this.first;

  this.first = head.next; // move first pointer to the next element.
  if (this.first) {
    this.first.previous = null;
  } else { // if list has size zero, then we need to null out last.
    this.last = null;
  }
  this.size -= 1;
  return head.value;
}
```

Check for edge cases:

- List is already empty.
- Removing the last node.

As you can see, when we want to remove the first node, we make the 2nd element (`head.next`) the first one.

Deleting element from the tail

Removing the last element from the list would require iterate from the head until we find the last one: `O(n)`. But, since we referenced the last element, we can do it in *O(1)* instead!

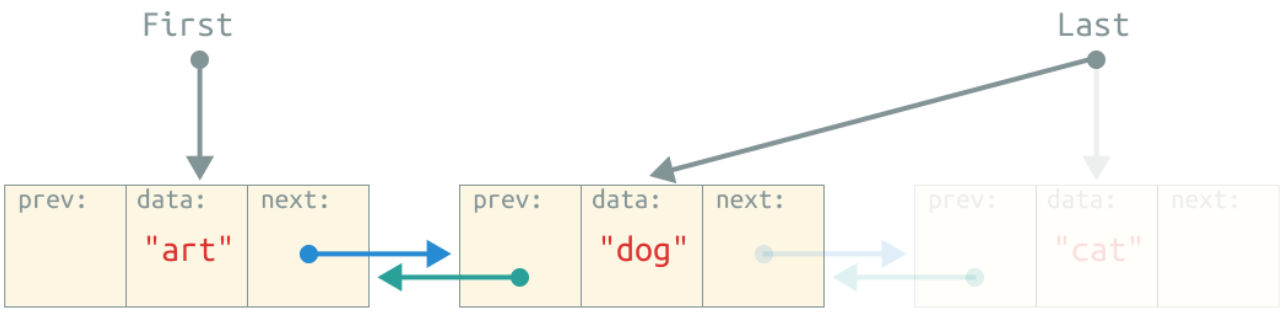


Figure 12. Removing the last element from the list.

For instance, if we want to remove the last node “cat”. We use the last pointer to avoid iterating through the whole list. We check `last.previous` to get the “dog” node and make it the new `last` and remove its next reference to “cat.” Since nothing is pointing to “cat” it is out of the list and eventually is deleted from memory by the garbage collector.

Linked List’s remove from the end of the list

```
/**
 * Removes element to the end of the list.
 * Similar to Array.pop().
 * Runtime: O(1)
 * @returns {any} the last element's value which was removed
 */
removeLast() {
  if (!this.last) return null; // Check if list is already empty.
  const tail = this.last;

  this.last = tail.previous;
  if (this.last) {
    this.last.next = null;
  } else { // if list has size zero, then we need to null out first.
    this.first = null;
  }
  this.size -= 1;
  return tail.value;
}
```

The code is very similar to `removeFirst`, but instead of first, we update `last` reference, and instead of nullifying `previous`, we null out the `next` pointer.

Deleting element from the middle

To remove a node from the middle, we make the surrounding nodes bypass the one we want to delete.

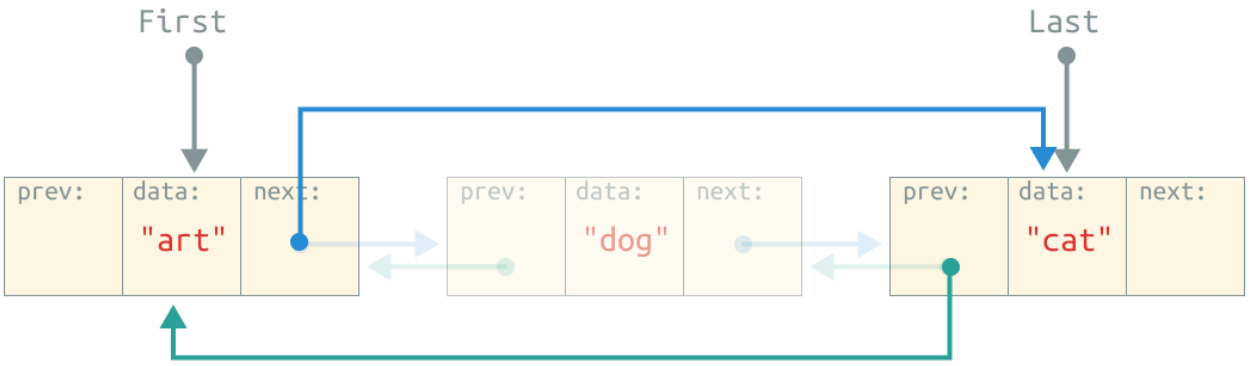


Figure 13. Remove the middle node

In the illustration, we are removing the middle node “dog” by making art’s `next` variable to point to cat and cat’s `previous` to be “art,” totally bypassing “dog.”

Let’s implement it:

Linked List’s remove from the middle of the list

```
/**
 * Removes the element at the given position (index) in this list.
 * Runtime: O(n)
 * @param {any} position
 * @returns {any} the element's value at the specified position that was
removed.
 */
removeByPosition(position = 0) {
  if (position === 0) return this.removeFirst();
  if (position === this.size - 1) return this.removeLast();
  const current = this.findBy({ index: position }).node;
  if (!current) return null;
  current.previous.next = current.next;
  current.next.previous = current.previous;
  this.size -= 1;
  return current && current.value;
}
```

Notice that we are using the `get` method to get the node at the current position. That method loops through the list until it found the node at the specified location. This iteration has a runtime of *O(n)*.

2.4.9. Linked List vs. Array

Arrays give you instant access to data anywhere in the collection using an index. However, Linked List visits nodes in sequential order. In the worst-case scenario, it takes *O(n)* to get an element from a Linked List. You might be wondering: Isn’t an array always more efficient with *O(1)* access time? It depends.

We also have to understand the space complexity to see the trade-offs between arrays and linked lists. An array pre-allocates contiguous blocks of memory. If the array fillup, it has to create a larger array (usually 2x) and copy all the elements when it is getting full. That takes *O(n)* to copy all the items over. On the other hand, LinkedList’s nodes only reserve precisely the amount of memory they need. They don’t have to be next to each other in RAM, nor are large chunks of memory is booked beforehand like arrays. Linked List is more on a "grow as you go" basis. **Linked list wins on memory usage over an array.**

Another difference is that adding/deleting at the beginning of an array takes `O(n)`; however, the linked list is a constant operation `O(1)` as we will implement later. **Linked List has better runtime**

than an array for inserting items at the beginning.

Table 9. Big O cheat sheet for Linked List and Array

Data Structure	Searching By		Inserting at the			Deleting from			Space
	Index/Key	*Value*	*beginning*	*middle*	*end*	*beginning*	*middle*	*end*	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)
Linked List (singly)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	**O(n)**	O(n)
Linked List (doubly)	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(n)	**O(1)**	O(n)

If you compare the singly linked list vs. doubly linked list, you will notice that the main difference is inserting elements to and deleting elements from the end. For a singly linked list, it's **O(n)**, while a doubly-linked list is **O(1)**.

Comparing an array with a doubly-linked list, both have different use cases:

Use arrays when:

- You want to access **random** elements by numeric key or index in constant time O(1).
- You need two-dimensional and multi-dimensional arrays.

Use a doubly linked list when:

- You want to access elements in a **sequential** manner only like Stack or Queue.
- You want to insert elements at the start and end of the list. The linked list has O(1) while the array has O(n).
- You want to save some memory when dealing with possibly large data sets. Arrays pre-allocate a large chunk of contiguous memory on initialization. Lists are more "grow as you go."

For the next two linear data structures Stack and Queue, we are going to use a doubly-linked list to implement them. We could use an array as well, but since inserting/deleting from the start performs better with linked-lists, we will use that.

2.4.10. Linked List patterns for Interview Questions

Most linked list problems are solved using 1 to 3 pointers. Sometimes we move them in tandem or individually.

Examples of problems that can be solved using multiple pointers:

- Detecting if the linked list is circular (has a loop).
- Finding the middle node of a linked list in 1-pass without any auxiliary data structure.
- Reversing the linked list in 1-pass without any auxiliary data structure. e.g. `1->2->3` to `3->2->1`.

Let's do some examples!

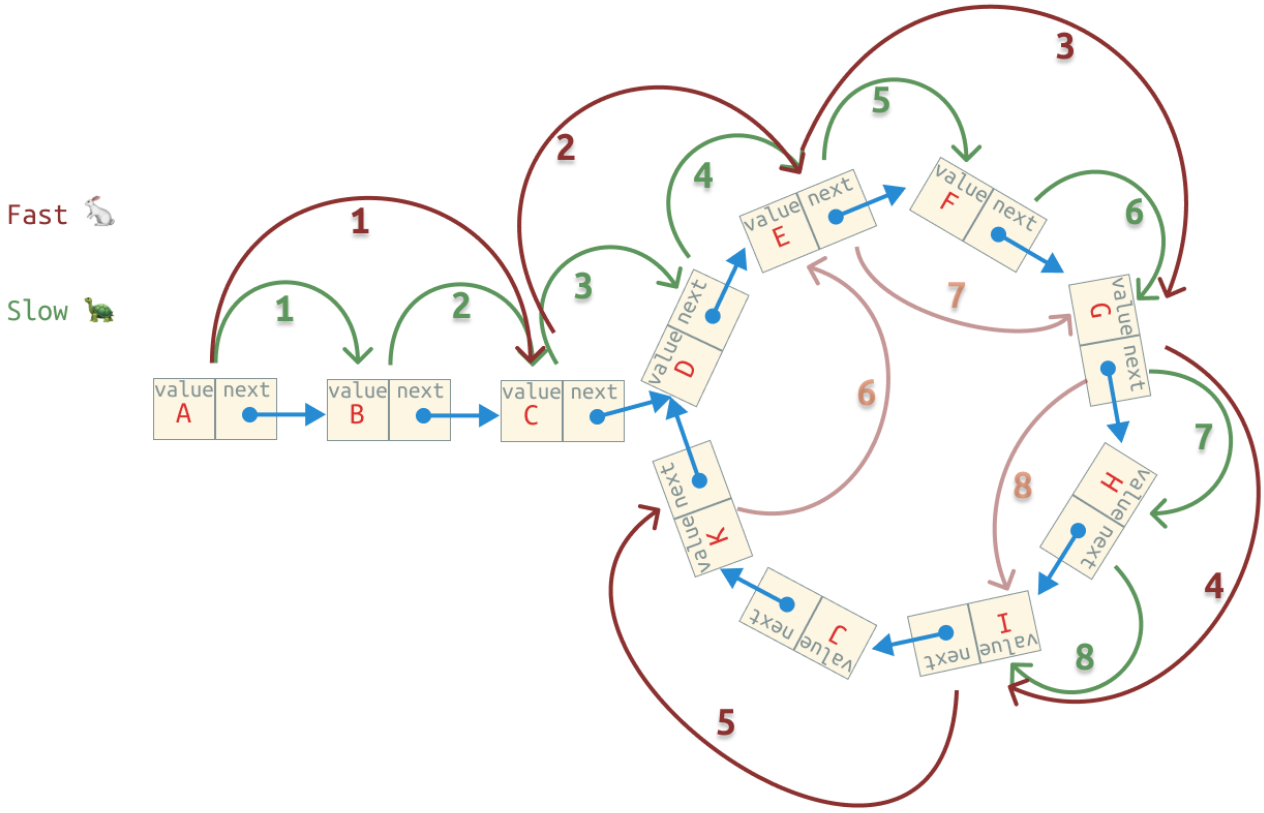
Fast/Slow Pointers

One standard algorithm to detect loops in a linked list is fast/slow runner pointers (a.k.a The Tortoise 🐢 and the Hare 🐰 or Floyd's Algorithm). The slow pointer moves one node per iteration, while the fast pointer moves two nodes every time. You can see an example code below:

Fast/Slow pointers

```
let fast = head, slow = head;
while (fast && fast.next) {
  slow = slow.next; // slow moves 1 by 1.
  fast = fast.next.next; // slow moves 2 by 2.
}
```

If the list has a loop, then at some point, both pointers will point to the same node. Take a look at the following image; take notice that both points to `node I` on the 8th iteration.

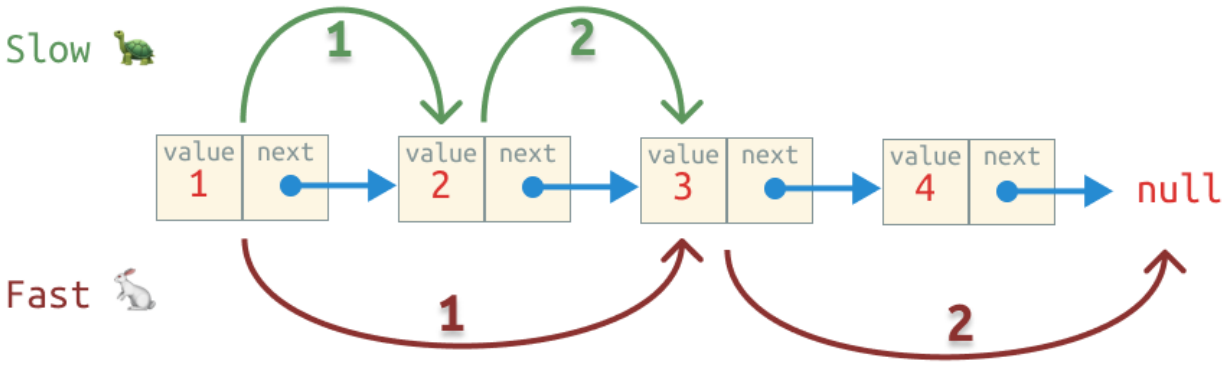


You can detect the intersection point (`node D` on the example) by using this algorithm:

- When `fast` and `slow` are the same, then create a (3rd) new pointer from the start.
- Keep moving the 3rd pointer and the `slow` simultaneously one by one.
- Where slow and 3rd pointer meets, that's the beginning of the loop or intersection (e.g., `node D`).

Fast/slow pointer has essential properties, even if the list doesn't have a loop!

If you don't have a loop, then fast and slow will never meet. However, by the time the `fast` pointer reaches the end, the `slow` pointer would be precisely in the middle!



This technique is useful for getting the middle element of a singly list in one pass without using any auxiliary data structure (like array or map).

LL-A) *Find out if a linked list has a cycle and, if so, return the intersection node (where the cycle begins).*

Signature

```
/**
 * Find the node where the cycle begins or null.
 * @param {Node} head
 * @returns {Node|null}
 */
function findCycleStart(head) {

};
```

Examples

```
findCycleStart(1 -> 2 -> 3); // null // no loops
findCycleStart(1 -> 2 -> 3 -> *1); // 1 // node 3 loops back to 1
findCycleStart(1 -> 2 -> 3 -> *2); // 2 // node 3 loops back to 2
```

Solution

One solution is to find a loop using a HashMap (`Map`) or HashSet (`Set`) to track the visited nodes. If we found a node that is already on `Set`, then that's where the loop starts.

Solution 1: Map/Set for detecting loop

```
function findCycleStartBrute(head) {
  const visited = new Set();
  let curr = head;
  while (curr) {
    if (visited.has(curr)) return curr;
    visited.add(curr);
    curr = curr.next;
  }
  return null;
}
```

Complexity Analysis

- Time Complexity: `O(n)`. We might visit all nodes on the list (e.g., no loops).
- Space complexity: `O(n)`. In the worst-case (no loop), we store all the nodes on the Set.

Can we improve anything here? We can solve this problem without using any auxiliary data structure using the fast/slow pointer.

Solution 2: Fast/Slow pointer

```
/**
 * Find where the cycle starts or null if no loop.
 * @param {Node} head - The head of the list
 * @returns {Node|null}
 */
function findCycleStart(head) {
  let slow = head;
  let fast = head;
  while (fast && fast.next) {
    slow = slow.next; // slow moves 1 by 1.
    fast = fast.next.next; // slow moves 2 by 2.
    if (fast === slow) { // detects loop!
      slow = head; // reset pointer to begining.
      while (slow !== fast) { // find intersection
        slow = slow.next;
        fast = fast.next; // move both pointers one by one this time.
      }
      return slow; // return where the loop starts
    }
  }
  return null; // not found.
}
```

Complexity Analysis

- Time Complexity: `O(n)`. In the worst case (no loop), we visit every node.
- Space complexity: `O(1)`. We didn't use any auxiliary data structure.

Multiple Pointers

LL-B) *Determine if a singly linked list is a palindrome. A palindrome is a sequence that reads the same backward as forward.*

Signature

```
/**
  class Node {
    constructor(value = null) {
      this.value = value;
      this.next = null;
    }
  }
 */

/**
 * Determine if a list is a palindrome
 * @param {Node} head
 * @returns {boolean}
 */
function isPalindrome(head) {
  // you code goes here!
}
```

Examples

```
const toList = (arr) => new LinkedList(arr).first;
isPalindrome(toList([1, 2, 3])); // false
isPalindrome(toList([1, 2, 3, 2, 1])); // true
isPalindrome(toList([1, 1, 2, 1])); // false
isPalindrome(toList([1, 2, 2, 1])); // true
```

Solution

To solve this problem, we have to check if the first and last node has the same value. Then we check if the second node and second last are the same, and so on. If we found any that's not equal; then it's not a palindrome. We can use two pointers, one at the start and the other at the end, and move them until they meet in the middle.

The issue is that with a singly linked list, we can't move backward! We could either convert it into a doubly-linked list (with the last pointer) or copy the nodes into an array. Let's do the latter as a first

approach.

Solution 1: List to array

```
function isPalindromeBrute(head) {
  const arr = [];
  for (let i = head; i; i = i.next) arr.push(i.value); ①
  let lo = 0;
  let hi = arr.length - 1;
  while (lo < hi) if (arr[lo++] !== arr[hi--]) return false; ②
  return true;
}
```

① Copy each one of the nodes' value into an array.

② Given two indices (`lo` and `hi`), one with the lowest index (0) and the other with the highest index (length - 1). Move both of them towards the center. If any values are not the same, then it's not a palindrome.

What's the time complexity?

Complexity Analysis

- Time Complexity: `O(n)`. We do two passes, one on the for-loop and the other in the array.
- Space complexity: `O(n)`. We are using auxiliary storage with the array O(n).

That's not bad, but can we do it without using any auxiliary data structure, O(1) space?

Here's another algorithm to solve this problem in O(1) space:

- Find the middle node of the list (using fast/slow pointers).
- Reverse the list from the middle to the end.
- Have two new pointers, one at the beginning of the list and the other at the head of the reversed list.
- If all nodes have the same value, then we have a palindrome. Otherwise, we don't.

Solution 2: Reverse half of the list

```
function isPalindrome(head) {
  let slow = head;
  let fast = head;
  while (fast) { // use slow/fast pointers to find the middle.
    slow = slow.next;
    fast = fast.next && fast.next.next;
  }

  const reverseList = (node) => { // use 3 pointers to reverse a linked
list
    let prev = null;
    let curr = node;
    while (curr) {
      const { next } = curr; // same as: "const next = curr.next;"
      curr.next = prev;
      prev = curr;
      curr = next;
    }
    return prev;
  };

  const reversed = reverseList(slow); // head of the reversed half
  for (let i = reversed, j = head; i; i = i.next, j = j.next) if (i
.value !== j.value) return false;
  return true;
}
```

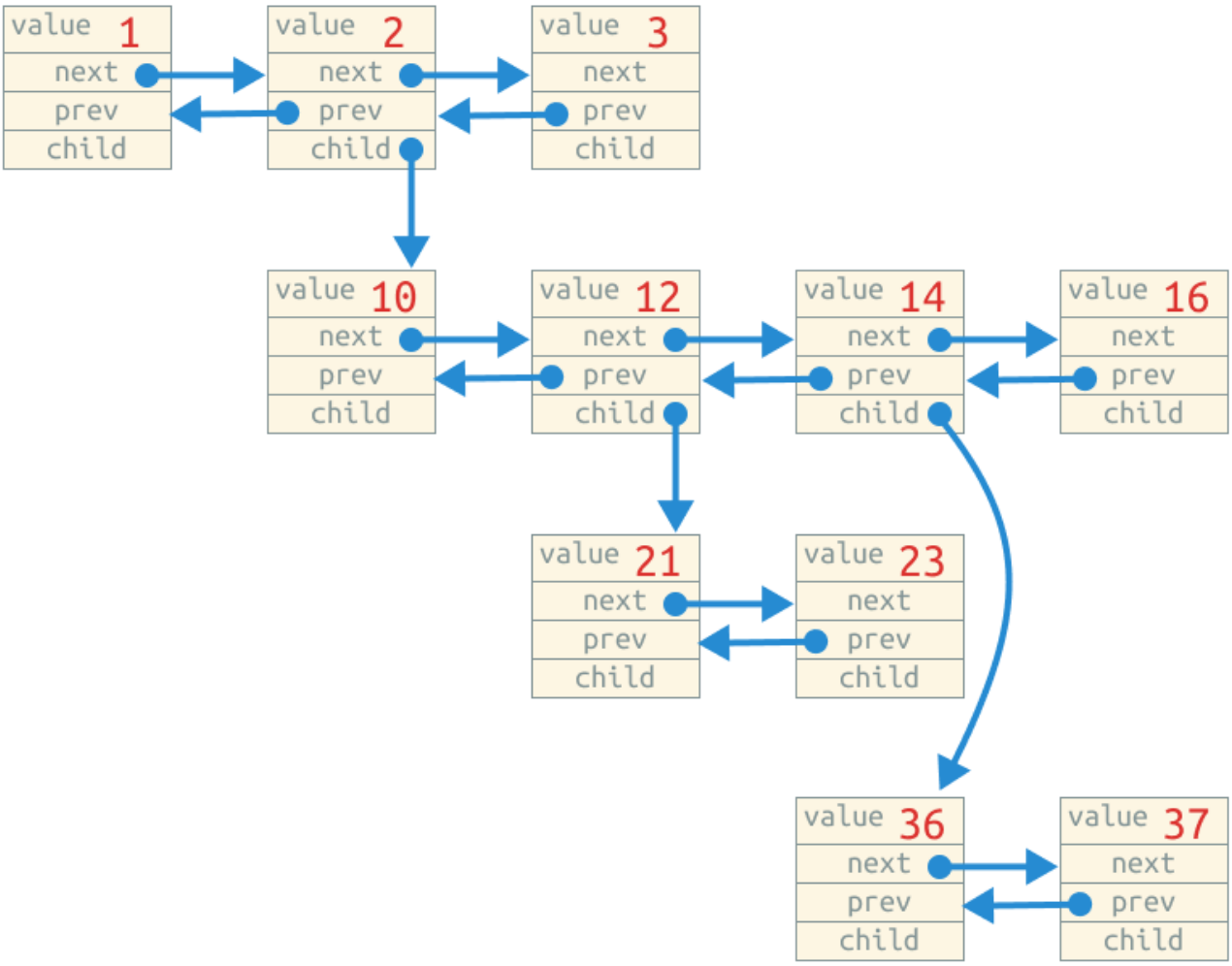
This solution is a little longer, but it's more space-efficient since it doesn't use any auxiliary data structure to hold the nodes.

Complexity Analysis

- Time Complexity: `O(n)`. We visit every node once.
- Space complexity: `O(1)`. We didn't use any auxiliary data structure. We changed data in-place.

Multi-level Linked Lists

It's good to know that linked lists might have other connections besides the `next` and `previous` pointers. They might also reference different lists forming a multi-leveled linked list.



Let's explore the following example:

LL-C) *Flatten a multi-level to a single level. You will be operating a doubly-linked list that, besides the pointers* `next` *and* `previous`*, also has a* `child` *pointer. Return the head of the flattened list.*

Signature

```
/**
 * Flatten a multi-level list
 * @param {Node} head
 * @return {Node}
 */
function flatten(head) {

}
```

Examples

```
class Node {
  value = null;
  next = null;
  prev = null;
  child = null;
  constructor(value) { this.value = value; }
}

const ll = (nums) => Array.from(new LinkedList(nums, Node));
const l1 = ll([1, 2, 3, 4, 5, 6]);
const l2 = ll([10, 12, 14, 16]);
const l3 = ll([21, 23]);
const l4 = ll([36, 37]);
l1[2].child = l2;
l2[1].child = l3;
l2[2].child = l4;
const head = l1[0];

// Head:
//
// 1--- 2--- 3--- 4--- 5--- 6
//      |
//     10---12---14---16
//           |    |
//           |   36---37
//           |
//          21--23

flatten(head); // 1->2->10->12->21->23->14->36->37->16->3->4->5->6
```

Our job is to flatten a multi-level LinkedList. So far, we know how to navigate a list using the `next` pointer. If we found another list on the `child` pointer, we can flatten it out by moving the child's chain to the `next` pointer. However, if we are not careful, we will lose whatever nodes were on `next`. One idea is to store that in an array and bring them back at a later time.

Algorithm summary:

- Starting from the `head`, visit all nodes using the `next` pointer.
  - If any node has a `child` pointer, move it to the `next`.
  - Save `next` on the array (stack) for later use.
  - When we don't have more nodes on `next`, pop from the array (stack).

Solution 1: Array/Stack approach

```
function flattenBrute(head) {
  const stack = [];
  for (let curr = head; curr; curr = curr.next) {
    if (!curr.next && stack.length) {
      curr.next = stack.pop(); // merge main thread with saved nodes.
      curr.next.previous = curr;
    }
    if (!curr.child) continue;
    if (curr.next) stack.push(curr.next); // save "next" nodes.
    curr.next = curr.child; // override next pointer with "child"
    curr.child.previous = curr;
    curr.child = null; // clear child pointer (was moved to "next").
  }

  return head;
}
```

Complexity Analysis

- Time Complexity: `O(n)`. We visit every node only once.
- Space complexity: `O(n)`. The stack array might hold almost all nodes.

This approach works well. However, we can do better in terms of space complexity. Instead of holding the data on an auxiliary array, we can append it to the end of the child's list.

Algorithm summary:

- Starting from the `head`, visit all nodes using the `next` pointer.
  - If node `curr` has a `child`.
    - Follow the child's chain to the end.
    - Then connect the child's tail to `curr.next`. By doing this, we merged the child's chain with the main thread.
    - Move the child's chain to `curr.next`.

Solution 2: In-place approach

```
/**
 * Flatten a multi-level to a single level
 * @param {Node} head
 * @return {Node}
 */
function flatten(head) {
  for (let curr = head; curr; curr = curr.next) {
    if (!curr.child) continue;

    let last = curr.child;
    while (last && last.next) last = last.next; // find "child"'s last
    if (curr.next) { // move "next" to "child"'s last postion
      last.next = curr.next;
      curr.next.previous = last;
    }
    curr.next = curr.child; // override "next" with "child".
    curr.child.previous = curr;
    curr.child = null; // clean "child" pointer.
  }

  return head;
}
```

Complexity Analysis

- Time Complexity: `O(n)`. In the worst-case, we will visit most nodes twice `2n` → `O(n)`.
- Space complexity: `O(1)`. No auxiliary structure was used to hold the lists.

2.4.11. Practice Questions

Merge Linked Lists into One

LL-1) *Merge two sorted lists into one (and keep them sorted)*

Examples:

```
mergeTwoLists(2->3->4, 1->2); // 1->2->2->3->4
mergeTwoLists(2->3->4,null); // 2->3->4
```

Common in interviews at: FAANG, Adobe, Microsoft

```
/**
 * Given two sorted linked lists merge them while keeping the asc order.
 * @examples
 *    mergeTwoLists([2,4,6], [1,3]); // => [1,2,3,4,6]
 *    mergeTwoLists([2,4,6], []); // => [2,4,6]
 *    mergeTwoLists([], [1,3]); // => [1,3]
 *
 * @param {ListNode} l1 - The root node of list 1
 * @param {ListNode} l2 - The root node of list 2
 * @returns {ListNode} - The root of the merged list.
 */
function mergeTwoLists(l1, l2) {
  // write you code here
}
```

Solution: Merge Linked Lists into One

Check if two strings lists are the same

LL-2) *Given two linked lists with strings, check if the data is equivalent.*

Examples:

```
hasSameData(he->ll->o, hel->lo); // true
hasSameData(hello, hel->lo); // true
hasSameData(he->ll->o, h->i); // false
```

Common in interviews at: Facebook

```
/**
 * Check if two lists has the same string data.
 * Note: each lists can be huge, they have up to 10 million nodes.
 *
 * @examples
 *    hasSameData(['he', 'll', 'o'], ['hel', 'lo']); // true
 *    hasSameData(['hel', 'lo'], ['hi']); // false
 *
 * @param {ListNode} l1 - The root node of list 1.
 * @param {ListNode} l2 - The root node of list 2.
 * @returns {boolean} - true if has same data, false otherwise.
 */
function hasSameData(l1, l2) {
  // write you code here
}
```

Solution: Check if two strings lists are the same

2.5. Stack

The stack is a data structure that restricts the way you add and remove data. It only allows you to insert and retrieve in a **Last-In-First-Out** (LIFO) fashion.

An analogy is to think that the stack is a rod, and the data are discs. You can only take out the last one you put in.

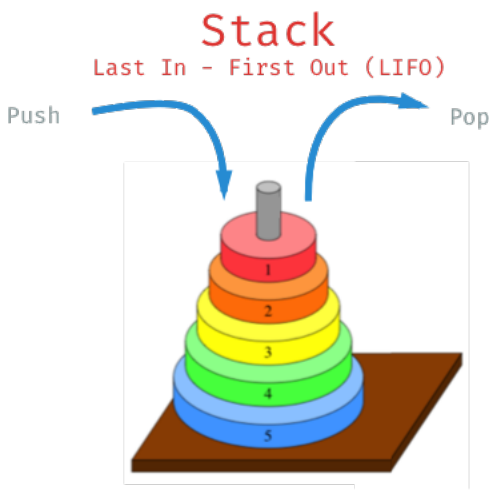


Figure 14. Stack data structure is like a stack of disks: the last element in is the first element out

As you can see in the image above, If you insert the disks in the order 5, 4, 3, 2, 1, then you can remove them in 1, 2, 3, 4, 5.

The stack inserts items to the end of the collection and also removes it from the rear. Both an array and linked list would do it in constant time. However, since we don't need the Array's random access, a linked list makes more sense.

Stack's constructor

```
/**
 * Data structure that adds and remove elements in a first-in, first-out
(FIFO) fashion
 */
class Stack {
  constructor() {
    this.items = new LinkedList();
  }
  // ... methods goes here ...
}
```

As you can see in the stack constructor, we use a linked list as the underlying data structure.

Let's now develop the insert and remove operations in a stack.

2.5.1. Insertion

We can insert into a stack using the linked list's `addLast` method.

Stack's add

```
/**
 * Add element into the stack. Similar to Array.push
 * Runtime: O(1)
 * @param {any} item
 * @returns {stack} instance to allow chaining.
 */
add(item) {
  this.items.addLast(item);
  return this;
}
```

We are returning `this`, in case we want to chain multiple add commands.

2.5.2. Deletion

Deleting is straightforward, as well.

Stack's remove

```
/**
 * Remove element from the stack.
 * Similar to Array.pop
 * Runtime: O(1)
 * @returns {any} removed value.
 */
remove() {
  return this.items.removeLast();
}
```

This time we used the linked list's `removeLast` method. That's all we need for a stack implementation. Check out the full implementation.

2.5.3. Implementation Usage

We can use our stack implementation as follows:

Stack usage example

```
const stack = new Stack();

stack.add('a');
stack.add('b');
stack.remove(); //↪ b
stack.add('c');
stack.remove(); //↪ c
stack.remove(); //↪ a
```

As you can see, if we add new items, they will be the first to go out to honor LIFO.

2.5.4. Stack Complexity

Implementing the stack with an array and linked list would lead to the same time complexity:

Table 10. Time/Space complexity for the stack operations

Data Structure	Searching By		Inserting at the			Deleting from			Space
	Index/Key	*Value*	*beginning*	*middle*	*end*	*beginning*	*middle*	*end*	
Stack	-	-	-	-	O(1)	-	-	O(1)	O(n)

It's not very common to search for values on a stack (other Data Structures are better suited for this). Stacks are especially useful for implementing Depth-First Search.

2.5.5. Practice Questions

Validate Parentheses / Braces / Brackets

ST-1) *Given a string with three types of brackets: `()`, `{}`, and `[]`. Validate they are correctly closed and opened.*

Examples:

```
isParenthesesValid('(){}[]'); // true
isParenthesesValid('('); // false (closing parentheses is missing)
isParenthesesValid('([{}])'); // true
isParenthesesValid('[{]}'); // false (brakets are not closed in the
right order)
isParenthesesValid('([{)}]'); // false (closing is out of order)
```

Common in interviews at: Amazon, Bloomberg, Facebook, Citadel

```
/**
 * Validate if the parentheses are opened and closed in the right order.
 *
 * @examples
 *  isParenthesesValid('(){}[]'); // true
 *  isParenthesesValid('([{}])'); // true
 *  isParenthesesValid('([{)}]'); // false
 *
 * @param {string} string - The string
 * @returns {boolean} - True if valid, false otherwise.
 */
function isParenthesesValid(string) {
  // write you code here
}
```

Solution: Validate Parentheses / Braces / Brackets

Daily Temperaturs

ST-2) *Given an array of integers from 30 to 100 (daily temperatures), return another array that, for each day in the input, tells you how many days you would have to wait until warmer weather. If no warmer climate is possible, then return* `0` *for that element.*

Examples:

```
dailyTemperatures([30, 28, 50, 40, 30]); // [2 (to 50), 1 (to 28), 0, 0,
0]
dailyTemperatures([73, 69, 72, 76, 73, 100]); // [3, 1, 1, 0, 1, 100]
```

Common in interviews at: Amazon, Adobe, Cisco

```
/**
 * Given an array with daily temperatures (30 °C to 100 °C),
 * return an array with the days count until a warmer temperature
 * for each elem from the input.
 *
 * @examples
 *  dailyTemperatures([30, 28, 50, 40, 30]); // [2, 1, 0, 0, 0]
 *  dailyTemperatures([73, 69, 72, 76, 73]); // [3, 1, 1, 0, 0]
 *
 * @param {number[]} t - Daily temperatures
 * @returns {number[]} - Array with count of days to warmer temp.
 */
function dailyTemperatures(t) {
  // write you code here
}
```

Solution: Daily Temperaturs

2.6. Queue

A queue is a linear data structure where the data flows in a **First-In-First-Out** (FIFO) manner.

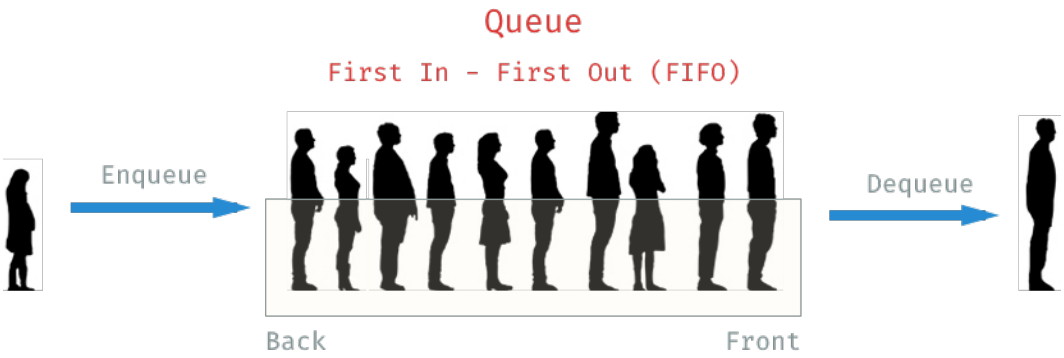


Figure 15. Queue data structure is like a line of people: the First-in, is the First-out

A queue is like a line of people at the bank; the person who arrived first is the first to go out.

Similar to the stack, we only have two operations (insert and remove). In a Queue, we add elements to the back of the list and remove it from the front.

We could use an array or a linked list to implement a Queue. However, it is recommended only to use a linked list. Why? An array has a linear runtime *O(n)* to remove an element from the start, while a linked list has constant time *O(1)*.

Queue's constructor

```
/**
 * Data structure where we add and remove elements in a first-in, first-
out (FIFO) fashion
 */
class Queue {
  constructor(iterable = []) {
    this.items = new LinkedList(iterable);
  }
  // ... methods go here ...
}
```

We initialize the Queue creating a linked list. Now, let's add the `enqueue` and `dequeue` methods.

2.6.1. Insertion

For inserting elements into a queue, also know as **enqueue**, we add items to the back of the list using `addLast`:

Queue's enqueue

```
/**
 * Add element to the back of the queue.
 * Runtime: O(1)
 * @param {any} item
 * @returns {queue} instance to allow chaining.
 */
enqueue(item) {
  this.items.addLast(item);
  return this;
}
```

As discussed, this operation has a constant runtime.

2.6.2. Deletion

For removing elements from a queue, also known as **dequeue**, we remove elements from the front of the list using `removeFirst`:

Queue's dequeue

```
/**
 * Remove element from the front of the queue.
 * Runtime: O(1)
 * @returns {any} removed value.
 */
dequeue() {
  return this.items.removeFirst();
}
```

As discussed, this operation has a constant runtime.

2.6.3. Implementation usage

We can use our Queue class as follows:

Queue usage example

```
const queue = new Queue();

queue.enqueue('a');
queue.enqueue('b');
queue.dequeue(); //↪ a
queue.enqueue('c');
queue.dequeue(); //↪ b
queue.dequeue(); //↪ c
```

You can see that the items are dequeued in the same order they were added, FIFO (first-in, first-out).

2.6.4. Queue Complexity

As an experiment, we can see in the following table that if we had implemented the Queue using an array, its enqueue time would be *O(n)* instead of *O(1)*. Check it out:

Table 11. Time/Space complexity for queue operations

Data Structure	Searching By		Inserting at the			Deleting from			Space
	Index/Key	*Value*	*beginning*	*middle*	*end*	*beginning*	*middle*	*end*	
Queue (w/array)	-	-	-	-	O(1)	**O(n)**	-	-	O(n)
Queue (w/list)	-	-	-	-	O(1)	O(1)	-	-	O(n)

2.6.5. Practice Questions

Recent Counter

QU-1) *Design a class that counts the most recent requests within a time window.*

Example:

```
const counter = new RecentCounter(10); // The time window is 10 ms.
counter.request(1000); // 1 (first request, it always counts)
counter.request(3000); // 1 (last requests was 2000 ms ago, > 10ms, so
doesn't count)
counter.request(3100); // 1 (last requests was 100 ms ago, > 10ms, so
doesn't count)
counter.request(3105); // 2 (last requests was 5 ms ago, <= 10ms, so it
counts)
```

Common in interviews at: FAANG, Bloomberg, Yandex

```
/**
 * Counts the most recent requests within a time window.
 * Each request has its timestamp.
 * If the time window is 2 seconds,
 * any requests that happened more than 2 seconds before the most
 * recent request should not count.
 *
 * @example - The time window is 3 sec. (3000 ms)
 *  const counter = new RecentCounter(3000);
 *  counter.request(100); // 1
 *  counter.request(1000); // 2
 *  counter.request(3000); // 3
 *  counter.request(3100); // 4
 *  counter.request(3101); // 4
 *
 */
class RecentCounter {
  /**
   * @param {number} maxWindow - Max. time window (in ms) for counting
requests
   *  Defaults to 1 second (1000 ms)
   */
  constructor(maxWindow = 1000) {
  }

  /**
   * Add new request and calculate the current count within the window.
   * @param {number} timestamp - The current timestamp (increasing
order)
   * @return {number} - The number of requests within the time window.
   */
  request(timestamp) {
  }
}
  // write you code here
}
```

Solution: Recent Counter

Design Snake Game

QU-2) *Design the* `move` *function for the snake game. The move function returns an integer representing the current score. If the snake goes out of the given height and width or hit itself, the game is over and return* `-1`.

Example:

```
const snakeGame = new SnakeGame(4, 2, [[1, 2], [0, 1]]);
expect(snakeGame.move('R')).toEqual(0); //  0
expect(snakeGame.move('D')).toEqual(0); //  0
expect(snakeGame.move('R')).toEqual(1); //  1 (ate food1)
expect(snakeGame.move('U')).toEqual(1); //  1
expect(snakeGame.move('L')).toEqual(2); //  2 (ate food2)
expect(snakeGame.move('U')).toEqual(-1); // -1 (hit wall)
```

Common in interviews at: Amazon, Bloomberg, Apple

```
/**
 * The snake game stars with a snake of length 1 at postion 0,0.
 * Only one food position is shown at a time. Once it's eaten the next
one shows up.
 * The snake can move in four directions up, down, left and right.
 * If the snake go out of the boundaries (width x height) the game is
over.
 * If the snake hit itself the game is over.
 * When the game is over, the `move` method returns -1 otherwise, return
the current score.
 *
 * @example
 *  const snakeGame = new SnakeGame(3, 2, [[1, 2], [0, 1]]);
 *  snakeGame.move('R'); //  0
 *  snakeGame.move('D'); //  0
 *  snakeGame.move('R'); //  0
 *  snakeGame.move('U'); //  1
 *  snakeGame.move('L'); //  2
 *  snakeGame.move('U'); // -1
 */
class SnakeGame {
  /**
   * Initialize game with grid's dimension and food order.
   * @param {number} width - The screen width (grid's columns)
   * @param {number} height - Screen height (grid's rows)
   * @param {number[]} food - Food locations.
   */
  constructor(width, height, food) {
  }
  /**
   * Move snake 1 position into the given direction.
   * It returns the score or game over (-1) if the snake go out of bound
or hit itself.
   * @param {string} direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D'
= Down.
   * @returns {number} - The current score (snake.length - 1).
   */
  move(direction) {
  }
}
```

Solution: *Design Snake Game*

2.7. Array vs. Linked List & Queue vs. Stack

In this part of the book, we explored the most used linear data structures such as Arrays, Linked Lists, Stacks, and Queues. We implemented them and discussed the runtime of their operations.

Use Arrays when...

- You need to access data in random order fast (using an index).
- Your data is multi-dimensional (e.g., matrix, tensor).

Use Linked Lists when:

- You will access your data sequentially.
- You want to save memory and only allocate memory as you need it.
- You want constant time to remove/add from extremes of the list.

Use a Queue when:

- You need to access your data on a first-come, first-served basis (FIFO).
- You need to implement a Breadth-First Search

Use a Stack when:

- You need to access your data as last-in, first-out (LIFO).
- You need to implement a Depth-First Search

Table 12. Time/Space Complexity of Linear Data Structures (Array, LinkedList, Stack & Queues)

Data Structure	Searching By		Inserting at the			Deleting from			Space
	Index/Key	*Value*	*beginning*	*middle*	*end*	*beginning*	*middle*	*end*	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)
Singly Linked List	O(n)	O(n)	O(1)	O(n)	**O(n)**	O(1)	O(n)	**O(n)**	O(n)
Doubly Linked List	O(n)	O(n)	O(1)	O(n)	**O(1)**	O(1)	O(n)	**O(1)**	O(n)
Stack	-	-	-	-	O(1)	-	-	O(1)	O(n)
Queue (w/array)	-	-	-	-	O(1)	**O(n)**	-	-	O(n)
Queue (w/list)	-	-	-	-	O(1)	**O(1)**	-	-	O(n)

PART THREE

3. Graph & Tree Data Structures

Graph-based data structures are everywhere, whether you realize it or not. You can find them in databases, Web (HTML DOM tree), search algorithms, finding the best route to get home, and many more uses. We are going to learn the basic concepts and when to choose one over the other.

In this chapter we are going to learn:

- Exciting Graph data structure applications
- Searching efficiently with a Tree data structures.
- One of the most versatile data structure of all Map.
- Keeping duplicates out with a Tree Set. By the end of this section, you will know the data structures trade-offs and when to use one over the other.

3.1. Tree

A tree is a non-linear data structure where a node can have zero or more connections. The topmost node in a tree is called **root**. The linked nodes to the root are called **children** or **descendants**.

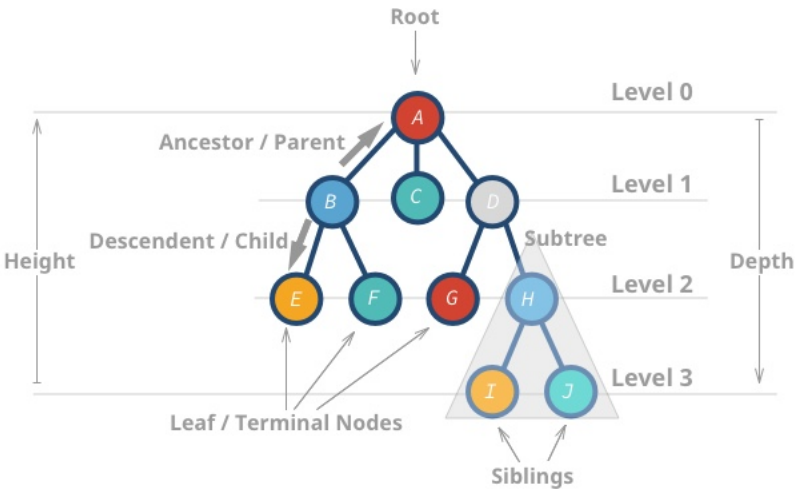


Figure 16. Tree Data Structure: root node and descendants.

As you can see in the picture above, this data structure resembles an inverted tree, hence the name. It starts with a **root** node and **branch** off with its descendants, and finally **leaves**.

3.1.1. Implementing a Tree

Implementing a tree is not too hard. It's similar to a Linked List. The main difference is that instead of having the `next` and `previous` links, we have an 0 or more number of linked nodes (children/descendants).

Tree's node constructor

```
/**
 * TreeNode - each node can have zero or more children
 */
class TreeNode {
  constructor(value) {
    this.value = value;
    this.descendants = [];
  }
}
```

Simple! Right? But there are some constraints that you have to keep at all times.

Tree data structures constraints

1. **Loops**: You have to be careful **not** to make a circular loop. Otherwise, this wouldn't be a tree anymore but a graph data structure! E.g., Node A has B as a child, then Node B list Node A as its descendant forming a loop.†
2. **Parents**: A node with more than two parents. Again, if that happens is no longer a tree but a Graph.
3. **Root**: a tree must have only one root. Two non-connected parts are not a tree. Graph can have non-connected portions and doesn't have root.

3.1.2. Basic concepts

Here's a summary of the three basic concepts:

- The topmost node is called **root**.
- A node's primary linked nodes are called **children**.
- A **leaf** or **terminal node** is a node without any descendant or children.
- A node immediate ancestor is called **parent**. Yep, and like a family tree, a node can have **uncles** and **siblings**, and **grandparents**.
- **Internal nodes** are all nodes except for the leaf nodes and the root node.
- The connection/link between nodes is called **edge**.
- The **height of a *tree*** is the distance (edge count) from the farthest leaf to the root.
- The **height of a *node*** is obtained by counting the edges between the *node* and the most distant leaf. For instance, from the image above:
  - Node A has a height of 3.
  - Node G has a height of 1.
  - Node I has a height of 0.
- The **depth of a tree** is the distance (edge count) from the root to the farthest leaf.

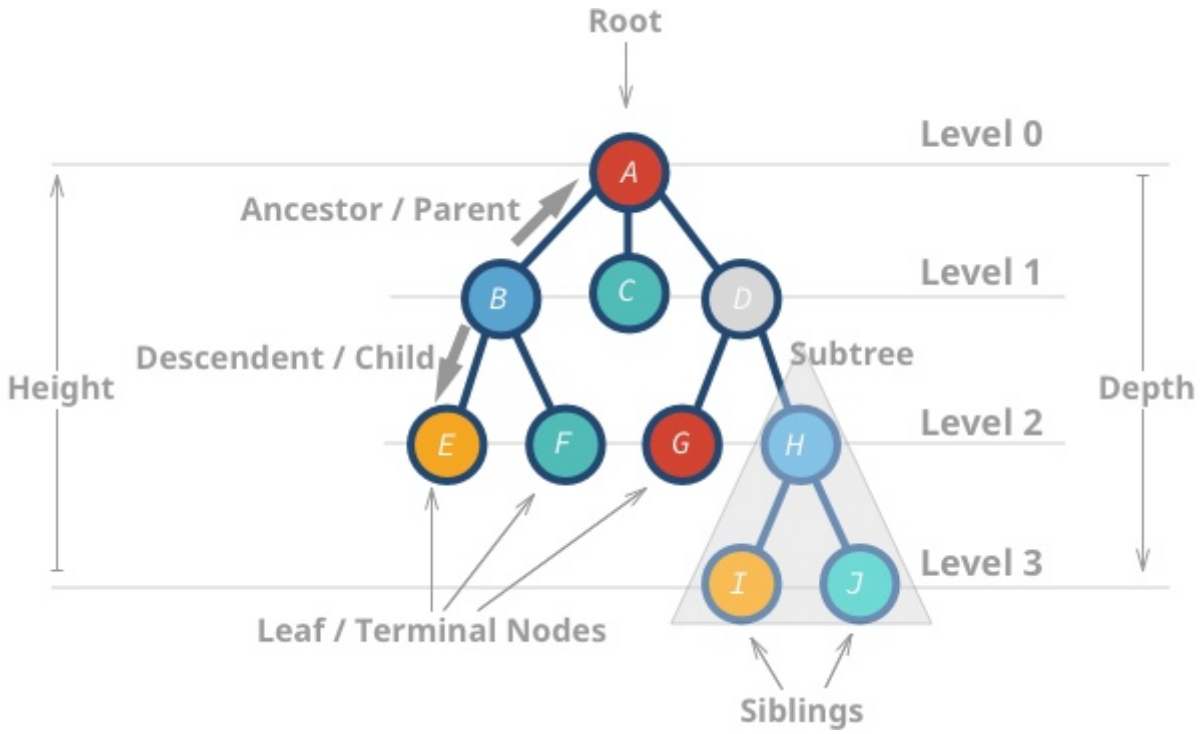


Figure 17. Tree anatomy

3.1.3. Types of Binary Trees

There are different kinds of trees, depending on the restrictions. E.g. The trees with two children or less are called **binary tree**, while trees with at most three children are **Ternary Tree**. Since binary trees are the most common, we will cover them here and others in another chapter.

Binary Tree

The binary restricts the nodes to have at most two children. Trees can have 0, 1, 2, 7, or more, but not binary trees.

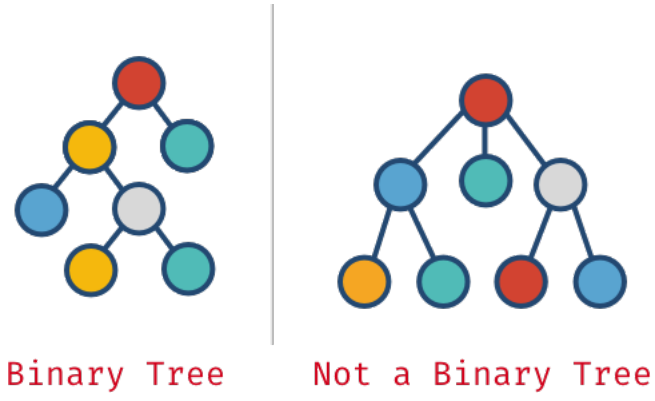


Figure 18. Binary tree has at most two children while non-binary trees can have more.

Binary trees are one of the most used kinds of trees, and they are used to build other data structures.

Binary Tree Applications

- Tree Map

- Tree Set
- Priority Queues
- Binary Search Tree (BST)

Binary Search Tree (BST)

The Binary Search Tree (BST) is a specialization of the binary tree. BST has the same restriction as a binary tree; each node has at most two children. However, there's another restriction: the values are ordered. It means the left child's value has to be less or equal than the parent. In turn, the right child's value has to be bigger than the parent.

> BST: left ≤ parent < right

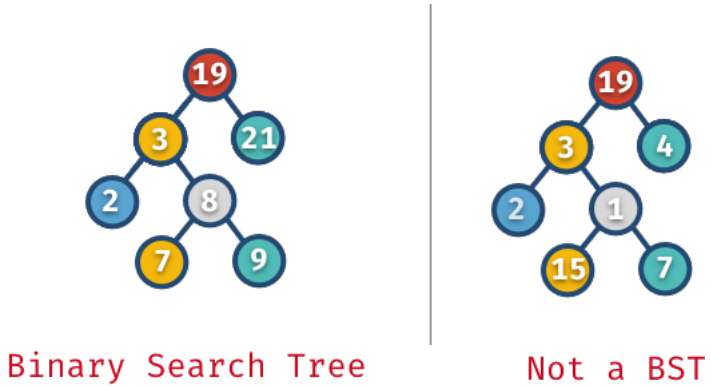


Figure 19. BST or ordered binary tree vs. non-BST.

Binary Heap

The heap (max-heap) is a binary tree where the parent's value is higher than both children's value. Opposed to the BST, the left child doesn't have to be smaller than the right child.

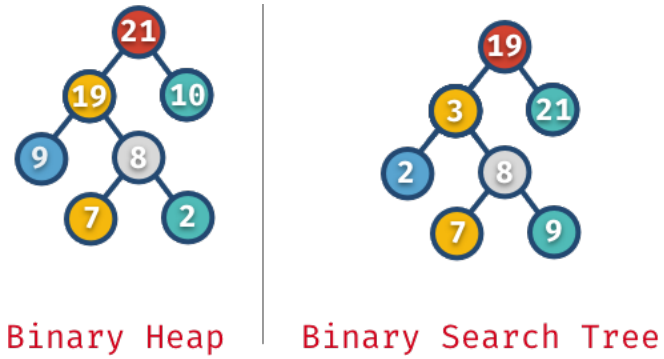


Figure 20. Heap vs BST

The (max) heap has the maximum value in the root, while BST doesn't.

There are two kinds of heaps: min-heap and max-heap. For a **max-heap**, the root has the highest value. The heap guarantee that as you move away from the root, the values get smaller. The opposite is true for a **min-heap**. In a min-heap, the lowest value is at the root, and as you go down the lower to the descendants, they will keep increasing values.

Figure 21. Max-heap keeps the highest value at the top while min-heap keep the lowest at the root.

Heap vs. Binary Search Tree

Heap is better at finding max or min values in constant time **O(1)**, while a balanced BST is good a finding any element in **O(log n)**. Heaps are often used to implement priority queues, while BST is used when you need every value sorted.

3.2. Binary Search Tree

To recap, the Binary Search Tree (BST) is a tree data structure that keeps the following constraints:

- Each node must have at most two children. Usually referred to as "left" and "right."
- All trees must have a "root" node.
- The order of nodes values must be: `left child < parent < right child`.
- Nodes might need re-ordering after each insert/delete operation to keep the `left <= parent < right` constraint.

3.2.1. Implementing a Binary Search Tree

The first step is to implement the Binary Tree Node, which can hold 0, 1, or 2 children.

Binary Tree Node's constructor

```
/**
 * Binary Tree Node
 *
 */
class BinaryTreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
    this.meta = {};
  }
}
```

Does this look familiar to you? It's almost like the linked list node, but instead of having `next` and `previous`, it has `left` and `right`. That guarantees that we have at most two children.

We also added the `meta` object to hold some metadata about the node, like duplicity, color (for red-black trees), or any other data needed for future algorithms.

We implemented the node; now, let's layout other methods that we can implement for a BST:

Binary Search Tree's class

```
class BinarySearchTree {
  constructor() {
    this.root = null;
    this.size = 0;
  }

  add(value) { /* ... */ }
  find(value) { /* ... */ }
  remove(value) { /* ... */ }
  getMax() { /* ... */ }
  getMin() { /* ... */ }
}
```

With the methods `add` and `remove`, we have to guarantee that our tree always has one root element from where we can navigate left or right based on the value we are looking for. Let's implement those `add` method first:

Inserting new elements in a BST

For inserting an element in a BST, we have two scenarios:

1. If the tree is empty (root element is null), we add the newly created node as root, and that's it!
2. If the tree has a root, compare the new value with the root. Then we have three possibilities:
   a. `root == newValue`: we increase the duplicity counter in that case, and done!
   b. `root > newValue`, we search on the left side of the root.
   c. `root < newValue`, we search on the right side of the root.
3. Repeat the comparison between the current node and `newValue`, until we find the value or (null) space.

For instance, let's say that we want to insert the values 19, 21, 10, 2, 18 in a BST:

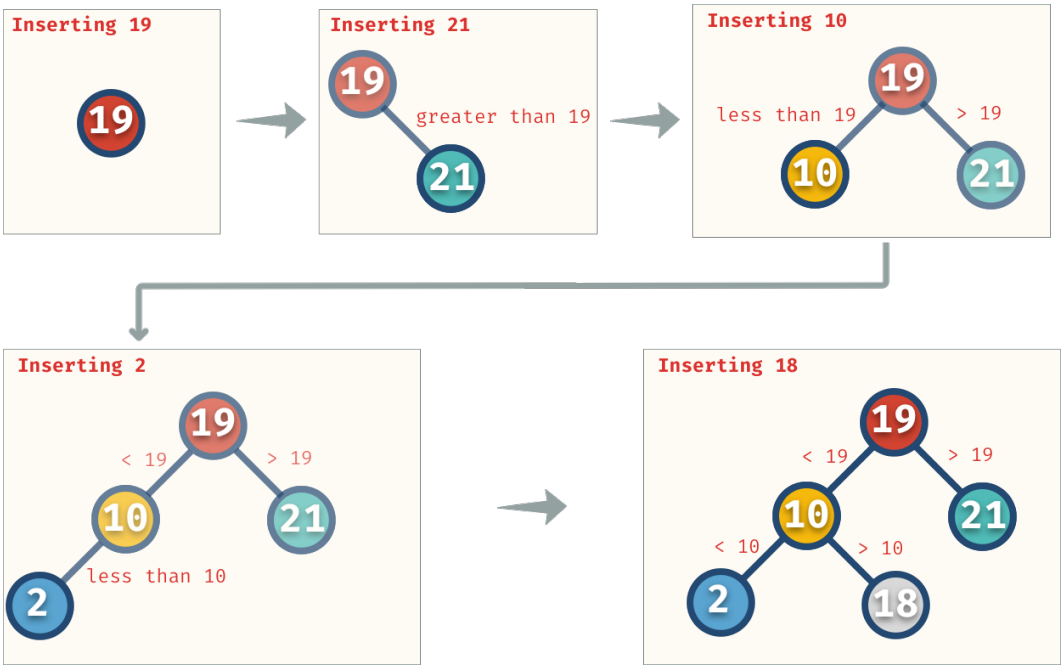


Figure 22. Inserting values on a BST.

In the last box of the image above, we start by the root when we insert node 18 (19). Since 18 is less than 19, then we move left. Node 18 is greater than 10, so we move right. There's an empty spot, and we place it there. Let's code it up:

Binary Search Tree's class

```
/**
 * Insert value on the BST.
 *
 * If the value is already in the tree,
 * then it increases the multiplicity value
 * @param {any} value node's value to insert in the tree
 * @returns {BinaryTreeNode} newly added node
 */
add(value) {
  const newNode = new BinaryTreeNode(value);

  if (this.root) {
    const { found, parent } = this.findNodeAndParent(value); ①
    if (found) { // duplicated: value already exist on the tree
      found.meta.multiplicity = (found.meta.multiplicity || 1) + 1; ②
    } else if (value < parent.value) {
      parent.setLeftAndUpdateParent(newNode);
    } else {
      parent.setRightAndUpdateParent(newNode);
    }
  } else {
    this.root = newNode;
  }

  this.size += 1;
  return newNode;
}
```

① We are using a helper function `findNodeAndParent` to iterate through the tree, finding a node with the current value "found" and its parent (implementation on the next section).

② We are taking care of duplicates. Instead of inserting duplicates, we are keeping a multiplicity tally. We have to decrease it when removing nodes.

Finding a value in a BST

We can implement the find method using the helper `findNodeAndParent` as follows:

Binary Search Tree's find methods

```
/**
 * @param {any} value value to find
 * @returns {BinaryTreeNode|null} node if it found it or null if not
 */
find(value) {
  return this.findNodeAndParent(value).found;
}


/**
 * Recursively finds the node matching the value.
 * If it doesn't find, it returns the leaf `parent` where the new value
should be appended.
 * @param {any} value Node's value to find
 * @param {BinaryTreeNode} node first element to start the search (root
is default)
 * @param {BinaryTreeNode} parent keep track of parent (usually filled
by recursion)
 * @returns {object} node and its parent like {node, parent}
 */
findNodeAndParent(value, node = this.root, parent = null) {
  if (!node || node.value === value) {
    return { found: node, parent };
  } if (value < node.value) {
    return this.findNodeAndParent(value, node.left, node);
  }
  return this.findNodeAndParent(value, node.right, node);
}
```

`findNodeAndParent` is a recursive function that goes to the left or right child, depending on the value. However, if the value already exists, it will return it in `found` variable.

Removing elements from a BST

Deleting a node from a BST has three cases.

The node is a

1. leaf
2. parent with one child
3. parent with two children/root.

Removing a leaf (Node with 0 children)

Deleting a leaf is the easiest; we look for their parent and set the child to null.

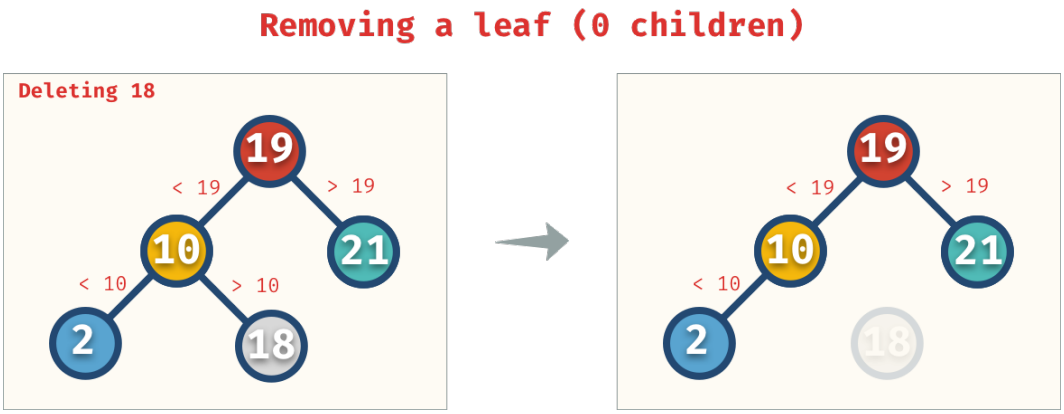


Figure 23. Removing node without children from a BST.

Node 18, will be hanging around until the garbage collector is run. However, there's no node referencing to it to no longer be reachable from the tree.

Removing a parent (Node with 1 children)

Removing a parent is not as easy since you need to find new parents for its children.

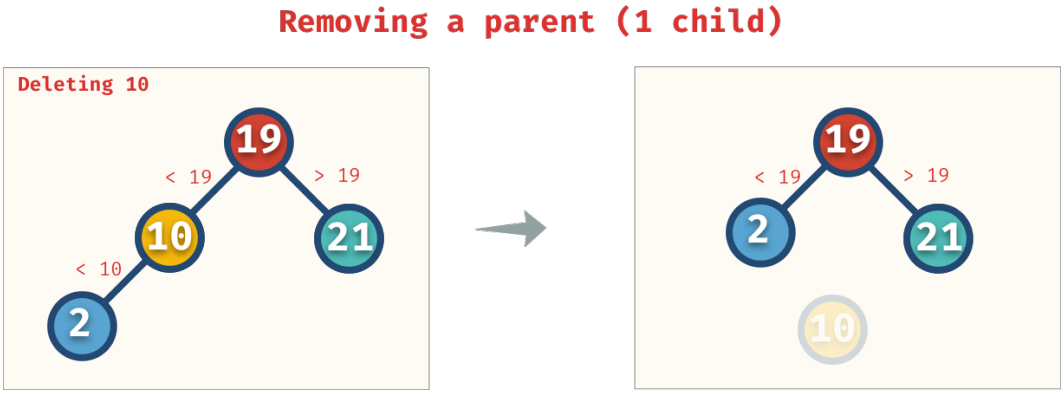


Figure 24. Removing node with 1 children from a BST.

In the example, we removed node `10` from the tree, so its child (node 2) needs a new parent. We made node 19 the new parent for node 2.

Removing a full parent (Node with 2 children) or root

Removing a parent of two children is the trickiest of all cases because we need to find new parents. (This sentence might sound tragic out of context 😂)

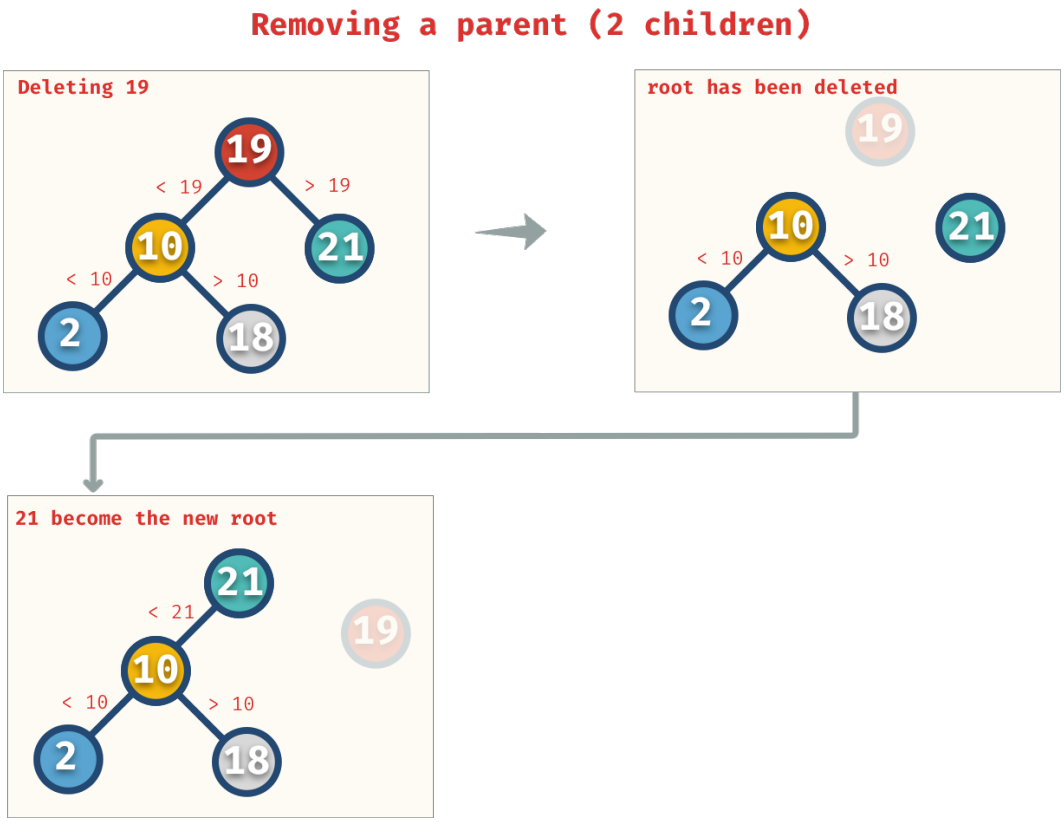


Figure 25. Removing node with two children from a BST.

In the example, we delete the root node 19. This deletion leaves two orphans (node 10 and node 21). There are no more parents because node 19 was the **root** element. One way to solve this problem is to **combine** the left subtree (Node 10 and descendants) into the right subtree (node 21). The final result is node 21 is the new root.

What would happen if node 21 had a left child (e.g., node 20)? Well, we would move node 10 and its descendants' bellow node 20.

Implementing removing elements from a BST

All the described scenarios removing nodes with zero, one and two children can be sum up on this code:

Binary Search Tree's remove method

```
/**
 * Remove a node from the tree
 * @returns {boolean} false if not found and true if it was deleted
 */
remove(value) {
  const { found: nodeToRemove, parent } = this.findNodeAndParent(value);
①

  if (!nodeToRemove) return false; ②

  // Combine left and right children into one subtree without
nodeToRemove
  const removedNodeChildren = this.combineLeftIntoRightSubtree
(nodeToRemove); ③

  if (nodeToRemove.meta.multiplicity && nodeToRemove.meta.multiplicity >
1) { ④
    nodeToRemove.meta.multiplicity -= 1; // handles duplicated
  } else if (nodeToRemove === this.root) { ⑤
    // Replace (root) node to delete with the combined subtree.
    this.root = removedNodeChildren;
    if (this.root) { this.root.parent = null; } // clearing up old
parent
  } else if (nodeToRemove.isParentLeftChild) { ⑥
    // Replace node to delete with the combined subtree.
    parent.setLeftAndUpdateParent(removedNodeChildren);
  } else {
    parent.setRightAndUpdateParent(removedNodeChildren);
  }

  this.size -= 1;
  return true;
}
```

① Try to find if the value exists on the tree.

② If the value doesn't exist, we are done!

③ Create a new subtree without the value to delete

④ Check the multiplicity (duplicates) and decrement the count if we have multiple nodes with the same value

⑤ If the `nodeToRemove` was the root, we wouldwould move the removed node's children as the new root.

⑥ If it was not the root, we will go to the deleted node's parent and put their children there.

We compute `removedNodeChildren`, which is the resulting subtree after combining the deleted node children.

The method to combine subtrees is the following:

BST's combine subtrees

```
/**
 * Combine left into right children into one subtree without given
parent node.
 *
 * @example combineLeftIntoRightSubtree(30)
 *
 *      30*                                40
 *    /     \                            /  \
 *   10      40      combined          35   50
 *     \    /  \    ---------->       /
 *     15  35   50                   10
 *                                     \
 *                                      15
 *
 * It takes node 30 left subtree (10 and 15) and put it in the
 * leftmost node of the right subtree (40, 35, 50).
 *
 * @param {BinaryTreeNode} node
 * @returns {BinaryTreeNode} combined subtree
 */
combineLeftIntoRightSubtree(node) {
  if (node.right) {
    const leftmost = this.getLeftmost(node.right);
    leftmost.setLeftAndUpdateParent(node.left);
    return node.right;
  }
  return node.left;
}
```

Take a look at the code above and the example. You will see how to remove node `30` and combine both children's subtree, and keeping the BST rules. Also, this method uses a helper to get the left-most node. We can implement it like this:

Binary Search Tree's get the leftmost node

```
/**
 * Get the node with the min value of subtree: the left-most value.
 * @param {BinaryTreeNode} node subtree's root
 * @returns {BinaryTreeNode} left-most node (min value)
 */
getLeftmost(node = this.root) {
  if (!node || !node.left) {
    return node;
  }
  return this.getMin(node.left);
}
```

That's all we need to remove elements from a BST. Check out the complete BST implementation here.

3.2.2. Differentiating a balanced and non-balanced Tree

As we insert and remove nodes from a BST, we could end up like the tree on the left:

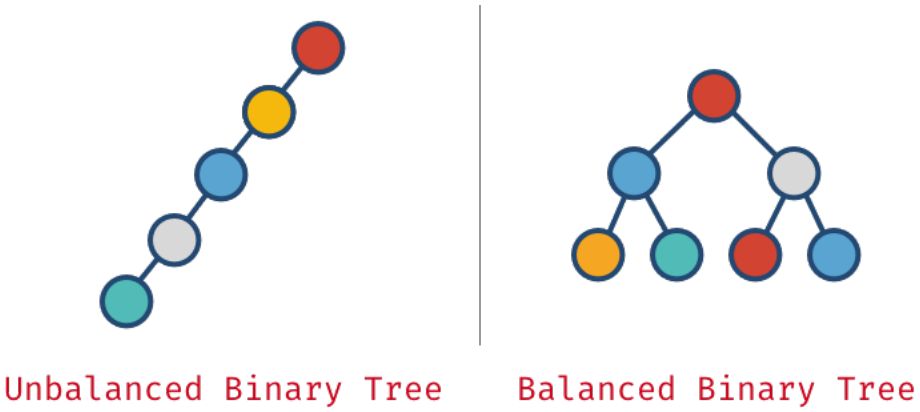


Figure 26. Balanced vs. Unbalanced Tree.

The tree on the left is unbalanced. It looks like a Linked List and has the same runtime! Searching for an element would be **O(n)**, yikes! However, on a balanced tree, the search time is **O(log n)**, which is pretty good! That's why we always want to keep the tree balanced. In further chapters, we will explore how to keep a tree balanced after each insert/delete.

3.2.3. Tree Complexity

We can sum up the tree operations using Big O notation:

Table 13. Time complexity for a Binary Search Tree (BST)

Data Structure	Searching By		Insert	Delete	Space Complexity
	Index/Key	*Value*			
BST (**un**balanced)	-	O(n)	O(n)	O(n)	O(n)
BST (balanced)	-	O(log n)	O(log n)	O(log n)	O(n)

3.3. Tree Search & Traversal

So far, we have covered how to insert/delete/search values in a binary search tree (BST). However, not all binary trees are BST, so there are other ways to look for values or visit all nodes in a particular order.

If we have the following tree:

```
          10
        /    \
       5      30
     /       /  \
    4       15   40
  /
 3
```

Depending on what traversal methods we used, we will have a different visiting order.

Tree traversal methods

- Breadth-first traversal (a.k.a level order traversal): `10, 5, 30, 4, 15, 40, 3`
- Depth-first traversal
  - In-order (left-root-right): `3, 4, 5, 10, 15, 30, 40`
  - Pre-order (root-left-right): `10, 5, 4, 3, 30, 15, 40`
  - Post-order (left-right-root): `3, 4, 5, 15, 40, 30, 10`

Why do we care? Well, there are specific problems that you can solve more optimally using one or another traversal method. For instance, to get the size of a subtree, finding maximums/minimums, and so on.

Let's cover the Breadth-first search (BFS) and the Depth-first search (DFS).

3.3.1. Breadth-First Search for Binary Tree

The breadth-first search goes wide (breadth) before going deep. Hence, the name. In other words, it goes level by level. It visits all the immediate nodes or children and then moves on to the children's children. Let's how can we implement it!

Breath-First Search (BFS) Implementation

```
/**
 * Breath-first search for a tree (always starting from the root
element).
 * @yields {BinaryTreeNode}
 */
* bfs() {
  const queue = new Queue();

  queue.add(this.root);

  while (!queue.isEmpty()) {
    const node = queue.remove();
    yield node;

    if (node.left) { queue.add(node.left); }
    if (node.right) { queue.add(node.right); }
  }
}
```

As you see, the BFS uses a Queue data structure. We enqueue all the children of the current node and then dequeue them as we visit them.

Note the asterisk (*) in front of the function means that this function is a generator that yields values.

JavaScript Generators

JavaScript generators were added as part of ES6; they allow process possibly expensive operations one by one. You can convert any function into a generator by adding the asterisk in front and `yield`ing a value.

Then you can use `next()` to get the value and also `done` to know if it's the last value. Here are some examples:

```
function* dummyIdMaker() {
  yield 0;
  yield 1;
  yield 2;
}

const generator = dummyIdMaker()

// getting values
console.log(generator.next()); // ↪ {value: 0, done: false}
console.log(generator.next()); // ↪ {value: 1, done: false}
console.log(generator.next()); // ↪ {value: 2, done: false}
console.log(generator.next()); // ↪ {value: undefined, done: true}

// iterating generator values with for..of loops
for(const n of dummyIdMaker()) {
  console.log(n);
}

// converting a generator to an array
console.log(Array.from(dummyIdMaker())); // [0, 1, 2]
```

3.3.2. Depth-First Search for Binary Tree

Depth-First search goes deep (depth) before going wide. It means that starting for the root, it goes as deep as it can until it found a leaf node (node without children), then it visits all the remaining nodes in the path.

Depth-First Search (DFS) Implementation with a Stack

```
/**
 * Depth-first search for a tree (always starting from the root element)
 * @see preOrderTraversal Similar results to the pre-order transversal.
 * @yields {BinaryTreeNode}
 */
* dfs() {
  const stack = new Stack();

  stack.add(this.root);

  while (!stack.isEmpty()) {
    const node = stack.remove();
    yield node;

    if (node.right) { stack.add(node.right); }
    if (node.left) { stack.add(node.left); }
  }
}
```

This is an iterative implementation of a DFS using an Stack. It's almost identical to the BFS, but instead of using a Queue we use a Stack. We can also implement it as recursive functions we will see in the Binary Tree Traversal section.

3.3.3. Depth-First Search vs. Breadth-First Search

We can see visually the difference between how the DFS and BFS search for nodes:

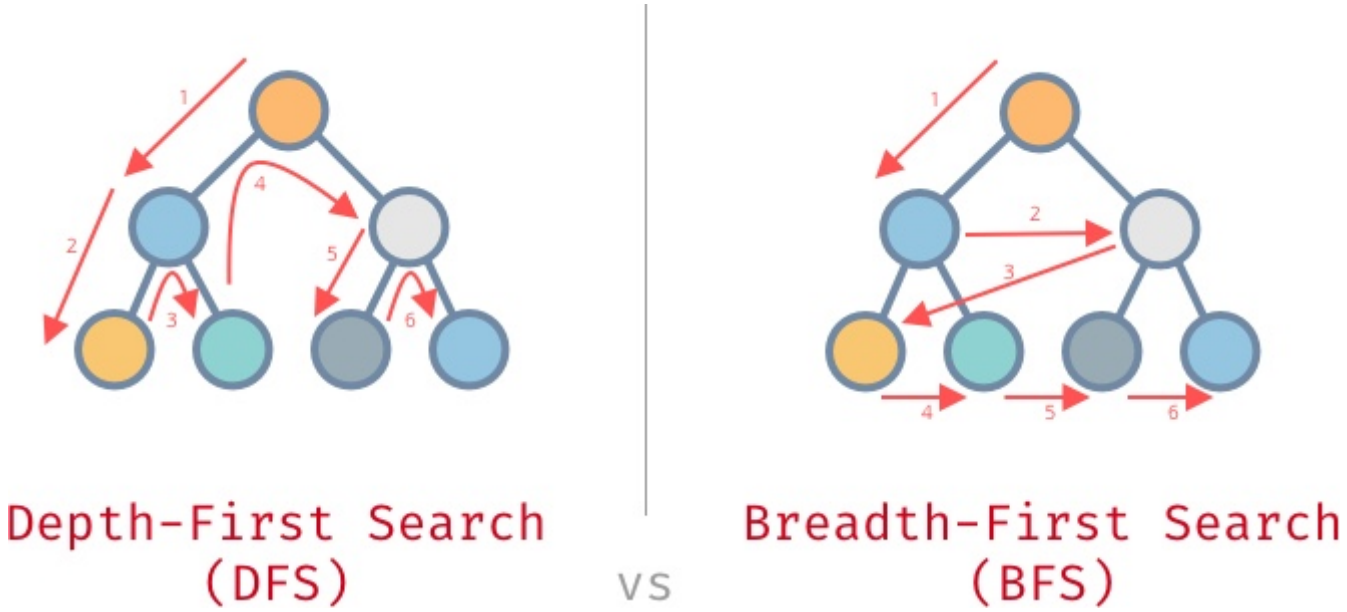


Figure 27. Depth-First Search vs. Breadth-First Search

As you can see, the DFS in two iterations is already at one of the farthest nodes from the root while

BFS search nearby nodes first.

Use DFS when:

- The node you are looking for is likely to be **far** from the root.

Use BFS when:

- The node you are looking for is **nearby** the root.

3.4. Binary Tree Traversal

In this section, we are going to focus on depth-first tree traversal.

3.4.1. In Order Traversal

If your tree happens to be a binary search tree (BST), then you can use "in order" traversal to get the values sorted in ascending order. To accomplish this, you have to visit the nodes in a `left-root-right` order.

If we have the following tree:

```
          10
        /    \
       5      30
     /       /  \
    4       15   40
  /
3
```

In-order traverval will return `3, 4, 5, 10, 15, 30, 40`.

Check out the implementation:

In-order traversal implementation

```
/**
 * In-order traversal on a tree: left-root-right.
 * If the tree is a BST, then the values will be sorted in ascendent
order
 * @param {BinaryTreeNode} node first node to start the traversal
 * @yields {BinaryTreeNode}
 */
* inOrderTraversal(node = this.root) {
  if (node && node.left) { yield* this.inOrderTraversal(node.left); }
  yield node;
  if (node && node.right) { yield* this.inOrderTraversal(node.right); }
}
```

This function gets the leftmost element (recursively) and then yields that node, then we go to the right child (if any) and repeat the process. This method will get us the values ordered.

3.4.2. Pre Order Traversal

Pre-order traversal visits nodes in this order `root-left-right` recursively.

Usage of pre-order traversal:

- Create a copy of the tree.
- Get prefix expression on of an expression tree used in the polish notation.

Pre-order traversal implementation

```
/**
 * Pre-order traversal on a tree: root-left-right.
 * Similar results to DFS
 * @param {BinaryTreeNode} node first node to start the traversal
 * @yields {BinaryTreeNode}
 */
* preOrderTraversal(node = this.root) {
  yield node;
  if (node.left) { yield* this.preOrderTraversal(node.left); }
  if (node.right) { yield* this.preOrderTraversal(node.right); }
}
```

If we have the following tree:

```
          10
        /    \
       5      30
     /       /  \
    4       15   40
  /
 3
```

Pre-order traverval will return `10, 5, 4, 3, 30, 15, 40`.

3.4.3. Post-order Traversal

Post-order traversal goes to each node in this order `left-right-root` recursively.

Usages of the post-order tree traversal

- Traversal is used to delete the tree because you visit the children before removing the parent.
- Get the postfix expression of an expression tree used in the reverse polish notation.

Post-order traversal implementation

```
/**
 * Post-order traversal on a tree: left-right-root.
 * @param {BinaryTreeNode} node first node to start the traversal
 * @yields {BinaryTreeNode}
 */
* postOrderTraversal(node = this.root) {
  if (node.left) { yield* this.postOrderTraversal(node.left); }
  if (node.right) { yield* this.postOrderTraversal(node.right); }
  yield node;
}
```

If we have the following tree:

```
          10
        /    \
       5      30
     /       /  \
    4       15   40
  /
 3
```

Post-order traverval will return `3, 4, 5, 15, 40, 30, 10`.

3.4.4. Practice Questions

Binary Tree Diameter

BT-1) *Find the diameter of a binary tree. A tree's diameter is the longest possible path from two nodes (it doesn't need to include the root). The length of a diameter is calculated by counting the number of edges on the path.*

Common in interviews at: FAANG

Examples:

```
/*           1
            / \
           2   3
          / \
         4   5          */
diameterOfBinaryTree(toBinaryTree([1,2,3,4,5])); // 3
// For len 3, the path could be 3-1-2-5 or 3-1-2-4

/*           1
            / \
           2   3
              / \
             4   5
            /     \
           6       7
          /         \
         8           9      */
const array = [1,2,3,null,null,4,5,6,null,null,7,8,null,null,9];
const tree = BinaryTreeNode.from(array);
diameterOfBinaryTree(tree); // 6 (path: 8-6-4-3-5-7-9)
```

Starter code:

```
/**
 * Find the length of the binary tree diameter.
 *
 * @param {BinaryTreeNode | null} root - Binary Tree's root.
 * @returns {number} tree's diameter (longest possible path on the tree)
 */
function diameterOfBinaryTree(root) {
  // write your code here...
}
```

Solution: Binary Tree Diameter

Binary Tree from right side view

BT-2) *Imagine that you are viewing the tree from the right side. What nodes would you see?*

Common in interviews at: Facebook, Amazon, ByteDance (TikTok).

Examples:

```
/*
 1      <- 1 (only node on level)
 /   \
2     3   <- 3 (3 is the rightmost)
 \
  4       <- 4 (only node on level) */
rightSideView(BinaryTreeNode.from([1, 2, 3, null, 4])); // [1, 3, 4]

rightSideView(BinaryTreeNode.from([])); // []
rightSideView(BinaryTreeNode.from([1, 2, 3, null, 5, null, 4, 6])); //
[1, 3, 4, 6]
```

Starter code:

```
/**
 * Find the rightmost nodes by level.
 *
 * @example rightSideView(bt([1,2,3,4])); // [1, 3, 4]
 *     1      <- 1
 *   /   \
 *  2     3   <- 3
 *   \
 *    4       <- 4
 *
 * @param {BinaryTreeNode} root - The root of the binary tree.
 * @returns {number[]} - array with the rightmost nodes values.
 */
function rightSideView(root) {
  // write your code here...
}
```

Solution: *Binary Tree from right side view*

3.5. Tree Map

A Map is an abstract data structure to store pairs of data: **key** and **value**. It also has a fast key lookup of `O(1)` for Map or `O(log n)` for Tree Map.

We can implement a Map using two different underlying data structures: Hash Map or Tree Map.

3.5.1. HashMap vs TreeMap

A map can be implemented using hash functions or a binary search tree:

- **HashMap**: it's a map implementation using an **array** and a **hash function**. The hash function's job is to convert the `key` into an index that maps to the `value`. HashMap has an average runtime of **O(1)**.
- **TreeMap**: it's a map implementation that uses a self-balanced Binary Search Tree (like AVL Tree or Red-Black Tree). The BST nodes store the key, and the value and nodes are sorted by key guaranteeing an **O(log n)** lookup time.

When to use a TreeMap vs. HashMap?

- `HashMap` is more time-efficient. A `TreeMap` is more space-efficient.
- `TreeMap` search complexity is **O(log n)**, while an optimized `HashMap` is **O(1)** on average.
- `HashMap`'s keys are in insertion order (or random depending on the implementation). `TreeMap`'s keys are always sorted.
- `TreeMap` offers some statistical data for free such as: get minimum, get maximum, median, find ranges of keys. `HashMap` doesn't.
- `TreeMap` has a guarantee always an **O(log n)**, while `HashMap`s has an amortized time of **O(1)** but in the rare case of a rehash, it would take an **O(n)**.

3.5.2. TreeMap Time complexity vs HashMap

As we discussed so far, there is a trade-off between the implementations.

Table 14. Time complexity for different Maps implementations

Data Structure	Searching By		Insert	Delete	Space Complexity
	Index/Key	*Value*			
Hash Map	O(1)	O(n)	O(1)*	O(1)	O(n)
Tree Map (Red-Black Tree)	O(log n)	O(n)	O(log n)	O(log n)	O(n)

* = Amortized run time. E.g. rehashing might affect run time to **O(n)**.

We already covered Hash Map, so in this chapter, we will focus on TreeMap.

JavaScript only provides (Hash) `Map`. That's enough for most needs. But we will implement a TreeMap so it's more clear how it works and when it should be used.

Ok, now that you know the advantages, let's implement it!

Let's get started with the essential functions. They have the same interface as the `HashMap` (but the implementation is different).

TreeMap class overview

```
class TreeMap {
  constructor(){}
  set(key, value) {}
  get(key) {}
  has(key) {}
  delete(key) {}
}
```

3.5.3. Inserting values into a TreeMap

For inserting a value on a TreeMap, we first need to initialize the tree:

TreeMap constructor

```
/**
 * Initialize tree
 */
constructor() {
  this.tree = new Tree();
}
```

The tree can be an instance of any Binary Search Tree that we implemented so far. For better performance, it should be a self-balanced tree like a Red-Black Tree or AVL Tree.

Let's implement the method to add values to the tree.

TreeMap `add` method and `size` attribute

```
/**
 * Insert a key/value pair into the map.
 * If the key is already there replaces its content.
 * Runtime: O(log n)
 * @param {any} key
 * @param {any} value
 * @returns {TreeNode} Return the Map object to allow chaining
 */
set(key, value) {
  const node = this.tree.get(key);
  if (node) {
    node.data(value);
    return node;
  }
  return this.tree.add(key).data(value);
}

/**
 * Size of the map
 */
get size() {
  return this.tree.size;
}
```

Adding values is very easy (once we have the underlying tree implementation).

3.5.4. Getting values out of a TreeMap

When we search by key in a treemap, it takes **O(log n)**. The following is a possible implementation:

TreeMap `get` and `has` method

```
/**
 * Gets the value out of the map
 * Runtime: O(log n)
 * @param {any} key
 * @returns {any} value associated to the key, or undefined if there is
none.
 */
get(key) {
  const node = this.tree.get(key) || undefined;
  return node && node.data();
}

/**
 * Search for key and return true if it was found
 * Runtime: O(log n)
 * @param {any} key
 * @returns {boolean} indicating whether an element
 *  with the specified key exists or not.
 */
has(key) {
  return !!this.get(key);
}
```

One side effect of storing keys in a tree is that they don't come up in insertion order. Instead, they ordered by value.

TreeMap iterators

```
/**
 * Default iterator for this map
 */
* [Symbol.iterator]() {
  yield* this.tree.inOrderTraversal(); ①
}

/**
 * Keys for each element in the Map object
 *  in order ascending order.
 * @returns {Iterator} keys
 */
* keys() {
  for (const node of this) {
    yield node.value;
  }
}

/**
 * Values for each element in the Map object
 *  in corresponding key ascending order.
 * @returns {Iterator} values without holes (empty spaces of deleted
values)
 */
* values() {
  for (const node of this) {
    yield node.data();
  }
}
```

① We implemented the default iterator using the in-order traversal. That's useful for getting the keys sorted.

JavaScript Iterators and Generators

Generators are useful for producing values that can you can iterate in a `for...of` loop. Generators use the `function*` syntax, which expects to have a `yield` with a value.

3.5.5. Deleting values from a TreeMap

Removing elements from TreeMap is simple.

TreeMap `delete` *method*

```
/**
 * Removes the specified element from the map.
 * Runtime: O(log n)
 * @param {*} key
 * @returns {boolean} true if an element in the Map object existed
 *  and has been removed, or false if the element did not exist.
 */
delete(key) {
  return this.tree.remove(key);
}
```

The BST implementation does all the heavy lifting.

That's it! To see the full file in context, click here: here

3.6. Tree Set

A tree set is a data structure that stores unique values and keeps them sorted. You can get check if a value exists in `O(log n)` time.

Another way to implement a Set is by using a hash function, as we covered on Hash Set. There are some critical differences between the two implementations.

3.6.1. HashSet vs TreeSet

We can implement a `map` using a **balanced binary search tree** or a **hash function**. If we use them to implement a `Set`, we would have a `HashSet` and `TreeSet`. As with all data structures, there are trade-offs. Here are some key differences:

- `TreeSet`, would return the values sorted in ascending order.
- `HashSet`, would return the values in insertion order.
- Operations on a `HashSet` would take on average O(1), and in the worst case (rehash is due), it would take O(n).
- Operation on a `TreeSet` is always O(log n).

3.6.2. Time Complexity Hash Set vs Tree Set

Table 15. Time complexity HashSet vs TreeSet

Data Structure	Searching By		Insert	Delete	Space Complexity
	Index/Key	*Value*			
Hash Set	O(1)	-	O(1)*	O(1)	O(n)
Tree Set	O(log n)	-	O(log n)	O(log n)	O(n)

* = Amortized run time. E.g. rehashing might affect run time to **O(n)**.

JavaScript only provides (Hash) `Set` that's enough for most needs. But we will implement a Tree Set so it's more clear how it works and when it should be used.

3.6.3. Implementing a Tree Set

TreeSet's constructor method and size attribute

```
const Tree = require('../trees/red-black-tree');

/**
 * TreeSet implements a Set (collection of unique values)
 *  using a balanced binary search tree to guarantee a O(log n) in all
operations.
 */
class TreeSet {
  /**
   * Initialize tree and accept initial values.
   * @param {array} iterable initial values (new set won't have
duplicates)
   */
  constructor(iterable = []) {
    this.tree = new Tree();
    Array.from(iterable).forEach((value) => this.add(value)); ①
  }

  /**
   * Size of the set
   */
  get size() {
    return this.tree.size;
  }
}
```

① Converts an array or any iterable data structure to a set.

An everyday use case for Sets is to remove duplicated values from an array. We can do that by bypassing them in the constructor as follows:

Removing duplicates from an Array using a Set

```
set = new TreeSet([1, 2, 3, 2, 1]);
expect(set.size).toBe(3);
expect(Array.from(set.keys())).toEqual([1, 2, 3]);
```

Ok, now let's implement the add method.

Adding elements to a TreeSet

For adding values to the set, we `Tree.add` method.

TreeSet's constructor method and size attribute

```
/**
 * Add a new value (duplicates will be added only once)
 * Runtime: O(log n)
 * @param {any} value
 */
add(value) {
  if (!this.has(value)) {
    this.tree.add(value);
  }
}
```

Our BST implementation can hold duplicated values. It has a multiplicity tally to keep track of duplicates. However, we don't dupe in a set. For that, we check if the value is already in the tree. Don't worry about adding extra lookups. The `Tree.has` is also very performant **O(log n)**.

Searching for values in a TreeSet

Again, we rely on the Tree implementation to do the heavy lifting:

TreeSet's `has` method

```
/**
 * Check if value is already on the set
 * Runtime: O(log n)
 * @param {any} value
 * @returns {boolean} true if exists or false otherwise
 */
has(value) {
  return this.tree.has(value);
}
```

Deleting elements from a TreeSet

We delete the elements from the TreeSet using the remove method of the BST.

TreeSet's `delete` method

```
/**
 * Delete a value from the set
 * Runtime: O(log n)
 * @param {any} value
 */
delete(value) {
  return this.tree.remove(value);
}
```

Voilà! That's it!

Converting TreeSet to Array

Another use case for a Set is to convert it to an array or use an iterator (for loops, forEach, ...). Let's provide the method for that:

TreeSet's iterator

```
/**
 * Default iterator for this set
 * @returns {iterator} values in ascending order
 */
* [Symbol.iterator]() {
  for (const node of this.tree.inOrderTraversal()) {
    yield node.value;
  }
}
```

We are using the `inOrderTraversal` method of the BST to go each key in an ascending order.

JavaScript Built-in `Symbol` iterator

The `Symbol.iterator` built-in symbol specifies the default iterator for an object. Used by `for...of`, `Array.from`, and others.

Now we can convert from set to array and vice versa easily. For instance:

TreeSet's iterator

```
const array = [1, 1, 2, 3, 5];

// array to set
const set = new TreeSet(array);

// set to array
Array.from(set); //↪ (4) [1, 2, 3, 5]
```

No more duplicates in our array!

Check out our GitHub repo for the full TreeSet implementation.

3.7. Graph

Graphs are one of my favorite data structures. They have many exciting applications like optimizing routes and social network analysis, to name a few. You are probably using apps that use graphs every day. First, let's start with the basics.

A graph is a non-linear data structure where a node can have zero or more connected nodes.

You can think of a Graph as an extension of a Linked List. Instead of having a `next` or `previous` reference, you can have as many as you want. You can implement a graph node as an array of associated nodes.

Node's constructor

```
/**
 * Graph node/vertex that hold adjacencies nodes
 * For performance, uses a HashSet instead of array for adjacents.
 */
class Node {
  constructor(value) {
    this.value = value;
    this.adjacents = new HashSet(); // adjacency list
  }
}
```

As you can see, it's pretty similar to the Linked List node. The only difference is that it uses an **array** of adjacent nodes instead of just one or two.

Another difference between a linked list and a Graph is that a linked list always has a root node (or first element), while the Graph doesn't. You can start traversing a graph from anywhere. Let's examine these graph properties!

3.7.1. Graph Properties

The connection between two nodes is called **edge**. Also, nodes might be called **vertex**.

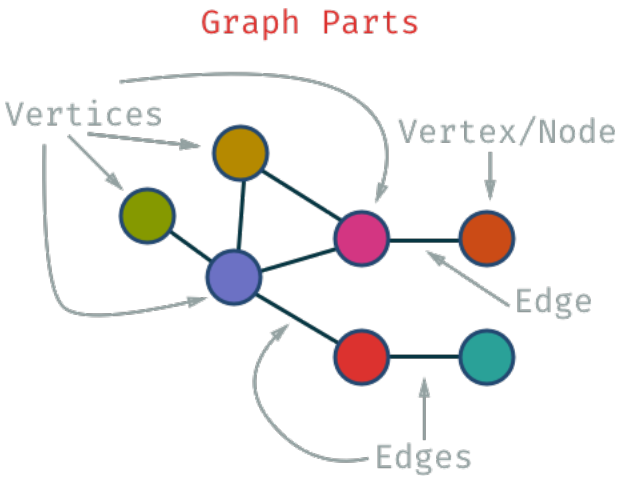


Figure 28. Graph is composed of vertices/nodes and edges

Directed Graph vs Undirected

A graph can be either **directed** or **undirected**.

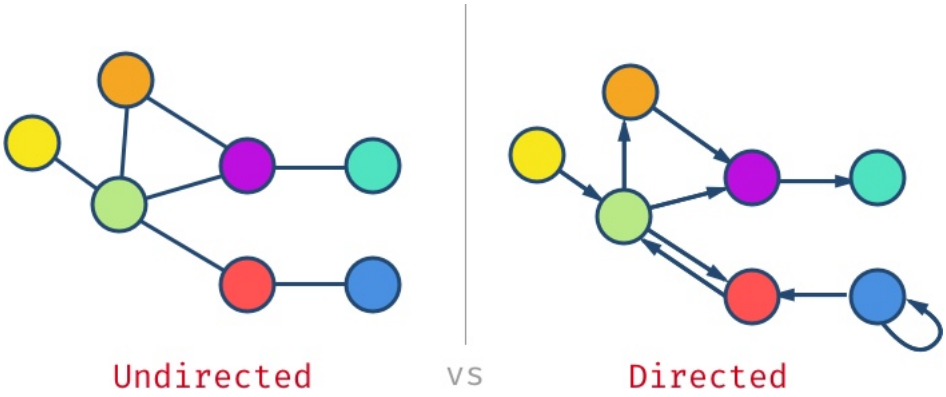


Figure 29. Graph: directed vs undirected

An **undirected graph** has edges that are **two-way street**. E.g., On the undirected example, you can traverse from the green Node to the orange and vice versa.

A **directed graph (digraph)** has edges that are **one-way street**. E.g., On the directed example, you can only go from green Node to orange and not the other way around. When one Node has an edge to itself is called a **self-loop**.

Graph Cycles

A graph can have **cycles** or not.

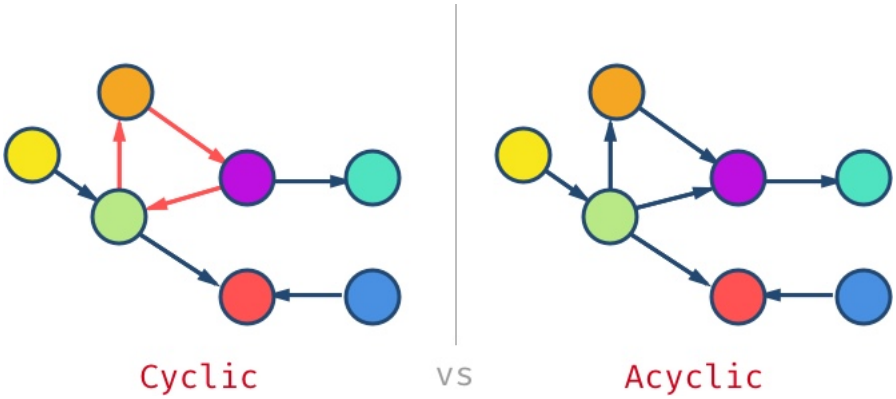


Figure 30. Cyclic vs. Acyclic Graphs.

A **cyclic graph** is the one that you can pass through a node more than once. E.g., On the cyclic illustration, if you start in the green Node, go the orange and purple; finally, you could come back to green again. Thus, it has a **cycle**. An acyclic graph is the one that you can't pass through a node more than once. E.g., in the acyclic illustration, can you find a path where you can pass through the same vertex more than one?

The **Directed Acyclic Graph (DAG)** is unique. It has many applications like scheduling tasks, spreadsheets' change propagation, and so forth. DAG is also called **Tree** data structure only when each Node has only **one parent**.

Connected vs Disconnected vs Complete Graphs

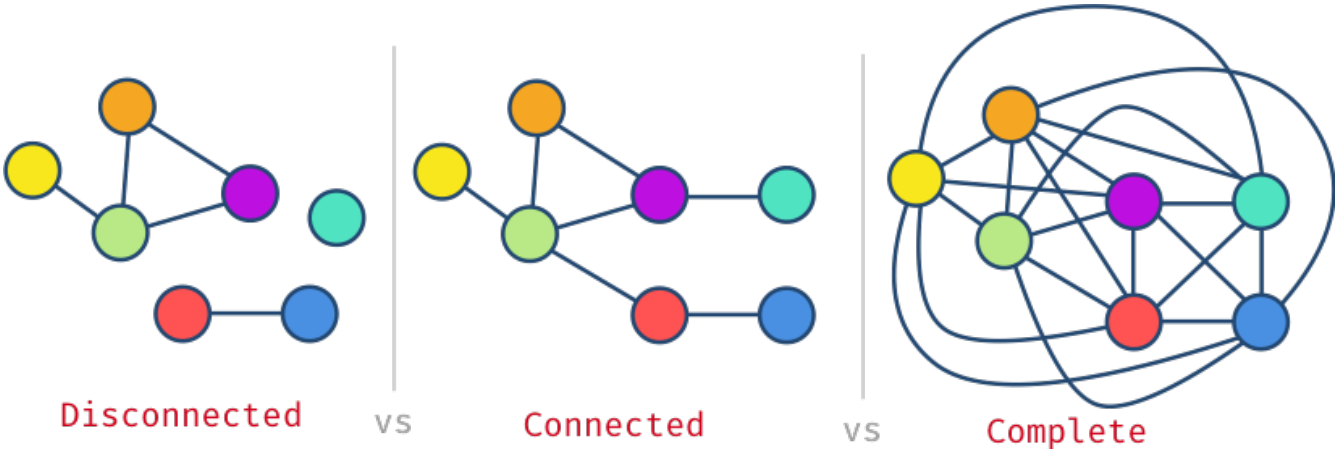


Figure 31. Different kinds of graphs: disconnected, connected, and complete.

A **disconnected graph** is one that has one or more subgraphs. In other words, a graph is **disconnected** if two nodes don't have a path between them.

A **connected graph** is the opposite of disconnected; there's a path between every Node. No one is left stranded.

A **complete graph** is where every Node is adjacent to all the other nodes in the Graph. E.g., If there are seven nodes, every Node has six edges.

Weighted Graphs

Weighted graphs have labels in the edges (a.k.a **weight** or **cost**). The link weight can represent

many things like distance, travel time, or anything else.

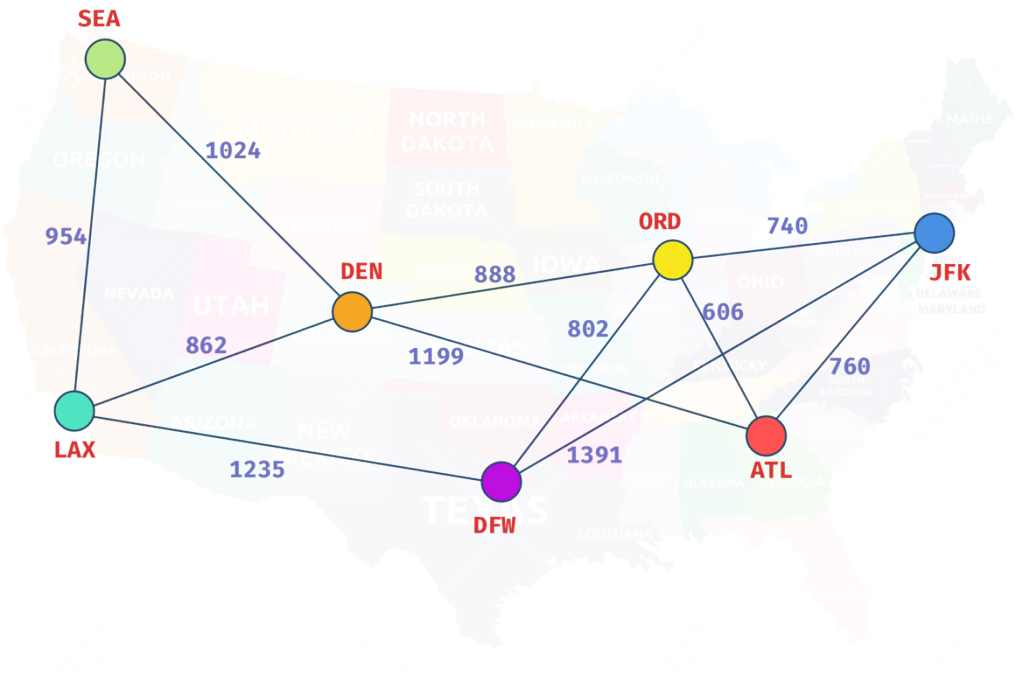


Figure 32. Weighted Graph representing USA airports distance in miles.

For instance, a weighted graph can have a distance between nodes. So, algorithms can use the weight and optimize the path between them.

3.7.2. Exciting Graph applications in real-world

Now that we know what graphs are and some of their properties. Let's discuss some real-life usages of graphs.

Graphs become a metaphor where nodes and edges model something from our physical world. To name a few:

- Optimizing Plane traveling
  - Nodes = Airport
  - Edges = Direct flights between two airports
  - Weight = miles between airports | cost | time
- GPS Navigation System
  - Node = road intersection
  - Edge = road
  - Weight = time between intersections
- Network routing
  - Node = server
  - Edge = data link
  - Weight = connection speed

There are endless applications for graphs in electronics, social networks, recommendation systems, and many more. That's cool and all, but how do we represent graphs in code? Let's see that in the next section.

3.7.3. Representing Graphs

There are two main ways to graphs one is:

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

Representing graphs as adjacency matrix is done using a two-dimensional array. For instance, let's say we have the following Graph:

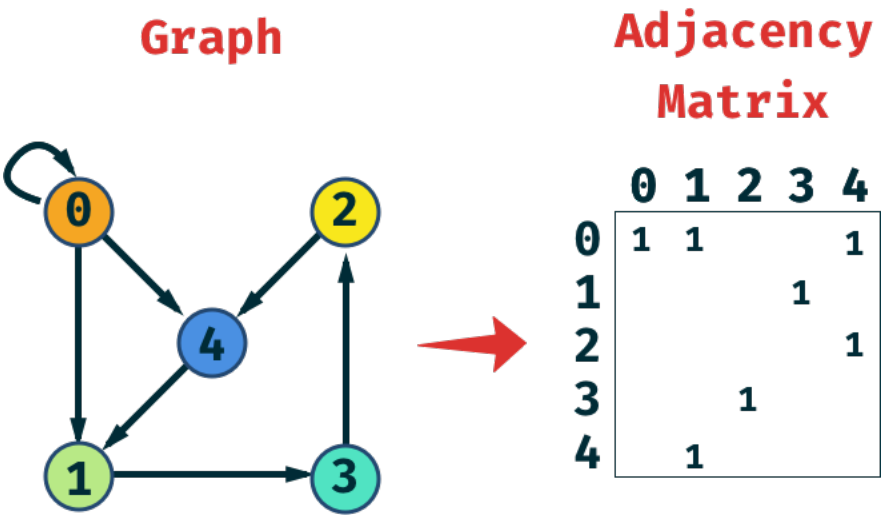


Figure 33. Graph and its adjacency matrix.

The number of vertices, $|V|$, defines the size of the matrix. In the example, we have five vertices, so we have a 5x5 matrix.

We fill up the matrix row by row. Mark with 1 (or any other weight) when you find an edge. E.g.

- **Row 0:** It has a self-loop, so it has a 1 in the coordinate 0,0. Node 0 also has an edge to 1 and 4, so we mark it.
- **Row 1:** node 1 has one edge to 3, so we check it.
- **Row 2:** Node 2 goes to Node 4, so we note the insertion with 1.
- etc.

The example graph above is a directed graph (digraph). In the case of an undirected graph, the matrix would be symmetrical by the diagonal.

If we represent the example graph in code, it would be something like this:

```
const digraph = [
  [1, 1, 0, 0, 1],
  [0, 0, 0, 1, 0],
  [0, 0, 0, 0, 1],
  [0, 0, 1, 0, 0],
  [0, 1, 0, 0, 0],
];
```

It would be very easy to tell if two nodes are connected. Let's query if node 2 is connected to 3:

```
digraph[2][3]; //=> 0
digraph[3][2]; //=> 1
```

As you can see, we don't have a link from node 2 to 3, but we do in the opposite direction. Querying arrays is constant time **O(1)**, so not bad at all.

The issue with the adjacency matrix is the space it takes. Let's say you want to represent the entire Facebook network on a digraph. You would have a massive matrix of 1.2 billion x 1.2 billion. The worst part is that most of it would be empty (zeros) since people are friends to at most a few thousands.

When the Graph has few connections compared to the number of nodes, we say that we have a **sparse graph**. Conversely, if we have almost complete maps, we say we have a **dense graph**.

The adjacency matrix's space complexity is $\mathbf{O(|V|^2)}$, where $|V|$ is the number of vertices/nodes.

Adjacency List

Another way to represent a graph is by using an adjacency list. This time instead of using an array (matrix), we use a list.

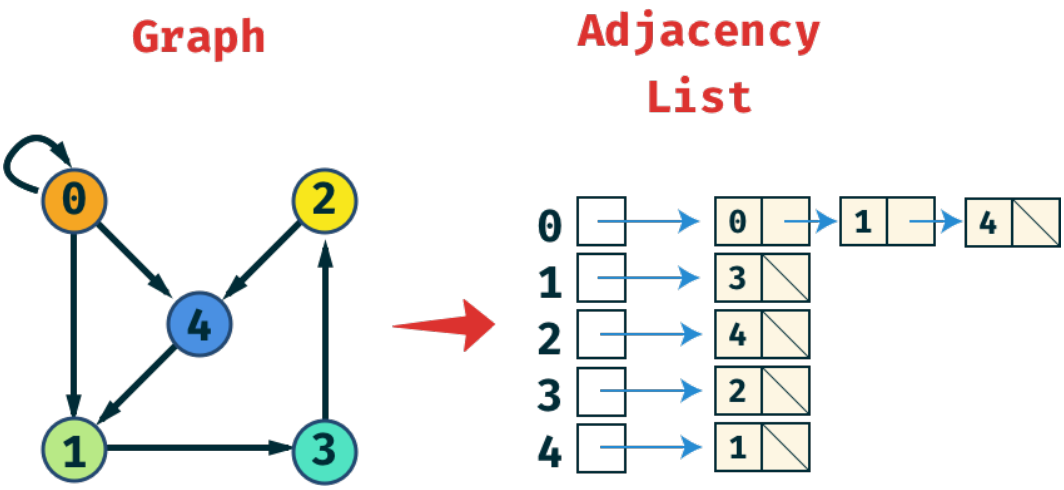


Figure 34. Graph represented as an Adjacency List.

If we want to add a new node to the list, we can do it by adding one element to the end of the array of nodes **O(1)**. In the next section, we will explore the running times of all operations in an adjacency list.

3.7.4. Implementing a Graph data structure

Since adjacency lists are more efficient (than adjacency matrix), we will use to implement a graph data structure.

Let's start by creating the constructor of the Graph class.

Graph's constructor

```
/**
 * Graph data structure implemented with an adjacent list
 */
class Graph {
  /**
   * Initialize the nodes map
   * @param {Symbol} edgeDirection either `Graph.DIRECTED` or
`Graph.UNDIRECTED`
   */
  constructor(edgeDirection = Graph.DIRECTED) {
    this.nodes = new HashMap();
    this.edgeDirection = edgeDirection;
  }
```

Notice that the constructor takes a parameter. The `edgeDirection` allows us to use one class for both undirected and directed graphs.

3.7.5. Adding a vertex

For adding a vertex, we first need to check if the Node already exists. If so, we return the Node.

Graphs's `addVertex` method

```
/**
 * Add a node to the graph.
 * Runtime: O(1)
 * @param {any} value node's value
 * @returns {Node} the new node or the existing one if it already exits.
 */
addVertex(value) {
  if (this.nodes.has(value)) { ①
    return this.nodes.get(value);
  }
  const vertex = new Node(value); ②
  this.nodes.set(value, vertex); ③
  return vertex;
}
```

① Check if value is already on the graph. If it is, then return it.

② Create new `Node` with the given value.

③ Set `hashMap` with value and node pair.

If the Node doesn't exist, we create the new Node and add it to a `HashMap`.

Tree Map stores key/pair value very efficiently. Lookup is `O(1)`.

The `key` is the Node's value, while the `value` is the newly created Node.

The `Node` class is constructed as follows:

Node's class (for Graph data structure)

```
/**
 * Graph node/vertex that hold adjacencies nodes
 * For performance, uses a HashSet instead of array for adjacents.
 */
class Node {
  constructor(value) {
    this.value = value;
    this.adjacents = new HashSet(); // adjacency list
  }
}
```

3.7.6. Deleting a vertex

Graphs's `removeVertex` *method*

```
/**
 * Removes node from graph
 * It also removes the reference of the deleted node from
 *  anywhere it was adjacent to.
 * Runtime: O(|V|) because adjacency list is implemented with a HashSet.
 * It were implemented with an array then it would be O(|V| + |E|).
 * @param {any} value node's value
 */
removeVertex(value) {
  const current = this.nodes.get(value); ①
  if (current) {
    Array.from(this.nodes.values()).forEach((node) => node
.removeAdjacent(current)); ②
  }
  return this.nodes.delete(value); ③
}
```

① Try to find if node exists.

② Remove related edges. See `removeAdjacent` below.

③ Remove Node with the given value.

Notice on callout 2 that we visit every edge on the Graph and remove the ones that contain the Node to remove.

For removing adjacent nodes, we use Node's method called `removeAdjacent` that can be implemented as follows:

Node's `removeAdjacent`

```
/**
   * Remove node from adjacency list
   * Runtime: O(1)
   * @param {Node} node
   * @returns removed node or `false` if node was not found
   */
removeAdjacent(node) {
  return this.adjacents.delete(node);
}
```

All adjacencies are stored as a HashSet to provide constant-time deletion.

3.7.7. Adding an edge

An edge is a connection between two nodes (vertices). If the Graph is undirected means that every

link is a two-way street. When we create the edge from node 1 to node 2, we also need to establish a connection between nodes 2 and 1 for undirected graphs.

If we are dealing with a digraph (directed Graph), then we create one edge.

Graphs's `addEdge` *method*

```
/**
 * Create a connection between the source node and the destination node.
 * If the graph is undirected, it will also create the link from
destination to source.
 * If the nodes don't exist, then it will make them on the fly.
 * Runtime: O(1)
 * @param {any} source
 * @param {any} destination
 * @returns {[Node, Node]} source/destination node pair
 */
addEdge(source, destination) {
  const sourceNode = this.addVertex(source); ①
  const destinationNode = this.addVertex(destination); ①

  sourceNode.addAdjacent(destinationNode); ②

  if (this.edgeDirection === Graph.UNDIRECTED) {
    destinationNode.addAdjacent(sourceNode); ③
  }

  return [sourceNode, destinationNode];
}
```

① Find or create nodes if they don't exists yet.

② Create edge from source to destination.

③ If it's an undirected graph, create the edge in the other direction.

We can add adjacencies using the `addAdjacent` method from the Node class.

Node's `addAdjacent`

```
/**
   * Add node to adjacency list
   * Runtime: O(1)
   * @param {Node} node
   */
addAdjacent(node) {
  this.adjacents.add(node);
}
```

3.7.8. Querying Adjacency

Graphs's `areAdjacents` *method*

```
/**
 * True if two nodes are adjacent.
 * @param {any} source node's value
 * @param {any} destination node's value
 */
areAdjacents(source, destination) {
  const sourceNode = this.nodes.get(source);
  const destinationNode = this.nodes.get(destination);

  if (sourceNode && destinationNode) {
    return sourceNode.isAdjacent(destinationNode);
  }

  return false;
}
```

Node's `isAdjacent`

```
/**
   * Check if a Node is adjacent to other
   * Runtime: O(1)
   * @param {Node} node
   */
isAdjacent(node) {
  return this.adjacents.has(node);
}
```

3.7.9. Deleting an edge

Graphs's `removeEdge` *method*

```
/**
 * Remove the connection between source node and destination.
 * If the graph is undirected, it will also create the link from
destination to source.
 *
 * Runtime: O(1): implemented with HashSet.
 * If implemented with array, would be O(|E|).
 *
 * @param {any} source
 * @param {any} destination
 */
removeEdge(source, destination) {
  const sourceNode = this.nodes.get(source);
  const destinationNode = this.nodes.get(destination);

  if (sourceNode && destinationNode) {
    sourceNode.removeAdjacent(destinationNode);

    if (this.edgeDirection === Graph.UNDIRECTED) {
      destinationNode.removeAdjacent(sourceNode);
    }
  }

  return [sourceNode, destinationNode];
}
```

Node's `removeAdjacent`

```
/**
   * Remove node from adjacency list
   * Runtime: O(1)
   * @param {Node} node
   * @returns removed node or `false` if node was not found
   */
removeAdjacent(node) {
  return this.adjacents.delete(node);
}
```

3.7.10. Graph Complexity

Table 16. Time complexity for a Graph data structure

Data Structure	Vertices		Edges		Space Complexity
	Add	*Remove*	*Add*	*Remove*	
Graph (adj. matrix)	O(|V|^2)	O(|V|^2)	O(1)	O(1)	O(|V|^2)
Graph (adj. list w/array)	O(1)	O(|V| + |E|))	O(1)	O(|E|)	O(|V| + |E|)
Graph (adj. list w/HashSet)	O(1)	O(|V|))	O(1)	O(1)	O(|V| + |E|)

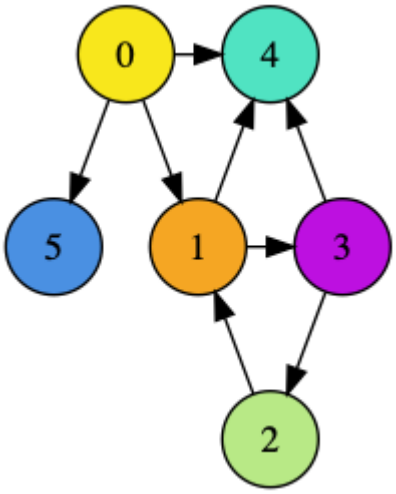
As you can see, using a `HashSet` on for the adjacency list make a performance improvement if you need to query for connectivity. However, this is rarely required. Most graph algorithms visit all adjacent nodes one by one.

3.8. Graph Search

Graph search allows you to visit search elements.

Graph search is very similar to Tree Search & Traversal. So, if you read that section, some of the concepts here will be familiar to you.

There are two ways to navigate the graph, one is using Depth-First Search (DFS), and the other one is Breadth-First Search (BFS). Let's see the difference using the following graph.



3.8.1. Depth-First Search for Graphs

With Depth-First Search (DFS), we go deep before going wide.

We use DFS on the graph shown above, starting with node `0`. A DFS will probably visit 5, then visit `1` and continue going down `3` and `2`. As you can see, we need to keep track of visited nodes, since, in graphs, we can have cycles like `1-3-2`. Finally, we back up to the remaining node `0` children: node `4`.

So, DFS would visit the graph: `[0, 5, 1, 3, 2, 4]`.

3.8.2. Breadth-First Search for Graphs

With Breadth-First Search (BFS), we go wide before going deep.

We use BFS on the graph shown above, starting with the same node `0`. A BFS will visit 5 as well, then visit `1` and not go down to its children. It will first finish all the children of node `0`, so it will visit node `4`. After all the children of node `0` are visited, it will continue with all the children of node `5`, `1`, and `4`.

In summary, BFS would visit the graph: `[0, 5, 1, 4, 3, 2]`

3.8.3. Depth-First Search vs. Breadth-First Search in a Graph

DFS and BFS can implementation can be almost identical; the difference is the underlying data

structured. In our implementation, we have a generic `graphSearch` where we pass the first element to start the search the data structure that we can to use:

DFS and BFS implementation

```
/**
 * Depth-first search
 * Use a stack to visit nodes (LIFO)
 * @param {Node} first node to start the dfs
 */
static* dfs(first) {
  yield* Graph.graphSearch(first, Stack);
}

/**
 * Depth-first search
 * Use a queue to visit nodes (FIFO)
 * @param {Node} first node to start the dfs
 */
static* bfs(first) {
  yield* Graph.graphSearch(first, Queue);
}

/**
 * Generic graph search where we can pass a Stack or Queue
 * @param {Node} first node to start the search
 * @param {Stack|Queue} Type Stack for DFS or Queue for BFS
 */
static* graphSearch(first, Type = Stack) {
  const visited = new Map();
  const visitList = new Type();

  visitList.add(first);

  while (!visitList.isEmpty()) {
    const node = visitList.remove();
    if (node && !visited.has(node)) {
      yield node;
      visited.set(node);
      node.getAdjacents().forEach((adj) => visitList.add(adj));
    }
  }
}
```

Using an Stack (LIFO) for DFS will make use keep visiting the last node children while having a Queue (FIFO) will allow to visit adjacent nodes first and "queue" their children for later visiting.

you can also implement the DFS as a recursive function, similar to what we did in the DFS for trees.

You might wonder what the difference between search algorithms in a tree and a graph is? Check out the next section.

3.8.4. DFS/BFS on Tree vs Graph

The difference between searching a tree and a graph is that the tree always has a starting point (root node). However, in a graph, you can start searching anywhere. There's no root.

Every tree is a graph, but not every graph is a tree. Only acyclic directed graphs (DAG) are trees.

3.8.5. Practice Questions

Course Schedule

gr-1) *Check if it's possible to take all courses while satisfying their prerequisites.*

Common in interviews at: Amazon, Facebook, Bytedance (TikTok).

Starter code:

```
/**
 * Check if you can finish all courses with their prerequisites.
 * @param {number} n - The number of courses
 * @param {[number, number][]} prerequisites - Array of courses pairs.
 *  E.g. [[200, 101]], to take course 202 you need course 101 first.
 * @returns {boolean} - True = can finish all courses, False otherwise
 */
function canFinish(n, prerequisites) {
  // write your code here...
}
```

Examples:

```
canFinish(2, [[1, 0]]); // true
// 2 courses: 0 and 1. One prerequisite: 0 -> 1
// To take course 1 you need to take course 0.
// Course 0 has no prerequisite, so you can take 0 and then 1.

canFinish(2, [[1, 0], [0, 1]]); // false
// 2 courses: 0 and 1. Two prerequisites: 0 -> 1 and 1 -> 0.
// To take course 1, you need to take course 0.
// To Course 0, you need course 1, so you can't any take them!

canFinish(3, [[2, 0], [1, 0], [2, 1]]); // true
// 3 courses: 0, 1, 2. Three prerequisites: 0 -> 2 and 0 -> 1 -> 2
// To take course 2 you need course 0, course 0 has no prerequisite.
// So you can take course 0 first, then course 1, and finally course 2.

canFinish(4, [[1, 0], [2, 1], [3, 2], [1, 3]]); // false
// 4 courses: 0, 1, 2, 3. Prerequisites: 0 -> 1 -> 2 -> 3 and 3 -> 1.
// You can take course 0 first since it has no prerequisite.
// For taking course 1, you need course 3. However, for taking course 3
// you need 2 and 1. You can't finish then!
```

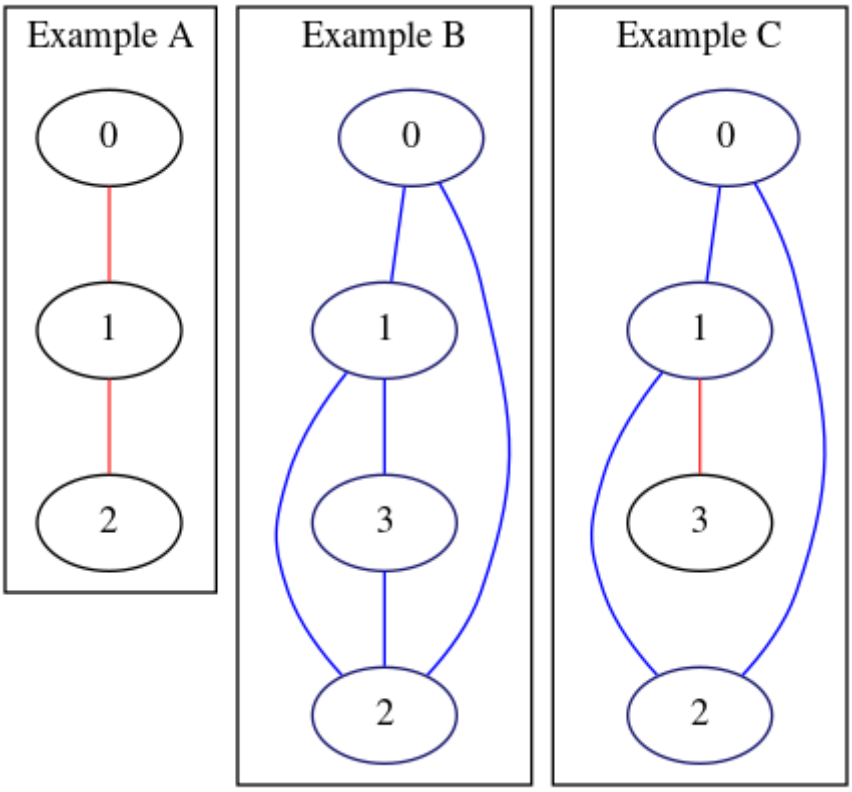
Solution: Course Schedule

Critical Network Paths

gr-2) *Given* `n` *servers and the connections between them, return the critical paths.*

Common in interviews at: FAANG.

Examples:




```
// Example A
criticalConnections(3, [[0, 1], [1, 2]]);// [[0, 1], [1, 2]]
// if you remove any link, there will be stranded servers.

// Example B
criticalConnections(4, [[0, 1], [1, 2], [2, 0], [1, 3], [3, 2]]);// []
// you can remove any connection and all servers will be reachable.

// Example C
criticalConnections(4, [[0, 1], [1, 2], [2, 0], [1, 3]]); // [[1, 3]]
// if you remove [1, 3], then server 3 won't be reachable.
// If you remove any other link. It will be fine.
```

Starter code:

```
function criticalConnections(n, connections) {
  // write your code here...
}
```

Solution: Critical Network Paths

3.9. Summary

In this section, we learned about Graphs, applications, properties, and how we can create them. We mention that you can represent a graph as a matrix or as a list of adjacencies. We went for implementing the latter since it's more space-efficient. We cover the basic graph operations like adding and removing nodes and edges. In the algorithms section, we are going to cover searching values in the graph.

Table 17. Time and Space Complexity for Graph-based Data Structures

Data Structure	Searching By		Insert	Delete	Space Complexity
	Index/Key	*Value*			
BST (unbalanced)	-	O(n)	O(n)	O(n)	O(n)
BST (balanced)	-	O(log n)	O(log n)	O(log n)	O(n)
Hash Map (naïve)	O(n)	O(n)	O(n)	O(n)	O(n)
HashMap (optimized)	O(1)	O(n)	O(1)*	O(1)	O(n)
TreeMap (Red-Black Tree)	O(log n)	O(n)	O(log n)	O(log n)	O(n)
HashSet	O(1)	-	O(1)*	O(1)	O(n)
TreeSet	O(log n)	-	O(log n)	O(log n)	O(n)

* = Amortized run time. E.g. rehashing might affect run time to **O(n)**.

PART FOUR

4. Algorithmic Toolbox

In this part of the book, we will cover examples of classical algorithms in more detail. Also, we will provide algorithmic tools for improving your problem-solving skills.

There's no single approach to solve all problems, but knowing well-known techniques can help you build your own faster.

We are going to start with Sorting Algorithms such as Insertion Sort, Merge Sort and some others. Later, you will learn some algorithmic paradigms that will help you identify common patterns and solve problems from different angles.

We are going to discuss the following techniques for solving algorithms problems:

- Greedy Algorithms: makes greedy choices using heuristics to find the best solution without looking back.
- Dynamic Programming: a technique for speeding up recursive algorithms when there are many *overlapping subproblems*. It uses *memoization* to avoid duplicating work.
- Divide and Conquer: *divide* problems into smaller pieces, *conquer* each subproblem, and then *join* the results.
- Backtracking: search *all (or some)* possible paths. However, it stops and *go back* as soon as notice the current solution is not working.
- *Brute Force*: generate all possible solutions and tries all of them. (Use it as a last resort or as the starting point and to later optimize it).

4.1. Sorting Algorithms

Sorting is one of the most common solutions when we want to extract some insights about data. We can sort to get the maximum or minimum value, and many algorithmic problems can benefit from sorting.

We are going to explore three basic sorting algorithms $O(n^2)$ which have low overhead:

- Bubble Sort
- Selection Sort
- Insertion Sort

and then discuss efficient sorting algorithms O(n log n) such as:

- Merge Sort
- Quicksort

Before we dive into the most well-known sorting algorithms, let's discuss the sorting properties.

4.1.1. Sorting Properties

Sorting implementations with the same time complexity might manipulate the data differently. We want to understand these differences to be aware of the side effects on data or extra resources they will require. For instance, some solutions will need auxiliary memory to store temporary data

while sorting, while others can do it in place.

Sorting properties are stable, adaptive, online, and in-place. Let's go one by one.

Stable

An stable sorting algorithms keep the relative order of items with the same comparison criteria.

This incredibly useful when you want to sort on multiple phases.

Let's say you have the following data:

```
const users = [
  { name: 'Bob', age: 32 },
  { name: 'Alice', age: 30 },
  { name: 'Don', age: 30 },
  { name: 'Charly', age: 32 },
];
```

If you sort by `name`*, you would have:*

```
[
  { name: 'Alice', age: 30 },
  { name: 'Bob', age: 32 },
  { name: 'Charly', age: 32 },
  { name: 'Don', age: 30 },
];
```

Then, here comes the *critical* part; if you sort by `age`, you might get (at least two) different results.

If the sorting algorithm is **stable***; it should keep the items with the same age ordered by* `name`*:*

```
[
  { name: 'Alice', age: 30 },
  { name: 'Don', age: 30 },
  { name: 'Bob', age: 32 },
  { name: 'Charly', age: 32 },
];
```

However, if the sorting is **not stable***, then you will lose the relative order of the items and get something like this:*

```
[
  { name: 'Don', age: 30 },
  { name: 'Alice', age: 30 },
  { name: 'Charly', age: 32 },
  { name: 'Bob', age: 32 },
];
```

Both results are sorted by `age`; however, having a stable sorting is better if you want to keep the relative position of data with the same value.

In-place

An in-place sorting algorithm would have a *space complexity* of O(1). In other words, it does not use any additional memory because it moves the items in-place. No extra memory for sorting is especially useful for large amounts of data or in memory constraint environments like robotics, smart devices, or embedded systems in appliances.

Online

It can sort a list as it receives it. Online sorting algorithms don't have to re-sort the whole collection for every new item added.

Adaptive

Algorithms with adaptive sorting run faster, close to *O(n)*, on an already sorted (or partially sorted) collection.

4.1.2. Bubble Sort

Bubble sort is a simple sorting algorithm that "bubbles up" the biggest values to the array's right side. It's also called *sinking sort* because of the most significant values "sink" to the array's right side. This algorithm is adaptive, which means that if the array is already sorted, it will take only *O(n)* to "sort". However, if the array is entirely out of order, it will require $O(n^2)$ to sort.

Bubble Sort Implementation

Bubble Sort implementation in JavaScript

```
/**
 * Bubble sort - Bubbles up bigger values to the right side
 * Runtime: O(n^2)
 * @param {Array|Set} collection elements to be sorted
 */
function bubbleSort(collection) {
  const array = Array.from(collection); ①

  for (let i = 1; i < array.length; i++) { ⑥
    let swapped = false;

    for (let current = 0; current < array.length - i; current++) { ④
      if (array[current] > array[current + 1]) { ②
        swap(array, current, current + 1); ③
        swapped = true;
      }
    }

    if (!swapped) break; ⑤
  }

  return array;
}
```

① Convert any iterable (array, sets, etc.) into an array or if it's already an array, it clones it, so the input is not modified.

② Starting from index 0 compare current and next element

③ If they are out of order, swap the pair

④ Repeat pair comparison until the last element that has been bubbled up to the right side `array.length - i`.

⑤ (optimization) If there were no swaps, this means that the array is sorted. This single-pass makes this sorting *adaptive*, and it will only require *O(n)* operations.

⑥ Each step moves the largest element from where it was to the right side. We need to do this `n - 1` times to cover all items.

The `swap` function is implemented as follows:

```
/**
 * Swap array elements in place
 * Runtime: O(1)
 * @param {array} array to be modified
 * @param {integer} from index of the first element
 * @param {integer} to index of the 2nd element
 */
function swap(array, from, to) {
  // ES6 array destructing
  [array[from], array[to]] = [array[to], array[from]];
}
```

It uses JavaScript ES6 destructing arrays.

JavaScript Array destructuring

Assignment separate from declaration

A variable can be assigned to its values using the destructing syntax.

```
let a, b;

[a, b] = [1, 2];
console.log(a); //↪ 1
console.log(b); //↪ 2
```

Swapping variables

Two variables' values can be swapped in one line using the destructuring expression.

```
[a, b] = [b, a];
console.log(a); //↪ 2
console.log(b); //↪ 1
```

Without the destructuring assignment, swapping two values requires a temporary variable.

Bubble sort has a Quadratic running time, as you might infer from the nested for-loop.

Bubble Sort Properties

- Stable: ✓ Yes

- In-place: ✓ Yes
- Online: ✓ Yes
- Adaptive: ✓ Yes, *O(n)* when already sorted
- Time Complexity: ⛔ Quadratic $O(n^2)$
- Space Complexity: ✓ Constant *O(1)*

4.1.3. Insertion Sort

Insertion sort is a simple sorting algorithm. It is one of the most natural ways of sorting. If I give you some cards to sort, you will probably use this algorithm without knowing.

Insertion Sort Implementation

Insertion sort does the following: It starts from the 2nd element, and it tries to find anything to the left that could be bigger than the current item. It will swap all the elements with higher value and insert the current element where it belongs.

Insertion sort

```
/**
 * Sorting by insertion - Look for bigger numbers on the left side
 *
 * It starts from the 2nd element,
 * and it tries to find any element (to the left)
 * that could be bigger than the current index.
 * It will move all the elements that are bigger
 * and insert the current element where it belongs.
 *
 * Runtime: O(n^2)
 * @param {Array|Set} collection elements to be sorted
 */
function insertionSort(collection) {
  const array = Array.from(collection); ①

  for (let right = 1; right < array.length; right++) { ②
    for (let left = right; array[left - 1] > array[left]; left--) { ③
      swap(array, left - 1, left); ④
    }
  }
  return array;
}
```

① Convert to an array or clone the array.

② Start with the 2nd element. Everything on the left is considered sorted.

③ Compare the current element (2nd) to the previous one. If `left - 1` is bigger, it will swap places. If not, it will continue checking the next one to the left.

④ We check every element on the left side and swap any of them that are out of order

Insertion Sort Properties

- Stable: ✓ Yes
- In-place: ✓ Yes

- Online: ✓ Yes
- Adaptive: ✓ Yes
- Time Complexity: ⛔ Quadratic $O(n^2)$
- Space Complexity: ✓ Constant $O(1)$

4.1.4. Selection Sort

The selection sort is a simple sorting algorithm. As its name indicates, it *selects* the lowest element from the list and moves it where it should be.

Selection sort algorithm

1. Start with the element in position 0.
2. Find the minimum item in the rest of the array. If a new minimum is found, swap them.
3. Repeat step #1 and #2 with the next element until the last one.

Selection sort implementation

For implementing the selection sort, we need two indexes.

Selection sort

```
/**
 * Selection sort - Look for smaller numbers on the right side
 *
 * It starts from the first element (index 0),
 * and tries to find any element (to the right)
 * that could be smaller than the current index.
 * If it finds one, it will swap the positions.
 *
 * Runtime: O(n^2)
 * @param {Array|Set} collection elements to be sorted
 */
function selectionSort(collection) {
  const array = Array.from(collection); ①

  for (let left = 0; left < array.length; left++) { ②
    let selection = left; ③

    for (let right = left + 1; right < array.length; right++) {
      if (array[selection] > array[right]) { ④
        selection = right; ⑤
      }
    }

    if (selection !== left) {
      swap(array, selection, left); ⑥
    }
  }

  return array;
}
```

① Converts any collection to an array or clone existing array.

② Visit all elements in the array starting from the 1st element (index 0).

③ Everything to the left side is considered sorted in its final position. So, select `left` as the initial minimum value.

④ Compare the `selection` to every element to the right side.

⑤ If it finds a value *smaller* than the selection, then update the `selection`.

⑥ Put the next smallest item to its final position

Selection sort minimizes the number of swaps. It does one swap per iteration while insertion sort and bubble sort could swap many times with the same array.

One index is for the position in question (selection/left) and another for finding the minimum in the rest of the array (right).

Selection Sort Properties

- In-place: ✓ Yes
- Stable: ✗ No
- Adaptive: ✗ No
- Online: ✗ No
- Time Complexity: ⛔ Quadratic $O(n^2)$
- Space Complexity: ✓ Constant $O(1)$

Why selection sort is not stable?

To recap, *stable* means that items with the same value keep their initial position. Let's see what would happen with the selection sort if we (select) sort the following array `2, 5, 2, 1`. To distinguish them, let's say `2a` and `2b`, so `2a, 5, 2b, 1`.

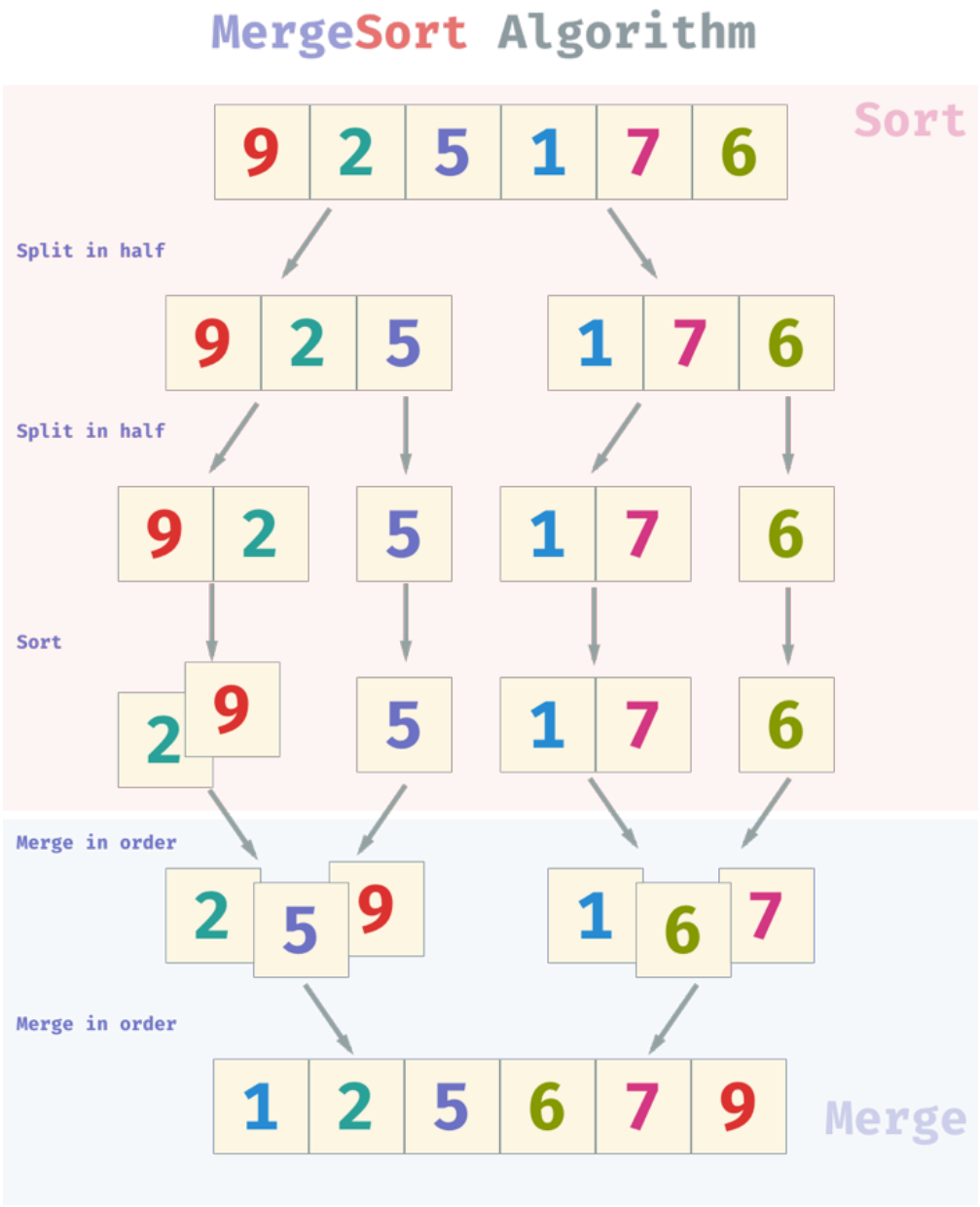
Initially, we select the first element, `2a` and check if there's anything less than 2 in the array. We find out that position 3 has an item with a smaller value (1), so we swap them.

Now, we have: `1, 5, 2b, 2a`. There you have it, `2b` now comes before `2a`.

4.1.5. Merge Sort

Merge Sort is an efficient sorting algorithm that uses divide and conquer paradigm to accomplish its task faster. However, It uses auxiliary memory in the process of sorting.

Merge sort algorithm splits the array into halves until two or fewer elements are left. It sorts these two elements and then merges back all halves until the whole collection is sorted.



Merge Sort Implementation

Merge Sort implementation in JavaScript (mergeSort)

```
/**
 * Merge sort
 * Runtime: O(n log n)
 * @param {Array|Set} collection elements to be sorted
 */
function mergeSort(collection) {
  const array = Array.from(collection); ①
  return splitSort(array);
}
```

① Convert any kind of iterable (array, sets, etc.) into an array

As you can see, this function is just a wrapper to transform things into an array. The heavy lifting is done in `splitSort`, as you can see below.

Merge Sort implementation in JavaScript (splitSort)

```
/**
 * Split array in half recursively until two or less elements are left.
 * Sort these two elements and combine them back using the merge
function.
 * @param {Array} array
 * @example
 *    splitSort([3, 2, 1]) => [1, 2, 3]
 *    splitSort([3]) => [3]
 *    splitSort([3, 2]) => [2, 3]
 */
function splitSort(array) {
  const size = array.length;
  // base case
  if (size < 2) {
    return array;
  } if (size === 2) {
    return array[0] < array[1] ? array : [array[1], array[0]]; ①
  }

  // recursive split in half and merge back
  const half = Math.ceil(size / 2);
  return merge( ③
    splitSort(array.slice(0, half)), ②
    splitSort(array.slice(half)), ②
  );
}
```

① Base case: Sort two or less items manually.

② Recursively divide the array in half until two or fewer elements are left.

③ Merge back the sorted halves in ascending order.

Let's take a look at the merge function:

Merge Sort implementation in JavaScript (merge)

```
/**
 * Merge two arrays in ascending order
 *
 * @param {array} array1 sorted array 1
 * @param {array} array2 sorted array 2
 * @returns {array} merged arrays in asc order
 *
 * @example
 *    merge([2,5,9], [1,6,7]) => [1, 2, 5, 6, 7, 9]
 */
function merge(array1, array2 = []) {
  const mergedLength = array1.length + array2.length;
  const mergedArray = Array(mergedLength);

  // merge elements on a and b in asc order. Run-time O(a + b)
  for (let index = 0, i1 = 0, i2 = 0;
    index < mergedLength; index++) { ①
    if (i2 >= array2.length
      || (i1 < array1.length && array1[i1] <= array2[i2])) {
      mergedArray[index] = array1[i1]; ②
      i1 += 1;
    } else {
      mergedArray[index] = array2[i2]; ②
      i2 += 1;
    }
  }

  return mergedArray; ③
}
```

① We need to keep track of 3 arrays indices: `index` which keeps track of the combined array position, `i1` which is the `array1` index and `i2` for `array2`.

② If `array1` current element (`i1`) has the lowest value, we insert it into the `mergedArray`. If not we then insert the `array2` element.

③ `mergedArray` is `array1` and `array2` combined in ascending order (sorted).

Merge sort has an *O(n log n)* running time. For more details about how to extract the runtime, go to

Linearithmic section.

Merge Sort Properties

- Stable: ✓ Yes
- In-place: ✗ No, it requires auxiliary memory O(n).
- Online: ✗ No, new elements will require to sort the whole array.
- Adaptive: ✗ No, mostly sorted array takes the same time O(n log n).
- Recursive: Yes
- Time Complexity: ✓ Linearithmic *O(n log n)*
- Space Complexity: ⚠ Linear *O(n)*, use auxiliary memory

4.1.6. Quicksort

Quicksort is an efficient recursive sorting algorithm that uses divide and conquer paradigm to sort faster. It can be implemented in-place, so it doesn't require additional memory.

In practice, quicksort outperforms other sorting algorithms like Merge Sort. And, of course, It also outperforms simple sorting algorithms like Selection Sort, Insertion Sort and Insertion Sort.

Quicksort picks a "pivot" element randomly and moves all the smaller parts than the pivot to the right and the ones that are bigger to the left. It does this recursively until all the array is sorted.

Quicksort Implementation

Quicksort implementation uses the divide-and-conquer in the following way:

Quicksort Algorithm

1. Pick a "pivot" element (at random).
2. Move everything lower than the pivot to the left, and everything more significant than the pivot to the right.
3. Recursively repeat step #1 and #2 in the sub-arrays on the left and on the right WITHOUT including the pivot.

Let's convert these words into code!

Quicksort implementation in JavaScript (QuickSort)

```
/**
 * QuickSort - Efficient in-place recursive sorting algorithm.
 * Avg. Runtime: O(n log n) | Worst: O(n^2)
 * @param {Number[]} array
 * @param {Number} low
 * @param {Number} high
 */
function quickSort(array, low = 0, high = array.length - 1) {
  if (low < high) { ④
    const partitionIndex = partition(array, low, high); ①
    quickSort(array, low, partitionIndex - 1); ②
    quickSort(array, partitionIndex + 1, high); ③
  }
  return array;
}
```

① Partition: picks a pivot and find the index where the pivot will be when the array is sorted.

② Do the partition of the sub-array at the left of the pivot.

③ Do the partition of the sub-array at the right of the pivot.

④ Only do the partition when there's something to divide.

The `partition` function does the real heavy lifting. 🏋️‍♀️

Quicksort implementation in JavaScript (partition)

```
/**
 * Linear-time Partitioning
 * Chooses a pivot and re-arrage the array that
 *  everything on the left is <= pivot and
 *  everything on the right is > pivot
 *
 * Runtime: O(n)
 * @param {*} array
 * @param {*} low start index
 * @param {*} high end index
 * @returns {integer} pivot index
 */
function partition(array, low, high) {
  const pivotIndex = low; ①
  let pivotFinalIndex = pivotIndex; ②

  for (let current = pivotIndex + 1; current <= high; current++) {
    if (array[current] < array[pivotIndex]) { ③
      pivotFinalIndex += 1; ④
      swap(array, current, pivotFinalIndex); ⑤
    }
  }

  swap(array, pivotIndex, pivotFinalIndex); ⑥
  return pivotFinalIndex;
}
```

① Take the leftmost element as the pivot.

② `pivotFinalIndex` is the placeholder for the final position where the pivot will be placed when the array is sorted.

③ Check all values other than the pivot to see if any value is smaller than the pivot.

④ If the `current` element's value is less than the pivot, then increment `pivotFinalIndex` to make room on the left side.

⑤ We also swap the smaller element to the left side since it's smaller than the pivot.

⑥ Finally, we move the pivot to its final position. Everything to the left is smaller than the pivot, and everything to the right is bigger.

What would happen if use Quicksort for an array in reverse order?

E.g. `[10, 7, 5, 4, 2, 1]`, if we always choose the first element as the pivot, we would have to swap everything to the left of `10`.

So in the first partition, we would have `[7, 5, 4, 2, 1, 10]`. Then, we take `7` would be the next pivot, and we have to swap everything to the left. Descending arrays are the worst-case for this Quicksort since it will perform $O(n^2)$ work. If we do it by the middle (or even better at random) instead of partitioning by the first element, we would have better performance. That's why we usually shuffle the array before doing Quicksort to avoid edge cases.

```
/**
 * Quick sort
 * Runtime: O(n log n)
 * @param {Array|Set} collection elements to be sorted
 * @returns {Array} sorted array
 */
function quickSortWrapper(collection) {
  const array = Array.from(collection); ①
  shuffle(array); ②
  return quickSort(array);
}
```

① Convert to array (or clone array). If you want to modify the input, directly remove this line.

② Shuffle array to avoid edge cases (desc order arrays)

And you can see the implementation of `shuffle` below:

Shuffling an array

```
/**
 * Shuffle items in an array in-place
 * Runtime: O(n)
 * @param {*} array
 */
function shuffle(array) {
  const { length } = array;
  for (let index = 0; index < length; index++) {
    const newIndex = Math.floor(Math.random() * length);
    swap(array, index, newIndex);
  }
  return array;
}
```

With the optimization, Quicksort has an *O(n log n)* running time. Similar to the merge sort, we divide the array into halves each time. For more details about how to extract the runtime, go to

Linearithmic.

Quicksort Properties

- Stable: ✗ No
- In-place: ✓ Yes
- Adaptive: ✗ No, mostly sorted array takes the same time O(n log n).
- Online: ✗ No, the pivot element can be choose at random.
- Recursive: Yes
- Time Complexity: ✓ Linearithmic *O(n log n)*
- Space Complexity: ✓ Logarithmic *O(log n)*, because of recursion.

4.1.7. Summary

We explored the most common sorting algorithms, some of which are simple and others more performant. Also, we cover the properties of sorting algorithms such as stable, in-place, online, and adaptive.

Table 18. Sorting algorithms comparison

Algorithms	Comments
Bubble Sort	Swap pairs bubbling up largest numbers to the right
Insertion Sort	Look for biggest number to the left and swap it with current
Selection Sort	Iterate array looking for smallest value to the right
Merge Sort	Split numbers in pairs, sort pairs and join them in ascending order
Quicksort	Choose a pivot, set smaller values to the left and bigger to the right.

Table 19. Sorting algorithms time/space complexity and properties

Algorithms	Avg	Best	Worst	Space	Stable	In-place	Online	Adaptive
Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes	Yes	Yes	Yes
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes	Yes	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No	Yes	No	No
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	No	No	No
Quicksort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No	Yes	No	No

4.1.8. Practice Questions

Merge Intervals

so-1) *Given an array of intervals* `[start, end]`*, merge all overlapping intervals.*

Common in interviews at: Facebook, Amazon, Bloomberg.

Starter code:

```
/**
 * Merge overlapping intervals.
 * @param {[numer, number][]} intervals - Array with pairs [start, end]
 * @returns {[numer, number][]} - Array of merged pairs [start, end]
 */
function merge(intervals) {
  // write your code here...
}
```

Examples:

```
merge([[0, 2], [2, 4]]); // [[0, 4]] (0-2 overlaps with 2-4)
merge([[2, 2], [3, 4]]); // [[2, 2], [3, 4]] (no overlap)
merge([[1, 10], [3, 4]]); // [[1, 10]] (1-10 covers the other)
```

Solution: Merge Intervals

Sort Colors (The Dutch flag problem)

so-2) *Given an array with three possible values (0, 1, 2), sort them in linear time, and in-place. Hint: similar to quicksort, where the pivot is 1.*

Common in interviews at: Amazon, Microsoft, Facebook.

Starter code:

```
/**
 * Sort array of 0's, 1's and 2's in linear time and in-place.
 * @param {numbers[]} nums - Array of number (0, 1, or 2).
 * @returns {void} Don't return anything, modify the input array.
 */
function sortColors(nums) {
  // write your code here...
}
```

Examples:

```
sortColors([0, 2, 1]); // [0, 1, 2]
sortColors([2, 0, 2, 1, 0, 1, 0]); // [0, 0, 0, 1, 1, 2, 2]
sortColors([1, 1, 1]); // [1, 1, 1]
```

Solution: Sort Colors (The Dutch flag problem)

4.2. Divide and Conquer

Divide and conquer is a strategy for solving algorithmic problems. It splits the input into manageable parts recursively and finally joins solved pieces to form the solution.

We have already implemented some algorithms using the divide and conquer technique.

Examples of divide and conquer algorithms:

- Merge Sort: **divides** the input into pairs, sort them, and them **join** all the pieces in ascending order.
- Quicksort: **splits** the data by a random number called "pivot," then move everything smaller than the pivot to the left and anything more significant to the right. Repeat the process on the left and right sides. Note: since this works in place doesn't need a "join" part.
- Binary Search: find a value in a sorted collection by **splitting** the data in half until it sees the value.
- Permutations: **Take out** the first element from the input and solve permutation for the remainder of the data recursively, then **join** results and append the items that were taken out.

In general, the divide and conquer algorithms have the following pattern:

1. **Divide** data into subproblems.
2. **Conquer** each subproblem.
3. **Combine** results.

As you might know, there are multiple ways to solve a problem. Let's solve the Fibonacci numbers using a divide and conquer algorithm. Later we are going to provide a more performant solution using dynamic programming.

4.2.1. Recursive Fibonacci Numbers

To illustrate how we can solve a problem using divide and conquer, let's write a program to find the n-th Fibonacci number.

Fibonacci Numbers

Fibonacci sequence is a series of numbers that starts with `0, 1`; the following values are calculated as the sum of the previous two. So, we have:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

We can get the n-th Fibonacci number with the following recursive program:

Recursive Fibonacci implemenation

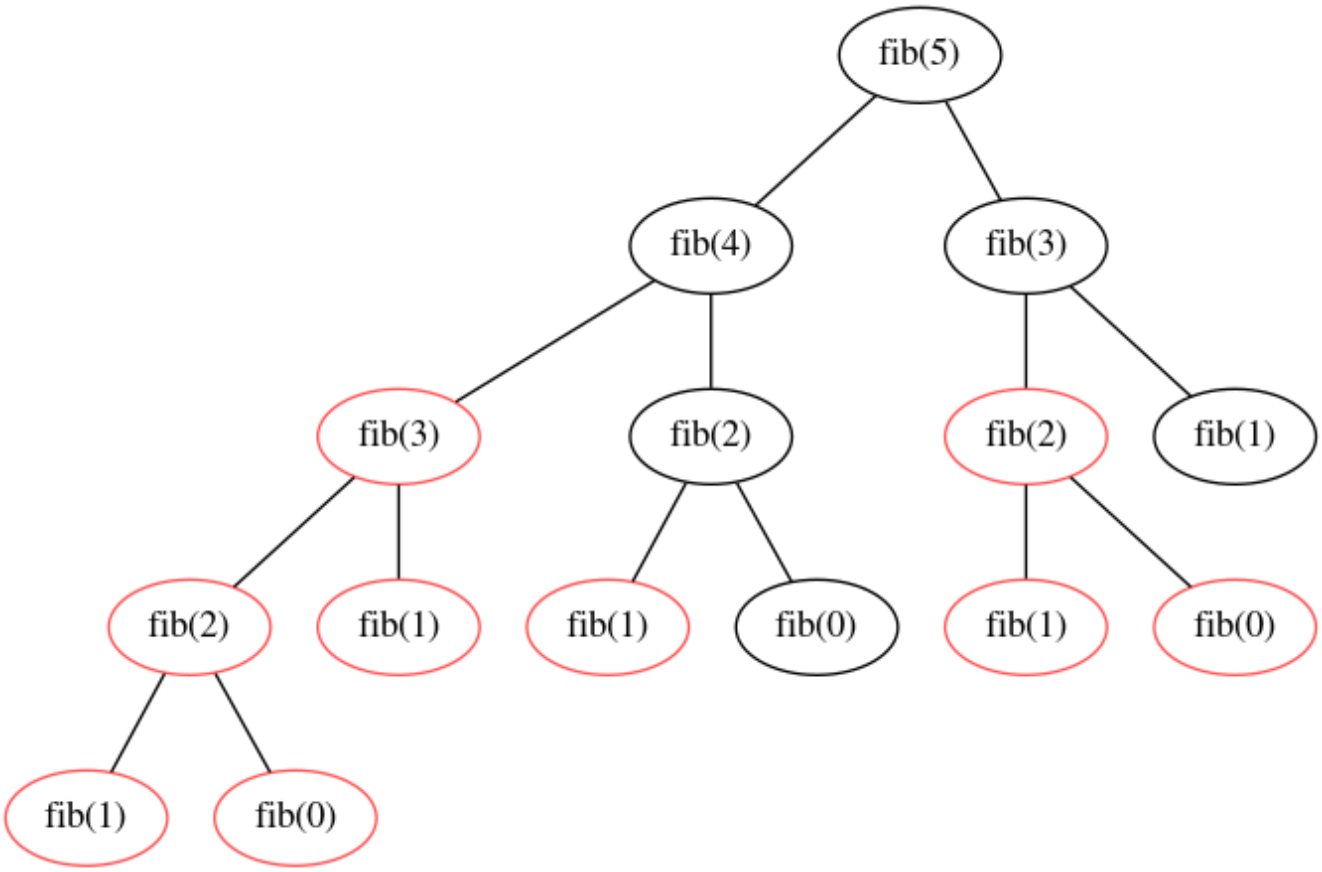
```
/**
 * Get Fibonacci number on the n-th position.
 * @param {integer} n position on the sequence
 * @returns {integer} n-th number
 */
function fib(n) {
  if (n < 0) return 0;
  if (n < 2) return n;

  return fib(n - 1) + fib(n - 2);
}
```

Let's see how this fit divide and conquer:

- **Divide**: `n` is divided in two subproblems `f(n-1)` and `f(n-2)`.
- **Conquer**: solve each subproblem independently
- **Combine**: sum the subproblem results to get the final solution.

The implementation above does the job, but what's the runtime?

For that, let's take a look at the job performed by calculating the `fib(5)` number. Since `fib(5) = fib(4) + fib(3)`, we need to find the answer for `fib(4)` and `fib(3)`. We do that recursively until we reach the base cases of `fib(1)` and `fib(0)`. If we represent the calls in a tree, we would have the following:



In the diagram, we see the two recursive calls needed to compute each number. So if we follow the *O(branches^{depth})*, we get O(2^n). 🐢 NOTE: Fibonacci is not a perfect binary tree since some nodes only have one child instead of two. The exact runtime for recursive Fibonacci is *O(1.6^n)* (still exponential time complexity).

Exponential time complexity is pretty bad. Can we do better?

You can notice every element in red, and with asterisks *, it's called more than once in the call tree. We are repeating calculations too many times!

> Those who cannot remember the past are condemned to repeat it.
>
> — Dynamic Programming

For these cases, when subproblems repeat themselves, we can optimize them using dynamic programming. Let's do that in the next section.

4.3. Dynamic Programming

Dynamic programming (DP) is a way to solve algorithmic problems with **overlapping subproblems**. Algorithms using DP find the base case and building a solution from the ground-up. Dp *keep track* of previous results to avoid re-computing the same operations.

How to explain dynamic programming to kids? 🧒

Write down 1+1+1+1+1+1+1+1+1+1

— What's that equal to?

— **Kid counting one by one** Ten!

— Add another "+1". What's the total now?

— **Quickly** Eleven!

— Why you get the result so quickly? Ah, you got it faster by adding one to the memorized previous answer. So Dynamic Programming is a fancy way of saying: "remembering past solutions to save time later."

4.3.1. Fibonacci Sequence with Dynamic Programming

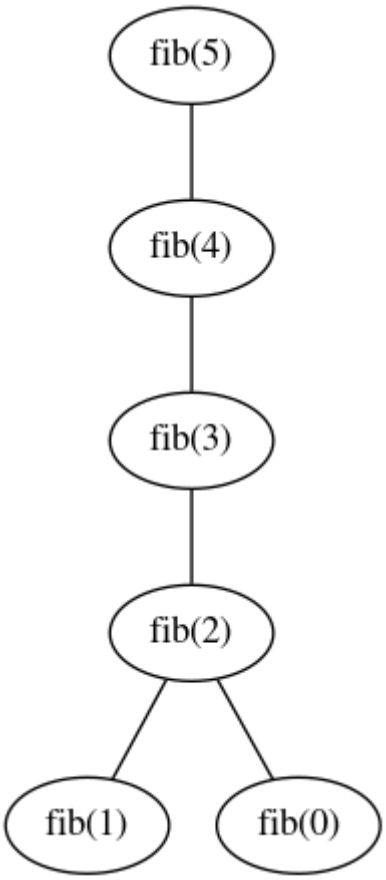
Let's solve the same Fibonacci problem but this time with dynamic programming.

When we have recursive functions, doing duplicated work is the perfect place for dynamic programming optimization. We can save (or cache) the results of previous operations and speed up future computations.

Recursive Fibonacci Implemenation using Dynamic Programming

```
/**
 * Get Fibonacci number on the n-th position.
 * @param {integer} n position on the sequence
 * @param {Map} memo cache of previous solutions
 * @returns {integer} n-th number
 */
function fib(n, memo = new Map()) {
  if (n < 0) return 0;
  if (n < 2) return n;

  if (memo.has(n)) {
    return memo.get(n);
  }

  const result = fib(n - 1) + fib(n - 2);
  memo.set(n, result);

  return result;
}
```

This implementation checks if we already calculated the value, if so it will save it for later use.



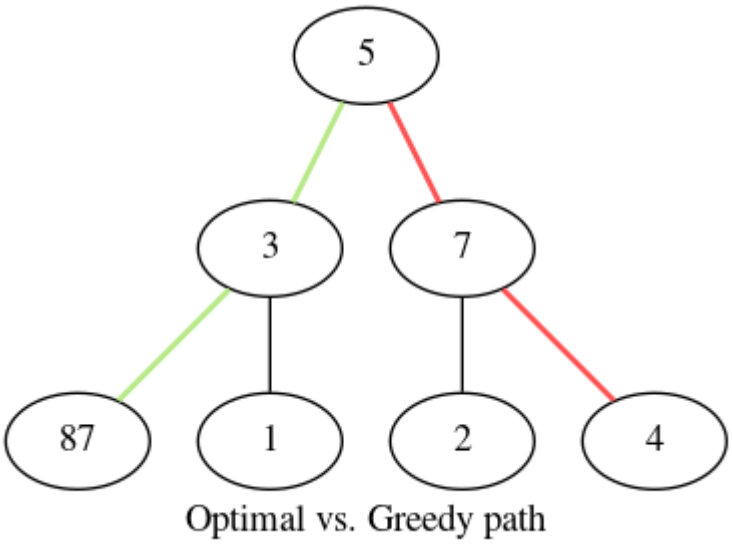
This graph looks pretty linear now. It's runtime *O(n)*!

TIP: Saving previous results for later is a technique called "memoization". This is very common to optimize recursive algorithms with overlapping subproblems. It can make exponential algorithms linear!

4.4. Greedy Algorithms

Greedy algorithms are designed to find a solution by going one step at a time and using heuristics to determine the best choice. They are quick but not always lead to the most optimum results since it might not take into consideration all the options to give a solution.

An excellent example of a greedy algorithm that doesn't work well is finding the largest sum on a tree.



Optimal vs. Greedy path

Let's say the greedy heuristics are set to take the more significant value. The greedy algorithm will start at the root and say, "Which number is bigger 3 or 7?" Then go with 7 and later 4. As you can see in the diagram, the most significant sum would be the path `7 - 3 - 87`. A greedy algorithm never goes back on its options. This greedy choice makes it different from dynamic programming, which is exhaustive and guaranteed to find the best option. However, when they work well, they are usually faster than other options.

Greedy algorithms are well suited when an optimal local solution is also a globally optimal solution.

Greedy algorithms make the choice that looks best at the moment based on a heuristic, such as the smallest, largest, best ratio, and so on. This algorithm only gives one shot at finding the solution and never goes back to consider other options.

Don't get the wrong idea; some greedy algorithms work very well if they are designed correctly.

Some examples of greedy algorithms that work well:

- Selection Sort: we select the best (minimum value) remove it from the input and then select the next minimum until everything is processed.
- Merge Sort: the "merge" uses a greedy algorithm, where it combines two sorted arrays by looking at their current values and choosing the best (minimum) at every time.

In general, we can follow these steps to design Greedy Algorithms:

1. Take a sample from the input data (usually in a data structure like array/list, tree, graph).
2. Greedy choice: use a heuristic function that will choose the best candidate. E.g., Largest/smallest number, best ratio, etc.
3. Reduce the processed input and repeat step #1 and #2 until all data is gone.
4. Return solution.
5. Check correctness with different examples and edge cases.

4.4.1. Fractional Knapsack Problem

We are going to use the "Fractional Knapsack Problem" to learn how to design greedy algorithms. The problem is the following:

> You are going to resell legumes (rice, beans, chickpeas, lentils), and you only brought a knapsack. What proportion of items can you choose to get the highest loot without exceeding the bag's maximum weight?

Let's say we have the following items available.

Knpasack Input

```
const items = [
  { value: 1, weight: 1},
  { value: 4, weight: 3 },
  { value: 5, weight: 4 },
  { value: 7, weight: 5 },
];

const maxWeight = 7;
```

So, we have four items that we can choose from. We can't take them all because the total weight is 13 and the maximum we can carry is 7. We can't just take the first one because with value 1 because it is not the best profit.

How would you solve this problem?

First, we have to define what parameters we will use to make our **greedy choice**. This some ideas:

- We can take items with the **largest** value in hopes to maximize profit. Based on that, we can take the last and the first to have a total weight of 7 and a total cost of 8.

We could also take items with the **smallest** weight so we can fit as much as possible in the knapsack. Let's analyze both options. So we can choose the first two items for a total value of 5 and a total weight of 4. This option is worse than picking the most significant amount! 👎

- One last idea, we can take items based on the **best** value/weight ratio and take fractions of an article to fill up the knapsack to maximum weight. In that case, we can buy the last item in full and 2/3 of the 2nd item. We get a total value of `9.67` and a total weight of `7`. These heuristics seem to be the most profitable. 👍

Items value/weight ratio

```
{ value: 1, weight: 1 }, // 1/1  = 1
{ value: 4, weight: 3 }, // 4/3 = 1.33 ✓
{ value: 5, weight: 4 }, // 5/4 = 1.25
{ value: 7, weight: 5 }, // 7/5 = 1.4 ✓
```

Let's implement this algorithm!

Factional Knapsack Problem Implementation

```
/**
 * Solves Bounded Knapsack Problem (BKP)
 *  You can take fractions or whole part of items.
 * @param {Array} input array of objects with the shape {value, weight}
 * @param {Number} max maximum weight for knapsack
 */
function solveFractionalKnapsack(input, max) {
  let weight = 0;
  let value = 0;
  const items = [];

  // sort by value/weight ratio
  input.sort((a, b) => a.value / a.weight - b.value / b.weight);

  while (input.length && weight < max) {
    const bestRatioItem = input.pop();

    if (weight + bestRatioItem.weight <= max) {
      bestRatioItem.proportion = 1; // take item as a whole
    } else { // take a fraction of the item
      bestRatioItem.proportion = (max - weight) / bestRatioItem.weight;
    }

    items.push(bestRatioItem);
    weight += bestRatioItem.proportion * bestRatioItem.weight;
    value += bestRatioItem.proportion * bestRatioItem.value;
  }

  return { weight, value, items };
}
```

What's the runtime of this algorithm?

We have to sort the array based on the value/weight ratio. Sorting runtime is O(n log n). The rest is linear operations, so the answer is *O(n log n)* for our greedy algorithm.

4.5. Backtracking

Backtracking algorithms are used to find **all (or some)** solutions that satisfy a constraint.

Backtracking builds a solution step by step, using recursion. If during the process it realizes a given path is not going to lead to a solution, it stops and steps back (backtracks) to try another alternative.

Some examples that use backtracking is solving Sudoku/crosswords puzzle and graph operations.

Listing all possible solutions might sound like brute force. However, it is not the same. Backtracking algorithms are faster because it tests if a path will lead to a solution or not.

Brute Force vs. Backtracking Algorithms

Brute force evaluates every possibility. **Backtracking** is an optimized brute force. It stops evaluating a path as soon as some of the conditions are broken. However, it can only be applied if a quick test can be run to tell if a candidate will contribute to a correct solution.

4.5.1. How to develop backtracking algorithms?

Backtracking algorithms can be tricky to get right or reason about, but we will follow this recipe to make it easier.

Steps to create backtracking algorithms

1. Iterate through the given input
2. Make a change
3. Recursively move to the next element.
4. Test if the current change is a possible solution
5. Revert the change (backtracking) and try with the next item

Let's do an exercise to explain better how backtracking works.

4.5.2. Permutations

> Return all the permutations (without repetitions) of a word.

For instace, if you are given the word `art` these are the possible permutations:

```
[ [ 'art' ],
  [ 'atr' ],
  [ 'rat' ],
  [ 'rta' ],
  [ 'tra' ],
  [ 'tar' ] ]
```

Now, let's implement the program to generate all permutations of a word.

We already solved this problem using an iterative program, now let's do it using backtracking.

Word permutations using backtracking

```
/**
 * Find all permutations (without duplicates) of a word/array
 *
 * @param {String|Array} word given string or array
 * @param {Array} solution (used by recursion) array with solutions
 * @param {Number} start (used by recursion) index to start
 * @returns {String[]} all permutations
 *
 * @example
 *  permutations('ab') // ['ab', 'ba']
 *  permutations([1, 2, 3]) // ['123', '132', '213', '231', '321',
'312']
 */
function permutations(word = '', solution = [], start = 0) {
  const array = Array.isArray(word) ? word : Array.from(word);

  if (start === array.length - 1) { ④
    solution.push(array.join(''));
  } else {
    for (let index = start; index < array.length; index++) { ①
      swap(array, start, index); ②
      permutations(array, solution, start + 1); ③
      swap(array, start, index); // backtrack ⑤
    }
  }

  return solution;
}
```

① Iterate through all the elements

② Make a change: swap letters

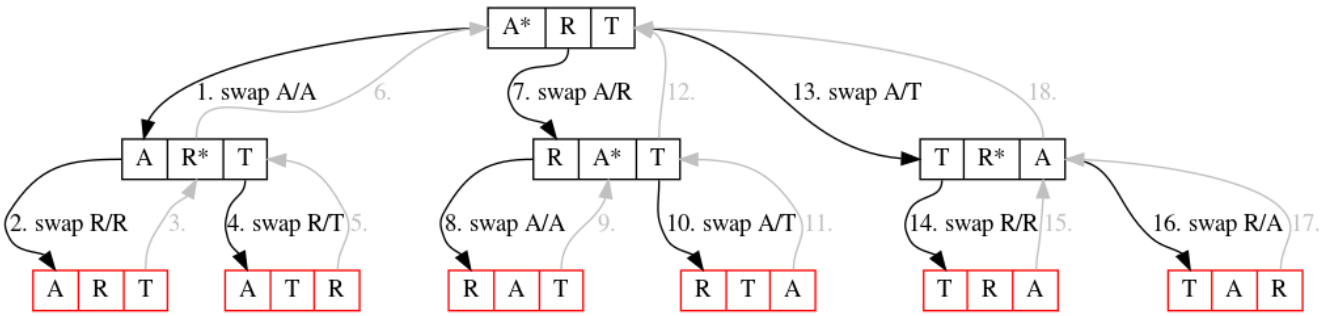
③ Recursive function moving to the next element

④ Test if the current change is a solution: reached the end of the string.

⑤ Revert the change (backtracking): Undo swap from step 2

As you can see, we iterate through each element and swap with the following letters until we reach the end of the string. Then, we roll back the change and try another path.

In the following tree, you can visualize how the backtracking algorithm is swapping the letters. We are taking `art` as an example.



Legend:

- The asterisk (`*`) indicates `start` index.
- **Black** arrows indicate the `swap` operations.
- **Grey** arrows indicate the *backtracking* operation (undo swap).
- The red words are the iterations added to the solution array.

Most of the backtracking algorithms do something similar. What changes is the test function or base case to determine if a current iteration is a solution or not.

4.6. Algorithmic Toolbox

Have you ever given a programming problem and freeze without knowing where to start? Well, in this section, we are going to give some tips so you don't get stuck while coding.

TL;DR: Don't start coding right away. First, solve the problem, then write the code. Make it work first; make it better later.

Steps to solve algorithmic problems

1. **Understand** the requirements. Reframe it in your own words.
2. Draw a **simple example** (no edge cases yet)
3. **Brainstorm** possible solutions
    a. How would you solve this problem **manually**? (without a computer) Is there any formula or theorem you can use?
    b. Is there any heuristics (largest, smallest, best ratio), or can you spot a pattern to solve this problem using a greedy algorithm?
    c. Can you address the simple base case and generalize for other cases using a **recursive solution**? Can you divide the problem into subproblems? Try Divide and Conquer.
    d. Do you have to generate multiple solutions or try different paths? Try Backtracking.
    e. List all the data structures that you know that might solve this problem.
    f. If anything else fails, how would you solve it the dumbest way possible (brute force). We can optimize it later.
4. **Test** your algorithm idea with multiple examples
5. **Optimize** the solution –Only optimize when you have something working. Don't try to do both at the same time!
    a. Can you trade-off space for speed? Use a Map to speed up results!
    b. Do you have a bunch of recursive and overlapping problems? Try Dynamic Programming.
    c. Re-read requirements and see if you can take advantage of anything. E.g. is the array sorted?
6. **Write Code**, yes, now you can code.
    a. Modularize your code with functions (don't do it all in one giant function, please 🙈)
    b. Comment down edge cases but don't address until the basic cases are working.
7. **Test** your code.
    a. Choose a typical input and test against your code.
    b. Brainstorm about edge cases (empty, null values, overflows, largest supported inputs)
    c. How would your code scale beyond the current boundaries?

These steps should get you going even with the most challenging algorithmic problems.

Stay effective!

Appendix A: Cheatsheet

This section summerize what we are going to cover in the rest of this book.

A.1. Runtimes

Table 20. Most common algorithmic running times and their examples

Big O Notation	Name	example (s)
O(1)	Constant	Finding if an array is empty
O(log n)	Logarithmic	Searching on a sorted array
O(n)	Linear	Finding duplicates in an array using a map
O(n log n)	Linearithmic	Sorting elements in an array
$O(n^2)$	Quadratic	Finding duplicates in an array (naïve approach)
$O(n^3)$	Cubic	3 Sum
$O(2^n)$	Exponential	Finding subsets of a set
O(n!)	Factorial	Getting all permutations of a word

Table 21. How long an algorithm takes to run based on their time complexity and input size

Input Size	O(1)	O(n)	O(n log n)	$O(n^2)$	$O(2^n)$	O(n!)
1	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
10	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	4 seconds
10k	< 1 sec.	< 1 sec.	< 1 sec.	2 minutes	∞	∞
100k	< 1 sec.	< 1 sec.	1 second	3 hours	∞	∞
1M	< 1 sec.	1 second	20 seconds	12 days	∞	∞

A.2. Linear Data Structures

Table 22. Time/Space Complexity of Linear Data Structures (Array, LinkedList, Stack & Queues)

Data Structure	Searching By		Inserting at the			Deleting from			Space
	Index/Key	*Value*	*beginning*	*middle*	*end*	*beginning*	*middle*	*end*	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)
Singly Linked List	O(n)	O(n)	O(1)	O(n)	**O(n)**	O(1)	O(n)	**O(n)**	O(n)
Doubly Linked List	O(n)	O(n)	O(1)	O(n)	**O(1)**	O(1)	O(n)	**O(1)**	O(n)
Stack	-	-	-	-	O(1)	-	-	O(1)	O(n)
Queue (w/array)	-	-	-	-	O(1)	**O(n)**	-	-	O(n)
Queue (w/list)	-	-	-	-	O(1)	**O(1)**	-	-	O(n)

A.3. Trees and Maps Data Structures

This section covers Binary Search Tree (BST) time complexity (Big O).

Table 23. Time and Space Complexity for Graph-based Data Structures

Data Structure	Searching By		Insert	Delete	Space Complexity
	Index/Key	*Value*			
BST (unbalanced)	-	O(n)	O(n)	O(n)	O(n)
BST (balanced)	-	O(log n)	O(log n)	O(log n)	O(n)
Hash Map (naïve)	O(n)	O(n)	O(n)	O(n)	O(n)
HashMap (optimized)	O(1)	O(n)	O(1)*	O(1)	O(n)
TreeMap (Red-Black Tree)	O(log n)	O(n)	O(log n)	O(log n)	O(n)
HashSet	O(1)	-	O(1)*	O(1)	O(n)
TreeSet	O(log n)	-	O(log n)	O(log n)	O(n)

* = Amortized run time. E.g. rehashing might affect run time to **O(n)**.

Table 24. Time complexity for a Graph data structure

Data Structure	Vertices		Edges		Space Complexity
	Add	*Remove*	*Add*	*Remove*	
Graph (adj. matrix)	$O(\|V\|^2)$	$O(\|V\|^2)$	O(1)	O(1)	$O(\|V\|^2)$
Graph (adj. list w/array)	O(1)	O(|V| + |E|))	O(1)	O(|E|)	O(|V| + |E|)
Graph (adj. list w/HashSet)	O(1)	O(|V|))	O(1)	O(1)	O(|V| + |E|)

A.4. Sorting Algorithms

Table 25. Sorting algorithms comparison

Algorithms	Comments
Bubble Sort	Swap pairs bubbling up largest numbers to the right
Insertion Sort	Look for biggest number to the left and swap it with current
Selection Sort	Iterate array looking for smallest value to the right
Merge Sort	Split numbers in pairs, sort pairs and join them in ascending order
Quicksort	Choose a pivot, set smaller values to the left and bigger to the right.

Table 26. Sorting algorithms time/space complexity and properties

Algorithms	Avg	Best	Worst	Space	Stable	In-place	Online	Adaptive
Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes	Yes	Yes	Yes
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes	Yes	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No	Yes	No	No

Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	No	No	No
Quicksort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No	Yes	No	No

Appendix B: Self-balancing Binary Search Trees

Binary Search Trees (BST) are an excellent data structure to find elements very fast *O(log n)*. However, when the BST branches have different branch sizes, then the performance suffers. In the worst case, all nodes can go to one side (e.g., right) and then the search time would be linear. At this point searching element won't be any better on that tree than an array or linked list. Yikes!

Self-balanced trees will automatically rebalance the tree when an element is inserted to keep search performance. We balance a tree by making the height (distance from a node to the root) of any leaf on the tree as similar as possible.

From unbalanced BST to balanced BST

In the example above: - Unbalanced BST: height node 3 is 2 and height node 2 is 1. - Balanced BST: height node 3 is 1 and height node 2 is 1. Much better!

As you might notice, we balanced the tree in the example by doing a rotation. To be more specific we rotated node 1 to the left to balance the tree. Let's examine all the possible rotation we can do to balance a tree.

B.1. Tree Rotations

We can do single rotations left and right and also we can do double rotations. Let's go one by one.

B.1.1. Single Right Rotation

Right rotation moves a node on the right as a child of another node.

Take a look at the examples in the code in the next section. As you will see we have an unbalanced tree `4-3-2-1`. We want to balance the tree, for that we need to do a right rotation of node 3. So, we move node 3 as the right child of the previous child.

Single right rotation implementation

```
/**
 * Single Right Rotation (RR Rotation)
* @example: #1 rotate node 3 to the right
 *
 *       4                                    4
 *      /                                    /
 *     3*                                   2
 *    /                                    /  \
 *   2    ---| right-rotation(3) |-->  1    3*
 *  /
 * 1
 *
 * @example: #2 rotate 16 to the right and preserve nodes
 *       64                                        64
 *       /                                         /
 *      16*                                       4
 *      / \                                      / \
 *     4  32    -- right-rotation(16) -->       2   16
 *    / \                                      /    / \
 *   2   8                                    1    8   32
 *  /
 * 1
 *
 * @param {TreeNode} node
 *    this is the node we want to rotate to the right. (E.g., node 16)
 * @returns {TreeNode} new parent after the rotation (E.g., node 4)
 */
function rightRotation(node) {
  const newParent = node.left; // E.g., node 4
  const grandparent = node.parent; // E.g., node 64
  const previousRight = newParent.right; // E.g., node 8

  // swap node 64's left children (node 16) with node 4 (newParent).
  swapParentChild(node, newParent, grandparent);

  // update node 4's right child to be node 16,
  // also make node 4 the new parent of node 16.
  newParent.setRightAndUpdateParent(node);
  // Update 16's left child to be the `previousRight` node.
  node.setLeftAndUpdateParent(previousRight);

  return newParent;
}
```

In the `rightRotation` we identify 3 nodes:

- `node` this is the node we want to rotate to the right. E.g., `node 3`
- `newParent` this is the new parent after the rotation. E.g., `node 2`
- `grandparent` this the current's node parent. E.g. `node 4`.

The `swapParentChild` as it name says, swap the children. For our example, it swaps `node 4`'s left children from `node 3` to `node 2`.

Take a look at the implementation.

Swap Parent and Child Implementation

```
/**
 * Swap parent's child
 *
 * @example Child on the left side (it also work for the right side)
 *
 * p = parent
 * o = old child
 * n = new child
 *
 * p           p
 *  \    =>     \
 *   o           n
 *
 * @param {TreeNode} oldChild current child's parent
 * @param {TreeNode} newChild new child's parent
 * @param {TreeNode} parent parent
 */
function swapParentChild(oldChild, newChild, parent) {
  if (parent) {
    // this set parent child
    const side = oldChild.isParentRightChild ? 'Right' : 'Left';
    parent[`set${side}AndUpdateParent`](newChild);
  } else {
    // no parent? so set it to null
    newChild.parent = null;
  }
}
```

After `swapParentChild`, we have the following:

```
      4
     /
    2 - 3*
  /
 1
```

Still not quite what we want. So, `newParent.setRightAndUpdateParent(node)` will make `node 3` the right child of `node 2`. Finally, we remove the left child of `node 3` to be `null`.

```
      4
     /
    2
  /   \
 1     3*
```

Check out the rightRotation implementation again. It should make more sense to you now.

This rotation is also known as `RR rotation`.

B.1.2. Single Left Rotation

Left rotation is similar to the `rightRotation` we explained above.

Single left rotation implementation

```
/**
 * Single Left Rotation (LL Rotation)
 *
 * @example: #1 tree with values 1-2-3-4
 *
 * 1                                  1
 *  \                                  \
 *   2*                                  3
 *    \    --left-rotation(2)->        /  \
 *     3                             2*    4
 *      \
 *       4
 *
 * @example: #2 left rotation
 *
 *     1                                          1
 *      \                                          \
 *       4*                                          16
 *      / \                                        / \
 *    2    16    -- left-rotation(4) ->           4   32
 *       /  \                                    / \    \
 *      8   32                                 2   8    64
 *            \
 *            64
 * @param {TreeNode} node current node to rotate (e.g. 4)
 * @returns {TreeNode} new parent after the rotation (e.g. 16)
 */
function leftRotation(node) {
  const newParent = node.right; // E.g., node 16
  const grandparent = node.parent; // E.g., node 1
  const previousLeft = newParent.left; // E.g., node 8

  // swap parent of node 4 from node 1 to node 16
  swapParentChild(node, newParent, grandparent);

  // Update node 16 left child to be 4, and
  // updates node 4 parent to be node 16 (instead of 1).
  newParent.setLeftAndUpdateParent(node);
  // set node4 right child to be previousLeft (node 8)
  node.setRightAndUpdateParent(previousLeft);

  return newParent;
}
```

As you can see, this function is just the opposite of `rightRotation`. Where ever we used the right now we use the left here and vice versa. This rotation is also known as `LL rotation`.

If you are curious about the `setRightAndUpdateParent` and `setLeftAndUpdateParent`. Here's the implementation:

Set and update parent implementation

```
/**
 * Set a left node descendants.
 * Also, children get associated to parent.
 */
setLeftAndUpdateParent(node) {
  this.left = node;
  if (node) {
    node.parent = this;
    node.parentSide = LEFT;
  }
}

/**
 * Set a right node descendants.
 * Also, children get associated to parent.
 */
setRightAndUpdateParent(node) {
  this.right = node;
  if (node) {
    node.parent = this;
    node.parentSide = RIGHT;
  }
}
```

You can also check out the full binary tree node implementation.

B.1.3. Left Right Rotation

This time are we going to do a double rotation.

Left-Right rotation implementation

```
/**
 * Left Right Rotation (LR Rotation)
 *
 * @example LR rotation on node 3
 *        4                        4
 *      /                        /                            4
 *    3                        3*                           /
 *  /                        /                            2
 * 1*   --left-rotation(1)->  2   --right-rotation(3)->  /  \
 *  \                        /                          1    3*
 *   2                      1
 *
 * @param {TreeNode} node
 *    this is the node we want to rotate to the right. E.g., node 3
 * @returns {TreeNode} new parent after the rotation
 */
function leftRightRotation(node) {
  leftRotation(node.left);
  return rightRotation(node);
}
```

As you can see we do a left and then a right rotation. This rotation is also known as `LR rotation`

B.1.4. Right Left Rotation

Very similar to `leftRightRotation`. The difference is that we rotate right and then left.

Right-Left rotation implementation

```
/**
 * Right Left Rotation (RL Rotation)
 *
 * @example RL rotation on 1
 *
 *   1*                            1*
 *    \                             \                              2
 *      3   -right-rotation(3)->    2   -left-rotation(1)->  /  \
 *    /                             \                      1*   3
 *   2                               3
 *
 * @param {TreeNode} node
 * @returns {TreeNode} new parent after the rotation
 */
function rightLeftRotation(node) {
  rightRotation(node.right);
  return leftRotation(node);
}
```

This rotation is also referred to as `RL rotation`.

B.2. Self-balancing trees implementations

So far, we have study how to make tree rotations which are the basis for self-balancing trees. There are different implementations of self-balancing trees such a Red-Black Tree and AVL Tree.

Appendix C: AVL Tree

AVL Tree is named after their inventors (**A**delson-**V**elsky and **L**andis). This self-balancing tree keeps track of subtree sizes to know if a rebalance is needed or not. We can compare the size of the left and right subtrees using a balance factor.

The **balanced factor** on each node is calculated recursively as follows:

```
Balance Factor = (left subtree height) - (right subtree
height)
```

The implementation will go in the BST node class. We will need two methods to calculate the left and right subtree, and with those, we can get the balance factor.

Balance Factor methods on the BST node

```
/**
 * @returns {Number} left subtree height or 0 if no left child
 */
get leftSubtreeHeight() {
  return this.left ? this.left.height + 1 : 0;
}

/**
 * @returns {Number} right subtree height or 0 if no right child
 */
get rightSubtreeHeight() {
  return this.right ? this.right.height + 1 : 0;
}

/**
 * Get the max height of the subtrees.
 *
 * It recursively goes into each children calculating the height
 *
 * Height: distance from the deepest leaf to this node
 */
get height() {
  return Math.max(this.leftSubtreeHeight, this.rightSubtreeHeight);
}

/**
 * Returns the difference the heights on the left and right subtrees
 */
get balanceFactor() {
  return this.leftSubtreeHeight - this.rightSubtreeHeight;
}
```

C.1. Implementing AVL Tree

Implementing an AVL Tree is not too hard since it builds upon what we did in the Binary Search Tree.

AVL Tree class

```
/**
 * AVL Tree
 * It's a self-balanced binary search tree optimized for fast lookups.
 */
class AvlTree extends BinarySearchTree {
  /**
   * Add node to tree. It self-balance itself.
   * @param {any} value node's value
   */
  add(value) {
    const node = super.add(value);
    this.root = balanceUpstream(node);
    return node;
  }

  /**
   * Remove node if it exists and re-balance tree
   * @param {any} value
   */
  remove(value) {
    const node = super.find(value);
    if (node) {
      const found = super.remove(value);
      this.root = balanceUpstream(node.parent);
      return found;
    }

    return false;
  }
}
```

As you can see, the AVL tree inherits from the BST class. The insert and remove operations work the same as in the BST, except that at the end we call `balanceUpstream`. This function checks if the tree is symmetrical after every change to the tree. If the tree went out of balance, it would execute the appropriated rotation to fix it.

Balance Upstream for AVL tree

```
/**
 * Bubbles up balancing nodes a their parents
 *
 * @param {BinaryTreeNode} node
 */
function balanceUpstream(node) {
  let current = node;
  let newParent;
  while (current) {
    newParent = balance(current);
    current = current.parent;
  }
  return newParent;
}
```

This function recursively goes from the modified node to the root checking if each node in between is balanced. Now, let's examine how does the balancing works on AVL tree.

Balance method for AVL tree

```
/**
 * Balance tree doing rotations based on balance factor.
 *
 * Depending on the `node` balance factor and child's factor
 * one of this rotation is performed:
 * - LL rotations: single left rotation
 * - RR rotations: single right rotation
 * - LR rotations: double rotation left-right
 * - RL rotations: double rotation right-left
 *
 * @param {BinaryTreeNode} node
 */
function balance(node) {
  if (node.balanceFactor > 1) {
    // left subtree is higher than right subtree
    if (node.left.balanceFactor < 0) {
      return leftRightRotation(node);
    }
    return rightRotation(node);
  } if (node.balanceFactor < -1) {
    // right subtree is higher than left subtree
    if (node.right.balanceFactor > 0) {
      return rightLeftRotation(node);
    }
    return leftRotation(node);
  }
  return node;
}
```

The first thing we do is to see if one subtree is longer than the other. If so, then we check the children balance to determine if need a single or double rotation and in which direction.

You can review Tree Rotations in case you want a refresher.

Appendix D: Interview Questions Solutions

D.1. Solutions for Array Questions

D.1.1. Max Subarray

AR-1) *Given an array of integers, find the maximum sum of consecutive elements (subarray).*

Examples:

```
maxSubArray([1, -3, 10, -5]); // 10 (taking only 10)
maxSubArray([-3, 4,-1, 2, 1, -5]); // 6 (sum [4,-1, 2, 1])
maxSubArray([-2, 1, -3, 4, -1, 3, 1]); // 7 (sum [4,-1, 3, 1])
```

Common in interviews at: FAANG, Microsoft

The first step is making sure we understand the problem well. Let's do basic examples:

```
A = [-5, 6, 9, -8]
B = [-1, 6, -3, 8]
```

What's the subarray with the maximum sum? For A, it will be `[6, 9]` and for B, it will be `[6, -3, 8]`.

We could generate all possible subarrays, add them up, and then pick the max number.

```
function maxSubArrayBrute1(nums) {
  let max = -Infinity;

  for (let i = 0; i < nums.length; i++) { // O(n^3)
    for (let j = i + 1; j <= nums.length; j++) { // O(n^2)
      const sum = nums.slice(i, j).reduce((a, n) => n + a, 0); // O(n)
      max = Math.max(max, sum); // O(1)
    }
  }

  return max;
}
```

This code is simple to understand; however, not very efficient. The runtime is `O(n^3)`.

Notice we're adding up the numbers from `i` to `j` on each cycle. But, we can optimize this. We can keep a local variable and add the new number to it. That way, we don't have to revisit previous numbers.

```
function maxSubArrayBrute2(nums) {
  let max = -Infinity;

  for (let i = 0; i < nums.length; i++) { // O(n) * O(n)
    let local = 0;
    for (let j = i; j < nums.length; j++) { // O(n)
      local += nums[j];
      max = Math.max(max, local);
    }
  }
  return max;
}
```

The runtime is much better: `O(n)`. Can we still do better?

We can use a greedy approach, where do one pass through the array. We only add the numbers if their sum is larger than just taking the current element.

```
/**
 * Find the maximun sum of contiguous elements in an array.
 *
 * @examples
 *  maxSubArray([1, -3, 10, -5]); // => 10
 *  maxSubArray([-3,4,-1,2,1,-5]); // => 6
 *
 * @param {number[]} a - Array
 * @returns {number} - max sum
 */
function maxSubArray(a) {
  let max = -Infinity;
  let local = 0;

  a.forEach((n) => {
    local = Math.max(n, local + n);
    max = Math.max(max, local);
  });

  return max;
}
```

The runtime is `O(n)` and the space complexity of `O(1)`.

D.1.2. Best Time to Buy and Sell a Stock

AR-2) *You have an array of integers. Each value represents the closing value of the stock on that day. You have only one chance to buy and then sell. What's the maximum profit you can obtain? (Note: you have to buy first and then sell)*

Examples:

```
maxProfit([1, 2, 3]) // 2 (buying at 1 and selling at 3)
maxProfit([3, 2, 1]) // 2 (no buys)
maxProfit([5, 10, 5, 10]) // 5 (buying at 5 and selling at 10)
```

Common in interviews at: Amazon, Facebook, Bloomberg

There are multiple examples that we can simulate: bear markets (when prices are going down), bullish markets (when prices are going up), and zig-zag markets (when prices are going up and down).

```
// zig-zag market
maxProfit([5, 10, 5, 10]); // => 5
// bullish market
maxProfit([1, 2, 3]); // => 2
// bearish market
maxProfit([3, 2, 1]); // => 0
```

During the bearish markets, the profit will always be 0. Since if you buy, we are only going to lose.

We can do a brute force solution doing all combinations:

```
function maxProfitBrute1(prices) {
  let max = 0;
  for (let i = 0; i < prices.length; i++) {
    for (let j = i + 1; j < prices.length; j++) {
      max = Math.max(max, prices[j] - prices[i]);
    }
  }
  return max;
}
```

The runtime for this solution is `O(n^2)`.

A better solution is to eliminate the 2nd for loop and only do one pass.

Algorithm:

- Do one pass through all the prices
  - Keep track of the minimum price seen so far.
  - calculate `profit = currentPrice - minPriceSoFar`
  - Keep track of the maximun profit seen so far.
- Return maxProfit.

```
/**
 * Find the max profit from buying and selling a stock given their daily
prices.
 * @examples
 *    maxProfit([5, 10, 5, 10]); // => 5
 *    maxProfit([1, 2, 3]); // => 2
 *    maxProfit([3, 2, 1]); // => 0
 * @param {number[]} prices - Array with daily stock prices
 * @returns {number} - Max profit
 */
function maxProfit(prices) {
  let max = 0;
  let local = Infinity;
  for (let i = 0; i < prices.length; i++) {
    local = Math.min(local, prices[i]);
    max = Math.max(max, prices[i] - local);
  }
  return max;
}
```

The runtime is `O(n)` and the space complexity of `O(1)`.

D.2. Solutions for Linked List Questions

D.2.1. Merge Linked Lists into One

LL-1) *Merge two sorted lists into one (and keep them sorted)*

Examples:

```
mergeTwoLists(2->3->4, 1->2); // 1->2->2->3->4
mergeTwoLists(2->3->4,null); // 2->3->4
```

Common in interviews at: FAANG, Adobe, Microsoft

We need to visit each node in both lists and merge them in ascending order. Note: We don't need to copy the values nor create new nodes.

Another case to take into consideration is that lists might have different lengths. So, if one list runs out, we have to keep taking elements from the remaining list.

Algorithm:

- Have a pointer for each list
- While there's a pointer that is not null, visite them
  - Compare each list node's value and take the smaller one.
  - Advance the pointer of the taken node to the next one.

Implementation:

```
/**
 * Given two sorted linked lists merge them while keeping the asc order.
 * @examples
 *    mergeTwoLists([2,4,6], [1,3]); // => [1,2,3,4,6]
 *    mergeTwoLists([2,4,6], []); // => [2,4,6]
 *    mergeTwoLists([], [1,3]); // => [1,3]
 *
 * @param {ListNode} l1 - The root node of list 1
 * @param {ListNode} l2 - The root node of list 2
 * @returns {ListNode} - The root of the merged list.
 */
function mergeTwoLists(l1, l2) {
  const sentinel = new ListNode();
  let p0 = sentinel;
  let p1 = l1;
  let p2 = l2;

  while (p1 || p2) {
    if (!p1 || (p2 && p1.value > p2.value)) {
      p0.next = p2;
      p2 = p2.next;
    } else {
      p0.next = p1;
      p1 = p1.next;
    }
    p0 = p0.next;
  }

  return sentinel.next;
}
```

Notice that we used a "dummy" node or "sentinel node" to have some starting point for the final list.

Complexity Analysis:

- Time: `O(m+n)`. Visiting each node from the list 1 and list 2 has a time complexity `O(m + n)`. `m` and `n` represent each list's length.

- Space: `O(1)`. We reuse the same nodes and only change their `next` pointers. We only create one additional node, "the sentinel node."

D.2.2. Check if two strings lists are the same

LL-2) *Given two linked lists with strings, check if the data is equivalent.*

Examples:

```
hasSameData(he->ll->o, hel->lo); // true
hasSameData(hello, hel->lo); // true
hasSameData(he->ll->o, h->i); // false
```

Common in interviews at: Facebook

We are given two linked lists that contain string data. We want to know if the concatenated strings from each list are the same.

The tricky part is that the same data can be distributed differently on the linked lists:

```
L1: he -> ll -> o
L2: h -> e -> llo
```

One naive approach could be to go through each list's node and concatenate the strings. Then, we can check if they are equal.

```
function hasSameDataBrute1(l1, l2) {
  function toString(node) {
    const str = [];
    for (let curr = node; curr; curr = curr.next) {
      str.push(curr.value);
    }
    return str.join('');
  }

  // console.log({s1: toString(l1), s2: toString(l2) });
  return toString(l1) === toString(l2);
}
```

Notice that the problem mentions that lists could be huge (millions of nodes). If the first character on each list is different, we are unnecessarily computing millions of nodes, when a straightforward check will do the job.

A better way to solve this problem is iterating over each character on both lists, and when we found a mismatch, we return `false` immediately. If they are the same, we still have to visit all of them.

Algorithm:

- Set a pointer to iterate over each node in the lists.
- For each node, have an index (starting at zero) and compare if both lists have the same data.
    - When the index reaches the last character on the current node, we move to the next node.
    - If we found that a character from one list doesn't match the other, we return `false`.

Implementation:

```
/**
 * Check if two lists has the same string data.
 * Note: each lists can be huge, they have up to 10 million nodes.
 *
 * @examples
 *    hasSameData(['he', 'll', 'o'], ['hel', 'lo']); // true
 *    hasSameData(['hel', 'lo'], ['hi']); // false
 *
 * @param {ListNode} l1 - The root node of list 1.
 * @param {ListNode} l2 - The root node of list 2.
 * @returns {boolean} - true if has same data, false otherwise.
 */
function hasSameData(l1, l2) {
  let p1 = l1;
  let p2 = l2;
  let i1 = -1;
  let i2 = -1;

  const findNextPointerIndex = (p, i) => {
    let node = p;
    let index = i;
    while (node && index >= node.value.length) {
      node = node.next;
      index = 0;
    }
    return [node, index];
  };

  while (p1 && p2) {
    [p1, i1] = findNextPointerIndex(p1, i1 + 1);
    [p2, i2] = findNextPointerIndex(p2, i2 + 1);
    if ((p1 && p2 && p1.value[i1] !== p2.value[i2])
      || ((!p1 || !p2) && p1 !== p2)) return false;
  }
  return true;
}
```

The function `findNextPointerIndex` is a helper to navigate each character on a linked list. Notice that we increase the index (`i + 1`) on each iteration. If the index overflows, it moves to the next node and reset the index to zero.

Complexity Analysis:

- Time: `O(n)`. We go over all the characters on each list
- Space: `O(1)`. Only using pointers and no auxiliary data structures.

D.3. Solutions for Stack Questions

D.3.1. Validate Parentheses / Braces / Brackets

ST-1) *Given a string with three types of brackets:* `()`, `{}`, *and* `[]`. *Validate they are correctly closed and opened.*

Examples:

```
isParenthesesValid('(){}[]'); // true
isParenthesesValid('('); // false (closing parentheses is missing)
isParenthesesValid('([{}])'); // true
isParenthesesValid('[{]}'); // false (brakets are not closed in the
right order)
isParenthesesValid('([{)}]'); // false (closing is out of order)
```

Common in interviews at: Amazon, Bloomberg, Facebook, Citadel

We need to validate that brackets are correctly opened and closed, following these rules:

- An opened bracket must be close by the same type.
- Open brackets mush be closed in the correct order.

We are facing a parsing problem, and usually, stacks are good candidates for them.

Algorithm:

- Create a mapping for each opening bracket to its closing counterpart.
- Iterate through the string
  - When we found an opening bracket, insert the corresponding closing bracket into the stack.
  - When we found a closing bracket, pop from the stack and make sure it corresponds to the current character.
- Check the stack is empty. If there's a leftover, it means that something didn't close properly.

Implementation:

```
/**
 * Validate if the parentheses are opened and closed in the right order.
 *
 * @examples
 *  isParenthesesValid('(){}[]'); // true
 *  isParenthesesValid('([{}])'); // true
 *  isParenthesesValid('([{)}]'); // false
 *
 * @param {string} string - The string
 * @returns {boolean} - True if valid, false otherwise.
 */
function isParenthesesValid(string) {
  const map = new Map([['(', ')'], ['{', '}'], ['[', ']']]);
  const stack = [];

  for (const c of string) {
    if (map.has(c)) stack.push(map.get(c));
    else if (c !== stack.pop()) return false;
  }

  return stack.length === 0;
}
```

Complexity Analysis:

- Time: `O(n)`. We iterate over each character of the string.
- Space: `O(n)`. We use an auxiliary stack.

D.3.2. Daily Temperatures

ST-2) *Given an array of integers from 30 to 100 (daily temperatures), return another array that, for each day in the input, tells you how many days you would have to wait until warmer weather. If no warmer climate is possible, then return* `0` *for that element.*

Examples:

```
dailyTemperatures([30, 28, 50, 40, 30]); // [2 (to 50), 1 (to 28), 0, 0,
0]
dailyTemperatures([73, 69, 72, 76, 73, 100]); // [3, 1, 1, 0, 1, 100]
```

Common in interviews at: Amazon, Adobe, Cisco

The first solution that might come to mind it's using two for loops. For each element, we have visit each temperature ahead to find a bigger one.

```
function dailyTemperaturesBrute1(t) {
  const ans = [];

  for (let i = 0; i < t.length; i++) {
    ans[i] = 0;
    for (let j = i + 1; j < t.length; j++) {
      if (t[j] > t[i]) {
        ans[i] = j - i;
        break;
      }
    }
  }

  return ans;
}
```

This solution is an `O(n^2)`. Can we do better? We can!

Here's an idea: start backward, so we know when there's a warmer temperature beforehand. The last element is always 0 (because there are no more temperatures ahead of it). We can place each element's index that we visit on a stack. If the current weather is bigger than the stack top, we remove it until a bigger one remains or the stack is empty. If the stack has a value, we calculate the number of days ahead. Otherwise, it is 0.

Algorithm:

- Traverse the daily temperatures backward
  - Push each temperature to a stack.
  - While the current temperature is larger than the one at the top of the stack, pop it.
  - If the stack is empty, then there's no warmer weather ahead, so it's 0.
  - If the stack has an element, calculate the index delta.

Implementation:

```
/**
 * Given an array with daily temperatures (30 °C to 100 °C),
 * return an array with the days count until a warmer temperature
 * for each elem from the input.
 *
 * @examples
 *  dailyTemperatures([30, 28, 50, 40, 30]); // [2, 1, 0, 0, 0]
 *  dailyTemperatures([73, 69, 72, 76, 73]); // [3, 1, 1, 0, 0]
 *
 * @param {number[]} t - Daily temperatures
 * @returns {number[]} - Array with count of days to warmer temp.
 */
function dailyTemperatures(t) {
  const last = (arr) => arr[arr.length - 1];
  const stack = [];
  const ans = [];

  for (let i = t.length - 1; i >= 0; i--) {
    while (stack.length && t[i] >= t[last(stack)]) stack.pop();
    ans[i] = stack.length ? last(stack) - i : 0;
    stack.push(i);
  }

  return ans;
}
```

The stack contains the indexes rather than the temperatures themselves.

Complexity Analysis:

- Time: `O(n)`. We visit each element on the array once.
- Space: `O(1)`. The worst-case scenario is ascending order without duplicates. The stack will hold at most 70 items (100 - 30). If we didn't have the range restriction, then space complexity would be `O(n)`.

D.4. Solutions for Queue Questions

D.4.1. Recent Counter

QU-1) *Design a class that counts the most recent requests within a time window.*

Example:

```
const counter = new RecentCounter(10); // The time window is 10 ms.
counter.request(1000); // 1 (first request, it always counts)
counter.request(3000); // 1 (last requests was 2000 ms ago, > 10ms, so
doesn't count)
counter.request(3100); // 1 (last requests was 100 ms ago, > 10ms, so
doesn't count)
counter.request(3105); // 2 (last requests was 5 ms ago, <= 10ms, so it
counts)
```

Common in interviews at: FAANG, Bloomberg, Yandex

We are asked to keep track of the request's count only within a given time window. A queue is a perfect application for this. We can add any new request to the Queue. Also, we need to check if the oldest element is outside the time window. If so, we remove it from the queue.

Algorithm:

- Enqueue new requests.
- Take a `peek` at the oldest request in the queue.
- While `current timestamp - oldest timestamp`, dequeue the oldest.
- Return the length of the queue.

Implementation:

```
/**
 * Counts the most recent requests within a time window.
 * Each request has its timestamp.
 * If the time window is 2 seconds,
 * any requests that happened more than 2 seconds before the most
 * recent request should not count.
 *
 * @example - The time window is 3 sec. (3000 ms)
 *  const counter = new RecentCounter(3000);
 *  counter.request(100); // 1
 *  counter.request(1000); // 2
 *  counter.request(3000); // 3
 *  counter.request(3100); // 4
 *  counter.request(3101); // 4
 *
 */
class RecentCounter {
  /**
   * @param {number} maxWindow - Max. time window (in ms) for counting
requests
   *  Defaults to 1 second (1000 ms)
   */
  constructor(maxWindow = 1000) {
    this.window = maxWindow;
    this.queue = new Queue();
  }

  /**
   * Add new request and calculate the current count within the window.
   * @param {number} timestamp - The current timestamp (increasing
order)
   * @return {number} - The number of requests within the time window.
   */
  request(timestamp) {
    this.queue.enqueue(timestamp);
    while (timestamp - this.queue.peek() > this.window) this.queue
.dequeue();

    return this.queue.size;
  }
}
```

Notice that we enqueue every request, and then we check all the ones that have "expire" and remove them from the queue.

Complexity Analysis:

- Time: `O(n)`, where `n` is the number of requests. One Enqueue/Dequeue operation is O(1). However, we might run into a worst-case where all requests have to be dequeued.
- Space: `O(W)`, where `W` is the time window. We can have at most W requests in the queue since they are in increasing order without duplicates.

D.4.2. Design Snake Game

QU-2) *Design the* `move` *function for the snake game. The move function returns an integer representing the current score. If the snake goes out of the given height and width or hit itself, the game is over and return* `-1`.

Example:

```
const snakeGame = new SnakeGame(4, 2, [[1, 2], [0, 1]]);
expect(snakeGame.move('R')).toEqual(0); //  0
expect(snakeGame.move('D')).toEqual(0); //  0
expect(snakeGame.move('R')).toEqual(1); //  1 (ate food1)
expect(snakeGame.move('U')).toEqual(1); //  1
expect(snakeGame.move('L')).toEqual(2); //  2 (ate food2)
expect(snakeGame.move('U')).toEqual(-1); // -1 (hit wall)
```

Common in interviews at: Amazon, Bloomberg, Apple

This game is perfect to practice working with Queues. There are at least two opportunities to use a Queue. You can enqueue the food location, and also you can keep the snake's body parts on a Queue. We insert a new position into the snake's queue on every move and dequeue the last location to indicate the snake moved. Every time the snake eats food, it grows one more unit. The food gets dequeue, and we place the next food location (if any).

Algorithm:

- Based on the snake's head current position, calculate the next location based on the given move `direction`.
- If the new position is outside the boundaries, it's game over (return -1).
- If the new location has food, remove that eaten food from its queue and place the next food on the map (if any).
- If the new position doesn't have food, remove the tail of the snake since it moved.
- If the snake new position hits itself, game over (return -1). To make this check, we have 2 options:
  - Queue: we can visit all the elements on the snake's queue (body) and check if a new position collides. That's `O(n)`
  - Set: we can maintain a `set` with all the snake locations, so the check is `O(1)`.

- Move the snake's head to a new location (enqueue)
- Return the score (snake's length - 1);

Implementation:

```
/**
 * The snake game stars with a snake of length 1 at postion 0,0.
 * Only one food position is shown at a time. Once it's eaten the next
one shows up.
 * The snake can move in four directions up, down, left and right.
 * If the snake go out of the boundaries (width x height) the game is
over.
 * If the snake hit itself the game is over.
 * When the game is over, the `move` method returns -1 otherwise, return
the current score.
 *
 * @example
 *  const snakeGame = new SnakeGame(3, 2, [[1, 2], [0, 1]]);
 *  snakeGame.move('R'); //  0
 *  snakeGame.move('D'); //  0
 *  snakeGame.move('R'); //  0
 *  snakeGame.move('U'); //  1
 *  snakeGame.move('L'); //  2
 *  snakeGame.move('U'); // -1
 */
class SnakeGame {

  /**
   * Initialize game with grid's dimension and food order.
   * @param {number} width - The screen width (grid's columns)
   * @param {number} height - Screen height (grid's rows)
   * @param {number[]} food - Food locations.
   */
  constructor(width, height, food) {
    this.width = width;
    this.height = height;
    this.food = new Queue(food);
    this.snake = new Queue([[0, 0]]);
    this.tail = new Set([[0, 0]]);
    this.dirs = {
      U: [-1, 0], D: [1, 0], R: [0, 1], L: [0, -1],
    };
  }
  /**
   * Move snake 1 position into the given direction.
```

```
    * It returns the score or game over (-1) if the snake go out of bound
or hit itself.
    * @param {string} direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D'
= Down.
    * @returns {number} - The current score (snake.length - 1).
    */
  move(direction) {
    let [r, c] = this.snake.back(); // head of the snake
    [r, c] = [r + this.dirs[direction][0], c + this.dirs[direction][1]];

    // check wall collision
    if (r < 0 || c < 0 || r >= this.height || c >= this.width) return -
1;

    const [fr, fc] = this.food.front() || []; // peek
    if (r === fr && c === fc) {
      this.food.dequeue(); // remove eaten food.
    } else {
      this.snake.dequeue(); // remove snake's if not food was eaten
      this.tail.delete(this.tail.keys().next().value);
    }

    // check collision with snake's tail
    if (this.tail.has(`${r},${c}`)) return -1; // O(1)

    this.snake.enqueue([r, c]); // add new position
    this.tail.add(`${r},${c}`);

    return this.snake.size - 1; // return score (length of the snake -
1)
  }
}
```

As you can see, we opted for using a set to trade speed for memory.

Complexity Analysis:

- Time: `O(1)`. Insert/Remove from Queue is constant time. Check for body collisions is `O(1)` when using a set. If instead of a set, you traversed the snake's queue to find a collision, it would be `O(n)`. Here `n` is the snake's max length, which is the screen size (height x width).
- Space: `O(n + m)`. `m` is the number of food items, and `n` is the snake's maximum size (height x width).

D.5. Solutions for Binary Tree Questions

D.5.1. Binary Tree Diameter

BT-1) *Find the diameter of a binary tree. A tree's diameter is the longest possible path from two nodes (it doesn't need to include the root). The length of a diameter is calculated by counting the number of edges on the path.*

We are asked to find the longest path on a binary tree that might or might not pass through the root node.

We can calculate the height (distance from root to farthest leaf) of a binary tree using this recursive function:

```
function getHeight(node) {
  if (!node) return 0;
  const leftHeight = getHeight(node.left);
  const rightHeight = getHeight(node.right);
  return 1 + Math.max(leftHeight, rightHeight);
}
```

That function will give us the height from the furthest leaf to the root. However, the problem says that it might or might not go through the root. In that case, we can keep track of the maximum distance (`leftHeight + rightHeight`) seen so far.

Algorithm:

- Initialize diameter to `0`
- Recursively find the height of the tree from the root.
- Keep track of the maximum diameter length seen so far (left + right).
- Return the diameter.

Implementation:

```
/**
 * Find the length of the binary tree diameter.
 *
 * @param {BinaryTreeNode | null} root - Binary Tree's root.
 * @returns {number} tree's diameter (longest possible path on the tree)
 */
function diameterOfBinaryTree(root) {
  let diameter = 0;

  const height = (node) => {
    if (!node) return 0;
    const left = height(node.left);
    const right = height(node.right);
    diameter = Math.max(diameter, left + right);
    return 1 + Math.max(left, right);
  };

  height(root);
  return diameter;
}
```

We are using `Math.max` to keep track of the longest diameter seen.

Complexity Analysis:

- Time: `O(n)`, where `n` is each of the tree nodes. We visite each one once.
- Space: `O(n)`. We use `O(1)` variables, but because we are using the `height` recursive function, we use the implicit call stack, thus `O(n)`.

D.5.2. Binary Tree from right side view

BT-2) *Imagine that you are viewing the tree from the right side. What nodes would you see?*

The first thing that might come to mind when you have to visit a tree, level by level, is BFS. We can visit the tree using a Queue and keep track when a level ends, and the new one starts.

Since during BFS, we dequeue one node and enqueue their two children (left and right), we might have two levels (current and next one). For this problem, we need to know what the last node on the current level is.

There are several ways to solve this problem by using BFS. Here are some ideas:

- **1 Queue + Sentinel node**: we can use a special character in the `Queue` like `'*'` or `null` to indicate a level change. So, we would start something like this `const queue = new Queue([root, '*']);`.
- **2 Queues**: using a "special" character might be seen as hacky, so you can also opt to keep two

queues: one for the current level and another for the next level.

- **1 Queue + size tracking**: we track the Queue's `size` before the children are enqueued. That way, we know where the current level ends.

We are going to implement BFS with "1 Queue + size tracking", since it's arguably the most elegant.

Algorithm:

- Enqueue root
- While the queue has an element
  - Check the current size of the queue
  - Dequeue only `size` times, and for each dequeued node, enqueue their children.
  - Check if the node is the last one in its level and add it to the answer.

Implementation:

```
/**
 * Find the rightmost nodes by level.
 *
 * @example rightSideView(bt([1,2,3,4])); // [1, 3, 4]
 *     1      <- 1
 *   /   \
 *  2     3   <- 3
 *   \
 *    4       <- 4
 *
 * @param {BinaryTreeNode} root - The root of the binary tree.
 * @returns {number[]} - array with the rightmost nodes values.
 */
function rightSideView(root) {
  if (!root) return [];
  const queue = new Queue([root]);
  const ans = [];

  while (queue.size) {
    const { size } = queue;
    for (let i = 0; i < size; i++) {
      const node = queue.dequeue();
      if (i === size - 1) ans.push(node.value);
      if (node.left) queue.enqueue(node.left);
      if (node.right) queue.enqueue(node.right);
    }
  }

  return ans;
}
```

This problem is also possible to be solved using DFS. The trick is to start with the right child and add it to the solution when it is the first one on its level.

```
function rightSideViewDfs(root) {
  const ans = [];

  const dfs = (node, level = 0) => {
    if (!node) return;
    if (level === ans.length) ans.push(node.value);
    dfs(node.right, level + 1); // right side first!
    dfs(node.left, level + 1);
  };

  dfs(root);
  return ans;
}
```

The complexity of any of the BFS methods or DFS is similar.

Complexity Analysis:

- Time: `O(n)`. We visit every node, once.
- Space: `O(n)`. For BFS, the worst-case space is given by the maximum **width**. That is when the binary tree is complete so that the last level would have `(n-1)/2` nodes, thus `O(n)`. For the DFS, the space complexity will be given by the tree's maximum **height**. In the worst-case, the binary tree is skewed to the right so that we will have an implicit call stack of size `n`.

D.6. Solutions for Hash Map Questions

D.6.1. Fit two movies in a flight

HM-1) *You are working in an entertainment recommendation system for an airline. Given a flight duration (target) and an array of movies length, you need to recommend two movies that fit exactly the length of the flight. Return an array with the indices of the two numbers that add up to the target. No duplicates are allowed. If it's not possible to return empty* `[]`.

This simple problem can have many solutions; let's explore some.

Brute force

One brute force approach could be doing two for loops. We sum two different numbers and check if they add up to the target. If yes, we return, and if not, we keep increasing the indices until we check every possible pair.

```
function twoSumBrute(nums, target) {
  for (let i = 0; i < nums.length; i++) {
    for (let j = i + 1; j < nums.length; j++) {
      if (nums[i] + nums[j] === target) return [i, j];
    }
  }

  return [];
}
```

This approach's time complexity is `O(n^2)`, because we visit every number twice in the worst-case. While the space complexity is `O(1)`.

Can we trade space for time? Yes!

Map

Based on `nums[i] + nums[j] === target` we can say that `num[j] === target - nums[i]`. We can do one pass and check if we have seen any number equal to `target - nums[i]`. A map is perfect for this job. We could have a HashMap that maps `num` to `index`. Let's see the algorithms to make it work.

Algorithm:

- Visit every number once
  - Calculate the complement `target - nums[i]`.
  - If the complement exists, return its index and the current index.
  - If not, save the complement and the index number.

Implementation:

```
/**
 * Find two numbers that add up to the target value.
 * Return empty array if not found.
 * @example twoSum([19, 7, 3], 10) // => [1, 2]
 * @param {number[]} nums - Array of integers
 * @param {number} target - The target sum.
 * @returns {[number, number]} - Array with index 1 and index 2
 */
function twoSum(nums, target) {
  const map = new Map();

  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (map.has(nums[i])) return [map.get(nums[i]), i];
    map.set(complement, i);
  }

  return [];
}
```

Complexity Analysis:

- Time: `O(n)`. We visit every number once.
- Space: `O(n)`. In the worst-case scenario, we don't find the target, and we ended up with a map with all the numbers from the array.

D.6.2. Subarray Sum that Equals K

HM-2) *Given an array of integers, find all the possible subarrays to add up to k. Return the count.*

This problem has multiple ways to solve it. Let's explore some.

Brute force

The most straightforward one is to convert the requirements into code: generate all possible subarrays, add them up, and check how many are equal to k.

```
function subarraySumBrute1(nums, k) {
  let ans = 0;

  for (let i = 0; i < nums.length; i++) {
    for (let j = i; j < nums.length; j++) {
      let sum = 0;
      for (let n = i; n <= j; n++) {
        sum += nums[n];
      }
      if (sum === k) ans++;
    }
  }

  return ans;
}
```

This solution's time complexity is `O(n^3)` because of the 3 nested loops.

How can we do better? Notice that the last for loop, compute the sum repeatedly just to add one more. Let's fix that!

Cummulative Sum

For this solution, instead of computing the sum from `i` to `j` all the time. We can calculate a cumulative sum. Every time we see a new number, we add it to the aggregate.

Since we want all possible subarray, We can increase `i` and get sum for each:

```
array = [1, 2, 3, 0, 1, 4, 0, 5];

// cummulative sum from left to right with i = 0
sum = [1, 3, 6, 6, 7, 11, 11, 16];
// cummulative sum from left to right with i = 1
sum = [2, 5, 5, 6, 10, 10, 15];
// cummulative sum from left to right with i = 2
sum = [3, 3, 4, 8, 8, 13];
// ... and so on ...
// cummulative sum from left to right with i = 7
sum = [5];
```

Here's the code:

```
function subarraySumBrute1(nums, k) {
  let ans = 0;

  for (let i = 0; i < nums.length; i++) {
    for (let j = i; j < nums.length; j++) {
      let sum = 0;
      for (let n = i; n <= j; n++) {
        sum += nums[n];
      }
      if (sum === k) ans++;
    }
  }

  return ans;
}
```

The time complexity for this solution is better, `O(n^2)`. Can we still do better?

Map

Let's get the intution from our previous cummulative sum:

```
subarraySum([1, 2, 3, 0, 1, 4, 0, 5], 5); // k = 5

// cummulative sum from left to right is
sum = [1, 3, 6, 6, 7, 11, 11, 16];
//           ^  ^
```

Notice that when the array has a 0, the cumulative sum has a repeated number. If you subtract those numbers, it will give you zero. In the same way, If you take two other ranges and subtract them (`sum[j] - sum[i]`), it will give you the sum of that range `sum(num[i]...num[j])`.

For instance, if we take the index `2` and `0` (with values 6 and 1) and susbtract them we get `6-1=5`. To verify we can add the array values from index 0 to 2, `sum([1, 2, 3]) === 5`.

With that intuition, we can use a Map to keep track of the aggregated sum and the number of times that sum.

Algorithm:

- Start sum at 0
- Visit every number on the array
  - Compute the cumulative sum
  - Check if `sum - k` exits; if so, it means that there's a subarray that adds up to k.

- Save the sum and the number of times that it has occurred.

Implementation:

```
/**
 * Find the number of subarrays that add up to k.
 * @example subarraySum([1, -1, 1], 0); // 3 ([1,-1,1], [1], [1])
 * @param {number[]} nums - Array of integers.
 * @param {number} k - The target sum.
 * @returns {number} - The number of solutions.
 */
function subarraySum(nums, k) {
  let ans = 0;
  let sum = 0;
  const map = new Map([[0, 1]]);

  for (let i = 0; i < nums.length; i++) {
    sum += nums[i];
    if (map.has(sum - k)) ans += map.get(sum - k);
    map.set(sum, 1 + (map.get(sum) || 0));
  }

  return ans;
}
```

You might wonder, what the map is initialized with `[0, 1]`. Consider this test case:

```
subarraySum([1], 1); // k = 1
```

The sum is 1, however `sum - k` is `0`. If it doesn't exist on the map, we will get the wrong answer since that number adds up to `k`. We need to add an initial case on the map: `map.set(0, 1)`. If `nums[i] - k = 0`, then that means that `nums[i] = k` and should be part of the solution.

Complexity Analysis:

- Time: `O(n)`. We visit every number once.
- Space: `O(n)`. The map size will be the same as the original array.

D.7. Solutions for Set Questions

D.7.1. Most common word

ST-1) *Given a text and a list of banned words. Find the most common word that is not on the banned list. You might need to sanitize the text and strip out punctuation '?!,'.* `

This problem requires multiple steps. We can use a `Set` for quickly looking up banned words. For getting the count of each word, we used a `Map`.

Algorithm:

- Convert text to lowercase.
- Remove any special characters `!?',;.`.
- Convert the paragraph into words array.
- Count how many times words occur.
- Exclude banned words from counts.
- Return the word (or first one) that is the most repeated.

Implementation:

```
/**
 * Find the most common word that is not banned.
 * @example mostCommonWord("It's blue and it's round", ['and']) // it
 * @param {string} paragraph - The text to sanitize and search on.
 * @param {string[]} banned - List of banned words (lowercase)
 * @returns {string} - The first word that is the most repeated.
 */
function mostCommonWord(paragraph, banned) {
  const words = paragraph.toLowerCase().replace(/\W+/g, ' ').split(/\s
+/);
  const b = new Set(banned);
  const map = words.reduce((m, w) => (b.has(w) ? m : m.set(w, 1 + (m.
get(w) || 0))), new Map());
  const max = Math.max(...map.values());
  for (const [w, c] of map.entries()) if (c === max) return w;
  return '';
}
```

Here are heavily relying on Regular Expressions:

- `\W+` would match all non-words.
- `\s+` catches all whitespace.

The line that is mapping words to count seems very busy. Here's another version of the same code a little bit more explicit:

```
function mostCommonWordExplicit(paragraph, banned) {
  const words = paragraph
    .toLowerCase()
    .replace(/\W+/g, ' ')
    .split(/\s+/);
  const exclude = new Set(banned);

  const wordsCount = words.reduce((map, word) => {
    if (exclude.has(word)) return map;
    const count = map.get(word) || 0;
    return map.set(word, 1 + count);
  }, new Map());

  const max = Math.max(...wordsCount.values());

  for (const [word, count] of wordsCount.entries()) {
    if (count === max) {
      return word;
    }
  }
  return '';
}
```

Complexity Analysis:

- Time: `O(m + n)`, where `n` is paragraph length and `m` is the number of banned words. If we were NOT using a `Set` for prohibited words, then the runtime would have been `O(mn)`.
- Space: `O(m + n)`. The extra space complexity is given by the size of the `Map` and `Set`.

D.7.2. Longest Without Repeating

ST-2) *Find the length of the longest substring without repeating characters.*

One of the most efficient ways to find repeating characters is using a `Map` or `Set`. Use a `Map` when you need to keep track of the count/index (e.g., string → count) and use a `Set` when you only need to know if there are repeated characters or not.

Algorithm:

- Visit each letter.
  - Insert the letter on a Set.
  - Keep track of the maximum size of the Set in `max`.
  - If the letter has been seen before, delete until there's no duplicate.
- Return max.

Implementation:

```
/**
 * Find the length of the longest substring without duplicates.
 * @example lenLongestSubstring('abccxyz'); // => 4 (cxyz)
 * @param {string} s - The string.
 * @returns {number} - The length of the longest unique substring.
 */
function lenLongestSubstring(s) {
  let max = 0;
  const set = new Set();

  for (let i = 0, j = 0; j < s.length; j++) {
    while (set.has(s[j])) set.delete(s[i++]);
    set.add(s[j]);
    max = Math.max(max, set.size);
  }

  return max;
}
```

We could also have used a Map and keep track of the indexes, but that's not necessary. In this case, the `Set` is all we need.

Complexity Analysis:

- Time: `O(n)`. We visit each letter once.
- Space: `O(W)`, where `W` is the max length of non-repeating characters. The maximum size of the Set gives the space complexity. In the worst-case scenario, all letters are unique (`W = n`), so our space complexity would be `O(n)`. In the avg. case where there are one or more duplicates, it uses less space than `n`, because `W < n`.

D.8. Solutions for Graph Questions

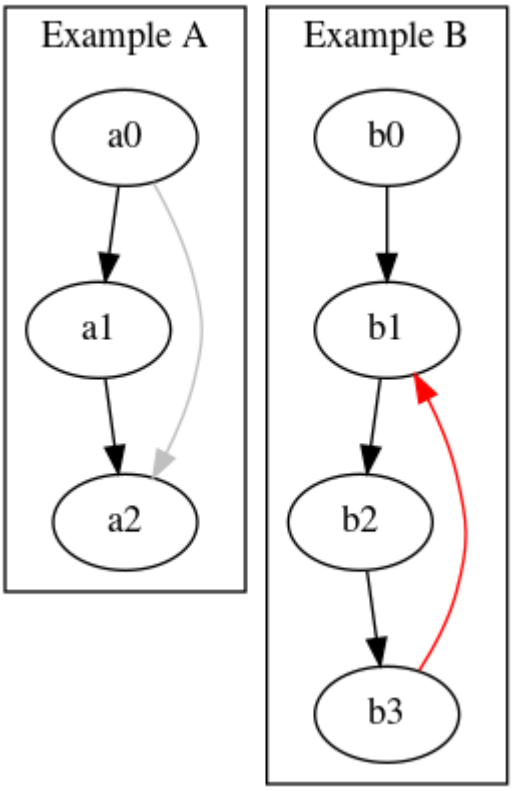
D.8.1. Course Schedule

gr-1) *Check if it's possible to take all courses while satisfying their prerequisites.*

Basically, we have to detect if the graph has a cycle or not. There are multiple ways to detect cycles on a graph using BFS and DFS.

One of the most straightforward ways to do it is using DFS one each course (node) and traverse their prerequisites (neighbors). If we start in a node, and then we see that node again, we found a cycle! (maybe)

A critical part of solving this exercise is coming up with good test cases. Let's examine these two:



Let's say we are using a regular DFS, where we visit the nodes and keep track of visited nodes. If we test the example A, we can get to the course 2 (a2) in two ways. So, we can't blindly assume that "seen" nodes are because of a cycle. To solve this issue, we can keep track of the parent.

For example B, if we start in course 0 (b0), we can find a cycle. However, the cycle does not involve course 0 (parent). When we visit course 1 (b1) and mark it as the parent, we will see that reach to course 1 (b1) again. Then, we found a cycle!

```
function canFinishBrute1(n, prerequisites) {
  const graph = new Map(); // inialize adjacency list as map of arrays
  for (let i = 0; i < n; i++) graph.set(i, []); // build nodes
  prerequisites.forEach(([u, v]) => graph.get(v).push(u)); // edges

  const hasCycles = (node, parent = node, seen = []) => {
    for (const next of graph.get(node)) {
      if (next === parent) return true;
      if (seen[next]) continue;
      seen[next] = true;
      if (hasCycles(next, parent, seen)) return true;
    }
    return false;
  };

  for (let i = 0; i < n; i++) {
    if (hasCycles(i)) return false;
  }

  return true;
}
```

We built the graph on the fly as an adjacency list (Map + Arrays). Then we visited each node, checking if there it has cycles. If none has cyles, then we return true.

The cycle check uses DFS. We keep track of seen nodes and also who the parent is. If we get to the parent more than once, we have a cycle like examples A and B.

What's the time complexity?

We visite every node/vertex: `O(|V|)` and then for every node, we visite all it's edges, so we have `O(|V|*|E|)`.

Can we do better?

There's no need to visit nodes more than once. Instead of having a local `seen` variable for each node, we can move it outside the loop. However, it won't be a boolean anymore (seen or not seen). We could see nodes more than once, without being in a cycle (example A). One idea is to have 3 states: `unvisited` (0), `visiting` (1) and `visited` (2). Let's devise the algorithm:

Algorithm:

- Build a graph as an adjacency list (map + arrays).
- Fill in every prerequisite as an edge on the graph.
- Visit every node and if there's a cycle, return false.

- When we start visiting a node, we mark it as 1 (visiting)
- Visit all its adjacent nodes
- Mark current node as 2 (visited) when we finish visiting neighbors.
- If we see a node in visiting state more than once, it's a cycle!
- If we see a node in a visited state, skip it.

Implementation:

```
/**
 * Check if you can finish all courses with their prerequisites.
 * @param {number} n - The number of courses
 * @param {[number, number][]} prerequisites - Array of courses pairs.
 *  E.g. [[200, 101]], to take course 202 you need course 101 first.
 * @returns {boolean} - True = can finish all courses, False otherwise
 */
function canFinish(n, prerequisites) {
  const graph = new Map(Array(n).fill().map((_, i) => ([i, []])));
  prerequisites.forEach(([u, v]) => graph.get(v).push(u));

  const seen = [];
  const hasCycle = (node) => {
    if (seen[node] === 1) return true; // if visiting, it's a cycle!
    if (seen[node] === 2) return false; // if visited, skip it.

    seen[node] = 1; // mark as visiting.
    for (const adj of graph.get(node)) if (hasCycle(adj)) return true;
    seen[node] = 2; // mark as visited.
    return false;
  };

  for (let i = 0; i < n; i++) if (hasCycle(i)) return false;
  return true;
}
```

In the first line, we initialize the map with the course index and an empty array. This time the `seen` array is outside the recursion.

Complexity Analysis:

- Time: `O(|V| + |E|)`. We go through each node and edge only once.
- Space: `O(|V| + |E|)`. The size of the adjacency list.

D.8.2. Critical Network Paths

gr-2) *Given* `n` *servers and the connections between them, return the critical paths.*

On idea to find if a path is critical is to remove it. If we visit the graph and see that some nodes are not reachable, then, oops, It was critical!

We can code precisely that. We can remove one link at a time and check if all other nodes are reachable. It's not very efficient, but it's a start.

```
function areAllNodesReachable(n, graph) {
  const seen = Array(n).fill(false);
  const queue = new Queue([0]);

  while (queue.size) {
    const node = queue.dequeue();
    if (seen[node]) continue;
    seen[node] = true;

    for (const adj of (graph.get(node) || [])) {
      queue.enqueue(adj);
    }
  }

  return !seen.some((s) => !s);
}

function criticalConnectionsBrute1(n, connections) {
  const critical = [];
  const graph = new Map(Array(n).fill(0).map((_, i) => [i, []]));
  connections.forEach(([u, v]) => {
    graph.get(u).push(v);
    graph.get(v).push(u);
  });

  for (const [u, v] of connections) {
    // remove edge
    graph.set(u, (graph.get(u) || []).filter((e) => e !== v));
    graph.set(v, (graph.get(v) || []).filter((e) => e !== u));

    if (!areAllNodesReachable(n, graph)) critical.push([u, v]);

    // add it back
    graph.get(u).push(v);
    graph.get(v).push(u);
  }

  return critical;
}
```

We are using a function `areAllNodesReachable`, which implements a BFS for visiting the graph, but DFS would have worked too. The runtime is `O(|E| + |V|)`, where `E` is the number of edges and `V` the number of nodes/servers. In `criticalConnectionsBrute1`, We are looping through all `connections` (`E`) to remove one connection at a time and then checking if all servers are still reachable with `areAllNodesReachable`.

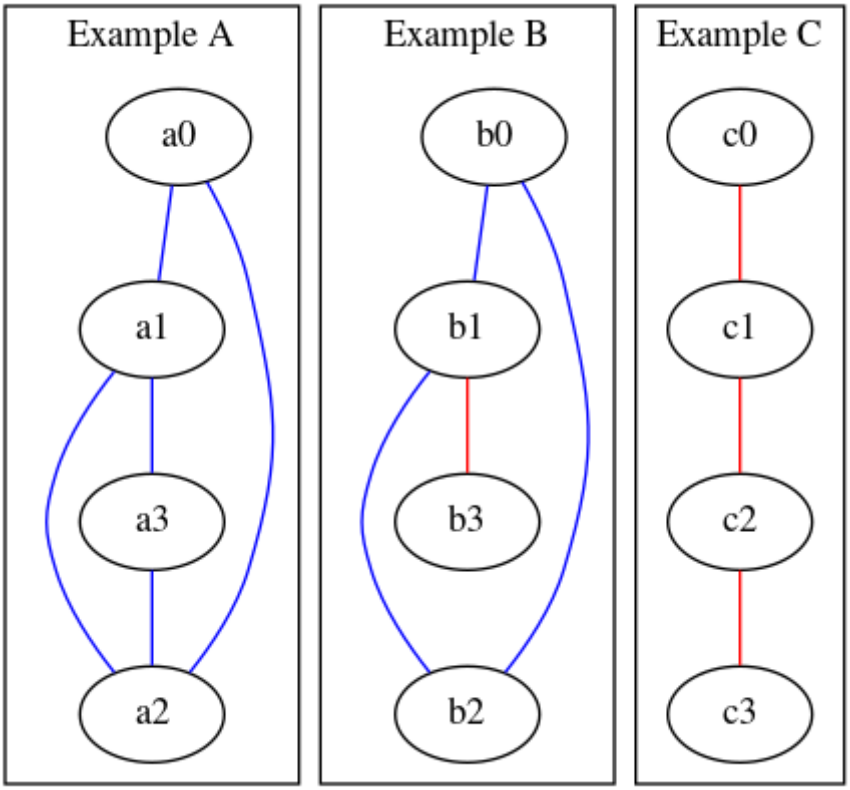
The time complexity is `O(|E|^2 * |V|)`. Can we do it on one pass? Sure we can!

Tarjan's Strongly Connected Components Algorithms

A connection is critical only if it's not part of the cycle.

In other words, a critical path is like a bridge that connects islands; if you remove it you won't cross from one island to the other.

Connections that are part of the cycle (blue) have redundancy. If you eliminate one, you can still reach other nodes. Check out the examples below.



The red connections are critical; if we remove any, some servers won't be reachable.

We can solve this problem in one pass using DFS. But for that, we keep track of the nodes that are part of a loop (strongly connected components). We use the time of visit (or depth in the recursion) each node.

For example C, if we start on `c0`, it belongs to group 0, then we move c1, c2, and c3, increasing the depth counter. Each one will be on its own group since there's no loop.

For example B, we can start at `b0`, and then we move to `b1` and `b2`. However, `b2` circles back to `b0`, which is on group 0. We can update the group of `b1` and `b2` to be 0 since they are all connected in a loop.

For an **undirected graph**, If we found a node on our DFS, that we have previously visited, we found a loop! We can mark all of them with the lowest group number. We know we have a critical path when it's a connection that links two different groups. For example A, they all will belong to group

0, since they are all in a loop. For Example B, we will have `b0`, `b1`, and `b2` on the same group while `b3` will be on a different group.

Algorithm:

- Build the graph as an adjacency list (map + array)
- Run dfs on any node. E.g. `0`.
  - Keep track of the nodes that you have seen using `group` array. But instead of marking them as seen or not. Let's mark it with the `depth`.
  - Visit all the adjacent nodes that are NOT the parent.
  - If we see a node that we have visited yet, do a DFS on it and increase the depth.
  - If the adjacent node has a lower grouping number, update the current node with it.
  - If the adjacent node has a higher grouping number, then we found a critical path.

Implementation:

```
function criticalConnections(n, connections) {
  const critical = [];
  const graph = new Map(Array(n).fill(0).map((_, i) => [i, []]));
  connections.forEach(([u, v]) => {
    graph.get(u).push(v);
    graph.get(v).push(u);
  });

  const dfs = (node, parent = null, depth = 0, group = []) => {
    group[node] = depth;
    for (const adj of (graph.get(node) || [])) {
      if (adj === parent) continue; // skip parent node
      if (group[adj] === undefined) dfs(adj, node, depth + 1, group);
      group[node] = Math.min(group[node], group[adj]); // update group.
      if (group[adj] >= depth + 1) critical.push([node, adj]);
    }
  };

  dfs(0);
  return critical;
}
```

This algorithm only works with DFS.

Complexity Analysis:

- Time: `O(|E| + |V|)`. We visit each node and edge only once.
- Space: `O(|E| + |V|)`. The graph has all the edges and nodes. Additionally, we use the `group`

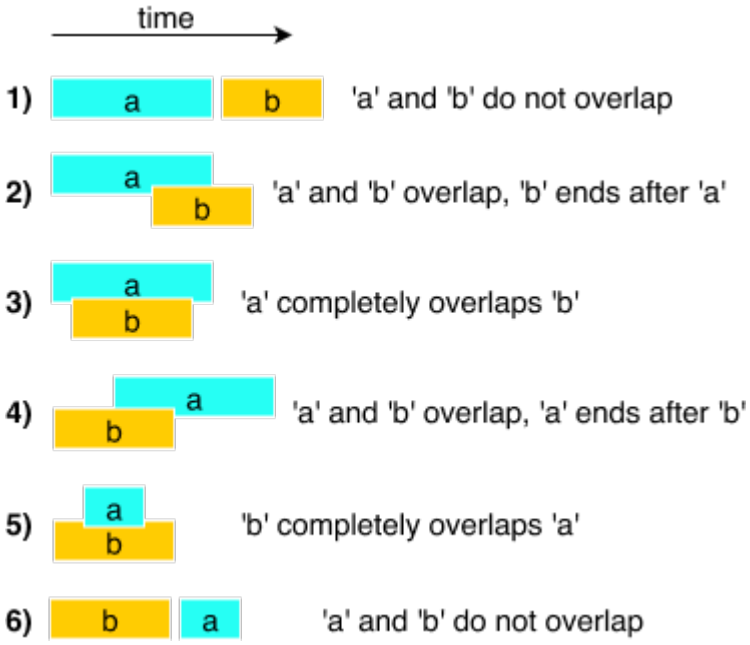
variable with a size of `|V|`.

D.9. Solutions for Sorting Questions

D.9.1. Merge Intervals

so-1) *Given an array of intervals* `[start, end]`*, merge all overlapping intervals.*

The first thing we need to understand is all the different possibilities for overlaps:



One way to solve this problem is sorting by start time. That will eliminate half of the cases! A will always start before B. Only 3 cases apply: - No overlap: E.g.,`[[1, 3], [4, 6]]`. - Overlap at the end: E.g., `[[1, 3], [2, 4]]`. - Eclipse: E.g.,`[[1, 9], [3, 7]]`.

Algorithm:

- Sort intervals by start time
- If the `curr`ent interval's start time is _equal_ or less than the `last` interval's end time, then we have an overlap.
  - Overlaps has two cases: 1) `curr`'s end is larger 2) `last`'s end is larger. For both cases, `Math.max` works.
- If there's no overlap, we add the interval to the solution.

Implementation:

```
/**
 * Merge overlapping intervals.
 * @param {[numer, number][]} intervals - Array with pairs [start, end]
 * @returns {[numer, number][]} - Array of merged pairs [start, end]
 */
function merge(intervals) {
  const ans = [];

  intervals.sort((a, b) => a[0] - b[0]); // sort by start time

  for (let i = 0; i < intervals.length; i++) {
    const last = ans[ans.length - 1];
    const curr = intervals[i];
    if (last && last[1] >= curr[0]) { // check for overlaps
      last[1] = Math.max(last[1], curr[1]);
    } else ans.push(curr);
  }
  return ans;
}
```

For the first interval, it will be added straight to the solution array. For all others, we will make a comparison.

Complexity Analysis:

- Time: `O(n log n)`. Standard libraries have a sorting time of `O(n log n)`, then we visit each interval in `O(n)`.
- Space: `O(n)`. In the worst-case is when there are no overlapping intervals. The size of the solution array would be `n`.

D.9.2. Sort Colors (The Dutch flag problem)

so-2) *Given an array with three possible values (0, 1, 2), sort them in linear time, and in-place. Hint: similar to quicksort, where the pivot is 1.*

We are asked to sort an array with 3 possible values. If we use the standard sorting method `Array.sort`, that will be `O(n log n)`. However, there's a requirement to solve it in linear time and constant space complexity.

The concept of quicksort can help here. We can choose `1` as a pivot and move everything less than 1 to the left and everything more significant than 1 to the right.

Algorithm:

- Initialize 3 pointers: `left = 0`, `right = len - 1` and `current = 0`.
- While the `current` pointer is less than `right`

- If `current` element is less than pivot 1, swap it to the left and increase the `left` and `current` pointer.
  - We can safely increase the current pointer
- If `current` element is bigger than pivot 1, swap it to the right and decrease `right` pointer.
  - Here, we don't increase the `current` pointer because the number that we swapped with could be another 2 and we might need to keep swapping while decreasing `right`.

Implementation:

```
/**
 * Sort array of 0's, 1's and 2's in linear time and in-place.
 * @param {numbers[]} nums - Array of number (0, 1, or 2).
 * @returns {void} Don't return anything, modify the input array.
 */
function sortColors(nums) {
  let left = 0;
  let right = nums.length - 1;
  let curr = 0;

  while (curr <= right) {
    if (nums[curr] < 1) {
      [nums[curr], nums[left]] = [nums[left], nums[curr]]; // swap
      left++;
      curr++;
    } else if (nums[curr] > 1) {
      [nums[curr], nums[right]] = [nums[right], nums[curr]]; // swap
      right--;
    } else {
      curr++;
    }
  }
}
```

We are using the destructive assignment to swap the elements. Here's another version a little bit more compact.

```
function sortColorsCompact(nums) {
  let i = 0; let lo = 0; let
    hi = nums.length - 1;
  const swap = (k, j) => [nums[k], nums[j]] = [nums[j], nums[k]];

  while (i <= hi) {
    if (nums[i] < 1) swap(i++, lo++);
    else if (nums[i] > 1) swap(i, hi--);
    else i++;
  }
}
```

Complexity Analysis:

- Time: `O(n)`. We only visit each number once.
- Space: `O(1)`. Operations are in-place. Only O(1) space variables were used.

Index