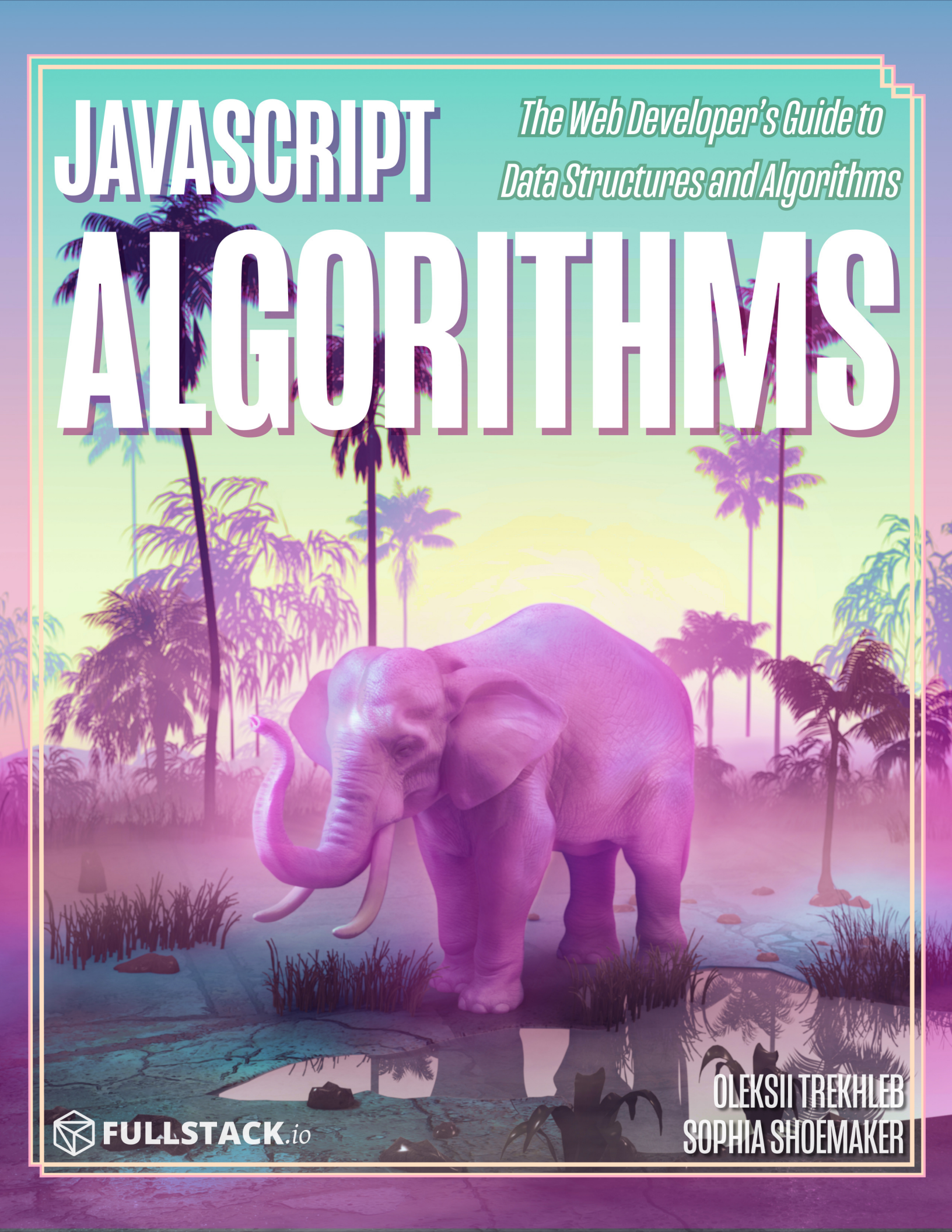
JAVASCRIPT ALGORITHMS

The Web Developer's Guide to Data Structures and Algorithms



OLEKSII TREKHLEB
SOPHIA SHOEMAKER

FULLSTACK.io

JavaScript Algorithms

The Web Developer's Guide to Data Structures and Algorithms

Written by Oleksii Trekhleb and Sophia Shoemaker
Edited by Nate Murray

Published by Fullstack.io.

Contents

Book Revision

Revision 1 - EARLY RELEASE - 2019-11-25

EARLY RELEASE VERSION

This version of the book is an early release. Most, but not all, of the chapters have been edited. Contents will change before First Edition.

Join Our Discord

Come chat with other readers of the book in the official newline Discord channel:

Join here: https://newline.co/discord/algorithms[1]

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **us@fullstack.io**.

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[2].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[3].

[1] https://newline.co/discord/algorithms
[2] https://twitter.com/fullstackio
[3] mailto:us@fullstack.io

PRE-RELEASE VERSION

This is a pre-release version of the book.

Most, but not all, of the chapters have been edited.

Join Our Discord

Come chat with other readers of the book in the official newline Discord channel:

Join here: https://newline.co/discord/algorithms[4]

Introduction

As a JavaScript developer you might think you don't need to learn about data structures or algorithms.

Did you know JavaScript itself (runtime + browser) uses things like a stack, a heap and a queue? The more you understand the JavaScript you use on a daily basis – *really understand it*, the better you can wield it's power and make less mistakes. If you thought algorithms and data structures were just a computer science course that wasn't necessary on a day to day basis as JavaScript developer, think again!

How To Read This Book

Each chapter covers an algorithm you might encounter in your work or at an interview. You can work through the algorithms one at a time or feel free to jump around.

[4] https://newline.co/discord/algorithms

Algorithms and Their Complexities

What is an Algorithm

In this book, we're going to write **algorithms** - *the sets of steps that will solve specific problems for us.*

We constantly use *algorithms* in our everyday life. For example what if we're somewhere on a street and want to get home to our apartment that is on the 20th floor? In order to achieve that we could do the following:

```
1. Take a WALK to your home.
2. Use STAIRS to go up to the 20th floor.
```

This is an algorithm. We've defined and used the set of steps that solve our task of getting home.

Algorithm Complexity

Let's take the example of "getting home" issue from the previous section. Is there another way of solving it? Yes, we implement different steps (read "algorithm") to achieve that goal:

```
1. Call a CAB that will drive you home.
2. Use the ELEVATOR to get upstairs to the 20th floor.
```

Now, let's think about which of these two algorithms we should choose:

First, we need to clarify our restrictions (how much money do we have, how urgently we need to get home, how healthy we are) and *evaluate algorithms* from the perspective of these restrictions.

As you can see each of these two algorithms requires a different amount of time and resources. The first one will require more time and energy but less money. The second one will require less time and energy but more money. And if we don't have money and are pretty healthy we need to choose the first algorithm and enjoy the walk. Otherwise, if we have enough money and need to get home urgently the second option is better.

What we've just done here is we've evaluated the *complexity of each algorithm.*

Time and Space Complexities

In the previous example, we've evaluated time and resources (i.e. money and/or health) that each algorithm requires. We will refer to these evaluations as *time complexity* and *resource complexity*. These two characteristics are crucial for us to decide which algorithm suits our needs.

In this book, we won't deal with "getting home" problems from the perspective of daily routine planning where we may apply our intuition *only*. Rather, we will deal with computational issues that are being solved on computer devices (i.e. finding the shortest path home in GPS navigator) where we must express our intuition about algorithm complexity in a much more strict, objective and mathematical way. When we speak about algorithms we normally use such metrics as *time complexity* (how much time the algorithm requires to solve a problem) and *space complexity* (how much memory the algorithm requires to solve a problem). These two metrics need to be evaluating to make a decision of which algorithm is better.

The question is how we may express the values of time and space complexities in a more formal *mathematical* way. And here is where *big O notation* comes to the rescue…

Big O Notation

Big O Notation Definition

Big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows.

The key here is that Big O notation is a *relative* metric that shows how fast execution time of the algorithm or its consumed memory will grow depending on input size growth.

Absolute values of time and memory vary on different hardware so we need to use a relative metric. The same program may be executed in 1 second on one computer and in 10 seconds on another one. Thus when you compare time complexities of two algorithms it is required for them to be running on the *same* predefined hardware configuration which is not convenient and sometimes is not reproducible.

Big O notation defines relationship `input size → spent time` for time complexity and `input size → consumed memory` for space complexity. It characterizes programs (functions) according to their growth rates: different programs with the same growth rate may be represented using the same O notation.

The letter O is used because the growth rate of a program (function) is also referred to as the *order of the function*.

Big O Notation Explanation

Let's illustrate the Big O concept on the "getting home" algorithm that was mentioned above, specifically the "stair climbing" portion of the algorithm.

Let's imagine we have two men: one is a young and fast Olympic champion and the other one is your typical middle-aged male. We've just agreed that we want the complexity of "stairs climbing" algorithm to be the same for both men so it should not depend on a man's speed but rather depends on how high the two men climb. In terms of a computer, our algorithm analysis will not depend on the hardware, but on the software problem itself. How many steps would these two men need to take to get to the 20^{th} floor? Well, let's assume that we have `20` steps between each floor. That means the two men need to take `20 * 20 = 400` steps to get to the 20^{th} floor. Let's generalize it for `n's` floor:

```
Input: (20 * n) steps
Output: (20 * n) steps (moves)
```

Do you see the pattern here? The time complexity of climbing the stairs has a linear dependency on the number of floors. In Big O syntax, this is written as follows:

```
O(20 * n)
```

One important rule to understand here:

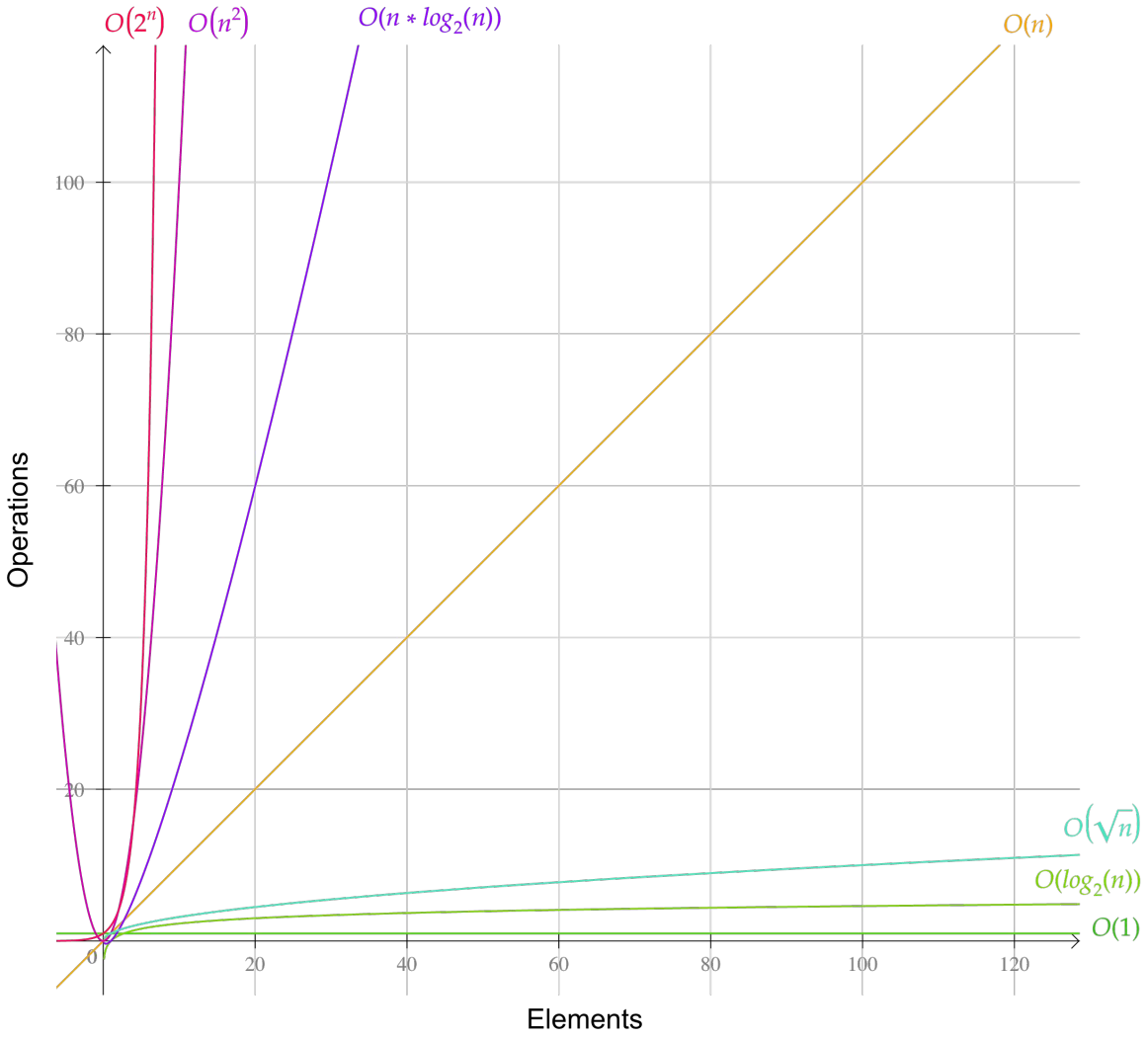
> We must always get rid of non-dominant parts when describing complexity using Big O notation: `O(n + 1)` becomes `O(n)`, `O(n + n^2)` becomes `O(n^2)`, `O(n + log(n))` becomes `O(n)` and so on.

Why is that? Because Big O must describe the order of the function and not an exact number of milliseconds or megabytes it consumes. It describes the trend and not absolute values. Therefore non-dominant constants do not matter. Non-dominant constants are the ones we may discard for huge values of `n`. For example in a situation when `n = 10000` which of these two parts will influence the final number more: `20` or `n = 10000`? It is obvious that the influence of `n` to the final number is `500` times bigger than the constant `20`. Therefore, we may discard it and the time complexity of the "stairs climbing" algorithm is as follows:

```
O(n)
```

This was an example of linear dependency, but there are other types of dependency exists as well: `O(log(n))`, `O(n * log(n))`, `O(n^2)`, `O(2^n)`, `O(n!)` etc.

Take a look at the graphs of functions commonly used in the analysis of algorithms, showing the number of operations `N` versus input size `n` for each function.



All these graphs show us how fast our function's memory and time consumption will grow depending on the input. Two different array sorting algorithms may accomplish the same task of sorting but one will do it in `O(n * log(n))` and another one in `O(n^2)` time. If we have a choice, we would prefer the first one over the second one.

With Big O notation we may compare algorithms (functions, programs) based on their Big O value independently of the hardware they are running on.

Take a look at the list of some of the most common Big O notations and their performance comparisons against different sizes of the input data. This will give you the feeling of how different algorithms complexities (time and memory consumptions) may be.

Big O Notation	Computations for 10 elements	Computations for 100 elements	Computations for 1000 elements
O(1)	1	1	1
O(log(n))	3	6	9
O(n)	10	100	1000
O(n log(n))	30	600	9000
O(n^2)	100	10000	1000000
O(2^n)	1024	1.26e+29	1.07e+301
O(n!)	3628800	9.3e+157	4.02e+2567

Big O Notation Examples

Let's move on and see some other examples of big O notation.

O(1) example

Let's write a function that raises a number to a specified power:

01-algorithms-and-their-complexities/fastPower.js

```
/**
 * Raise number to the power.
 *
 * Example:
 * number = 3
 * power = 2
 * output = 3^2 = 9
 *
 * @param {number} number
 * @param {number} power
 * @return {number}
 */
export function fastPower(number, power) {
  return number ** power;
}
```

What would be its time and space complexities?

Let's think about how the input of the function will affect the number of operations it will accomplish. For every input, the function does exactly one operation. That would mean that time complexity of this function is:

```
Time complexity: O(1)
```

Now let's roughly analyze how much memory it will require to evaluate this function. In this case, the number of variables we operate with doesn't depend on the input. So the space complexity is:

```
Space complexity: O(1)
```

O(n) example

And now let's implement the same power function but in an iterative way:

01-algorithms-and-their-complexities/iterativePower.js

```
/**
 * Raise number to the power.
 *
 * Example:
 * number = 3
 * power = 2
 * output = 3^2 = 9
 *
 * @param {number} number
 * @param {number} power
 * @return {number}
 */
export function iterativePower(number, power) {
  let result = 1;

  for (let i = 0; i < power; i += 1) {
    result *= number;
  }

  return result;
}
```

The function still does the same. But let's see how many operations it will perform depending on the input.

All operations outside and inside the `for()` loop require constant time and space. There are just assignment, multiplication and return operations. But `for()` loop itself acts as a multiplier for time complexity of the code inside of it because it will make the body of the loop to be executed `power` times. So the time complexity of this function may be expressed as a sum of time complexities of its code parts like `O(1) + power * O(1) + O(1)`. Where the first and last `O(1)` in the equation are for assignment and return operations. After we drop non-dominant parts like `O(1)` and move `power` into `O()` brackets we will get our time complexity as:

```
Time complexity: O(power)
```

Notice that we've written `O(power)` and not `O(number)`. This is important since we're showing that the execution time of our function directly depends on the `power` input. A big `number` input won't slow the function down but big `power` input will.

From the perspective of space complexity, we may conclude that our function won't create additional variables or stack layers with bigger input. Amount of memory it consumes does not change.

```
Space complexity: O(1)
```

O(n) recursive example

In mathematics, the factorial of a non-negative integer `n`, denoted by `n!`, is the product of all positive integers less than or equal to `n`. For example:

```
5! = 1 * 2 * 3 * 4 * 5 = 120
```

Let's write recursive factorial function:

01-algorithms-and-their-complexities/factorial.js

```
/**
 * Calculate factorial.
 *
 * Example:
 * number = 5
 * output = 120
 *
 * @param {number} number
 * @return {number}
 */
export function factorial(number) {
  if (number === 0) {
    return 1;
  }

  return factorial(number - 1) * number;
}
```

If we were not dealing with recursion we would say that all operations inside the function have `O(1)` time complexity and thus overall time complexity is also `O(1)` (the sum of all `O(1)`s with discarded non-dominant parts). But we *do* have recursion call here at the very last line of the function. Let's analyze it and see the tree of recursion calls for `4!`:

```
factorial(4)
    □ factorial(3) * 4
        □ factorial(2) * 3
            □ factorial(1) * 2
```

You see that from time perspective our recursion acts like a `for()` loop from the previous example multiplying time complexity of function body by `number`. Thus time complexity is calculated as follows:

```
Time complexity: O(number)
```

You may also notice that recursion makes the stack of function calls grow proportionally to the `number`. This happens because to calculate `factorial(4)` computer needs to calculate `factorial(3)` first and so on. Thus all intermediate calculations and variables must be stored in memory before the next one in the stack will be calculated. This will lead us to the conclusion that space complexity is in linear proportion to the `number`. The space complexity of one function call must be multiplied by the `number` of function calls. Since one call of `factorial()` function would cost us constant memory (because the number of variables and their sizes is constant) then we may calculate overall recursive space complexity as follows:

```
Space complexity: O(number)
```

O(n²) example

Let's write a function that generates all possible pairs out of provided letters:

01-algorithms-and-their-complexities/pairs.js

```
1  /**
2   * Get all possible pairs out of provided letters.
3   *
4   * Example:
5   * letter = ['a', 'b']
6   * output = ['aa', 'ab', 'ba', 'bb']
7   *
8   * @param {string[]} letters
9   * @return {string[]}
10  */
11 export function pairs(letters) {
12   const result = [];
13
14   for (let i = 0; i < letters.length; i += 1) {
15     for (let j = 0; j < letters.length; j += 1) {
16       result.push(`${letters[i]}${letters[j]}`);
```

```
    }
  }

  return result;
}
```

Now we have two nested `for()` loops that acts like multipliers for the code inside the loops. Inside the loops we have a code with constant execution time `O(1)` - it is just a pushing to the array. So let's calculate our time complexity as a sum of our function parts time complexities: `O(1) + letters.length * letters.length * O(1) + O(1)`. After dropping non-dominant parts we'll get our final time complexity:

```
Time complexity: O(letters.length^2)
```

This would mean that our function will slow down proportionally to the square of letters number we will provide for it as an input.

You may also notice that the more letters we will provide for the function as an input the more pairs it will generate. All those pairs are stored in `pairs` array. Thus this array will consume the memory that is proportional to the square of letters number.

```
Space complexity: O(letters.length^2)
```

Space complexity and auxiliary space complexity example

Additionally, to *space complexity*, there is also *auxiliary space complexity*. These two terms are often misused.

The difference between them is that auxiliary space complexity *does not include the amount of memory we need to store input data.* Auxiliary space complexity only shows how much *additional* memory the algorithm needs to solve the problem. Space complexity on contrary takes into account both: the amount of memory we need to store the input data and also the amount of additional memory the algorithm requires.

> Space complexity = Input + Auxiliary space complexity

Let's see it from the following example. Here is a function that multiplies all array elements by specific value:

01-algorithms-and-their-complexities/multiplyArrayInPlace.js

```
/**
 * Multiply all values of the array by certain value in-place.
 *
 * Example:
 * array = [1, 2, 3]
 * multiplier = 2
 * output = [2, 4, 6]
 *
 * @param {number[]} array
 * @param {number} multiplier
 * @return {number[]}
 */
export function multiplyArrayInPlace(array, multiplier) {
  for (let i = 0; i < array.length; i += 1) {
    array[i] *= multiplier;
  }

  return array;
}
```

The space complexity of this function is in linear proportion to input array size. So the bigger the input array size is the more memory this function will consume.

```
Space complexity: O(array.length)
```

Not let's analyze additional space. We may see that function does all of its operations in-place without allocating any *additional* memory.

```
Auxiliary space complexity: O(1)
```

Sometimes it may be unacceptable to modify function arguments. And instead of modifying the input array parameter in-place we might want to clone it first. Let's see how we change the array multiplication function to prevent input parameters modification.

01-algorithms-and-their-complexities/multiplyArray.js

```
/**
 * Multiply all values of the array by certain value with allocation
 * of additional memory to prevent input array modification.
 *
 * Example:
 * array = [1, 2, 3]
 * multiplier = 2
 * output = [2, 4, 6]
 *
 * @param {number[]} array
 * @param {number} multiplier
 * @return {number[]}
 */
export function multiplyArray(array, multiplier) {
  const multipliedArray = [...array];

  for (let i = 0; i < multipliedArray.length; i += 1) {
    multipliedArray[i] *= multiplier;
  }

  return multipliedArray;
}
```

In this version of the function, we're allocating additional memory for cloning the input array. Thus our space complexity becomes equal to `O(2 * array.length)`. After discarding non-dominant parts we get the following:

```
Space complexity: O(array.length)
```

Because our algorithm now allocates additional memory the auxiliary space complexity has also changed:

```
Auxiliary space complexity: O(array.length)
```

Understanding the difference is important so as not to get confused by space complexity evaluations. Auxiliary space complexity might also become handy for example if we want to compare standard sorting algorithms on the basis of space. It is better to use auxiliary space than space complexity because it will make the difference between algorithms clearer. Merge Sort uses `O(n)` auxiliary space, Insertion Sort and Heap Sort use `O(1)` auxiliary space. But at the same time, the space complexity of all these sorting algorithms is `O(n)`.

Other examples

You will find other big O examples like `O(log(n))`, `O(n * log(n))` in the next chapters of this book.

Quiz

Q1: Does time complexity answer the question of how many milliseconds an algorithm would take to finish?

Q2: Which time complexity means faster execution time: `O(n)` *or* `O(log(n))`*?*

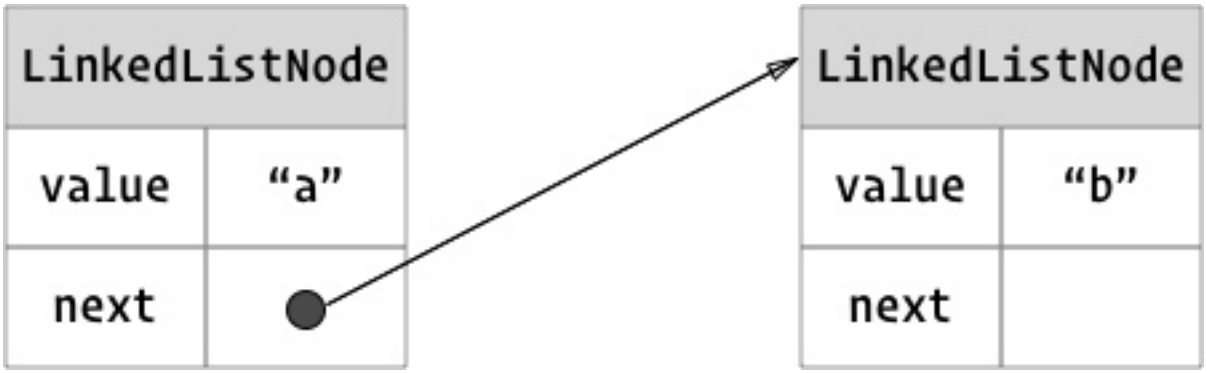
Q3: Is it true that algorithms with better time complexity also have better space complexity?

Linked List

- *Difficulty: easy*

Linked list and its common operations

A linked list is a collection of entities which are not stored in sequential order. Instead, each entity has a pointer to the next entity. Each entity, also referred to as a node, is composed of data and a reference (in other words, a link) to the next node in the sequence.



This structure allows for efficient insertion or removal of nodes from any position in the sequence during iteration. More complex implementations of linked lists add additional links, allowing efficient insertion or removal from arbitrary node references. A drawback of linked lists is that access time is linear. Faster access, such as random access, is not possible.

The main operations on linked lists are:

- `prepend` - add a node to the beginning of the list
- `append` - add a node to the end of the list
- `delete` - remove a node from the list
- `deleteTail` - remove the last node from the list
- `deleteHead` - remove the first node from the list
- `find` - find a node in the list

Applications

A linked list is used as a good foundation for many other data structures like queues, stacks and hash tables. The linked list implementation we will create in JavaScript will be used for many of the other data structures we will learn about in this book.

Implementation

Our LinkedList implementation will consist of two parts:

- A LinkedListNode class
- A LinkedList class

We will explore the details of each of these classes in the following sections.

`LinkedListNode`

Our `LinkedListNode` class is a class that will represent one item in our collection of items. The `LinkedListNode` has two important properties:

1. `this.value`
2. `this.next`

`this.value` contains the actual value we want to store in our node (a number, another object, a string, etc). `this.next` is a pointer to the next node our list of nodes.

What is a pointer?

Objects in JavaScript are dictionaries with key-value pairs. Keys are always strings and values can be a boolean or strings, numbers and objects. When we say our `next` property in our `LinkedListNode` class is a "pointer", what we really mean is that the value of `this.next` is a reference to another JavaScript object – another `LinkedListNode` object.

`constructor`

Our constructor method for our `LinkedListNode` class sets the initial values for `this.value` and `this.next` – both set to `null`.

02-linked-list/LinkedListNode.js

```
2   constructor(value, next = null) {
3     this.value = value;
4     this.next = next;
5   }
```

`toString`

Our `toString` method is a convenience method when we want to see what is contained in each node of our linked list.

02-linked-list/LinkedListNode.js

```
  toString(callback) {
    return callback ? callback(this.value) : `${this.value}`;
  }
```

LinkedList

The `LinkedList` class contains some important methods to be able to manipulate our collection of `LinkedListNode` objects. Let's take a look at what each method does:

constructor

Our `LinkedList` class contains two properties: `this.head` and `this.tail`. Each of these properties points to a `LinkedListNode` object. They are the beginning and the end of our collection of `LinkedListNodes`.

02-linked-list/LinkedList.js

```
  constructor() {
    /** @var LinkedListNode */
    this.head = null;

    /** @var LinkedListNode */
    this.tail = null;
  }
```

prepend

The prepend method adds a `LinkedListNode` object to the beginning of our LinkedList. First, we create a new `LinkedListNode` object.

02-linked-list/LinkedList.js

```
  prepend(value) {
    // Make new node to be a head.
    const newNode = new LinkedListNode(value, this.head);
```

Then we set our `this.head` to point to our newly created `LinkedListNode` object. If this is the first item in our LinkedList, we must also set `this.tail` to point to the new object, as it is also the last item in our LinkedList.

02-linked-list/LinkedList.js

```
19      this.head = newNode;
20
21      // If there is no tail yet let's make new node a tail.
22      if (!this.tail) {
23        this.tail = newNode;
24      }
```

Finally, we return the value that was added to our LinkedList

02-linked-list/LinkedList.js

```
25      return this;
26    }
```

append

The append method adds a `LinkedListNode` object to the end of our LinkedList. First, we create a new `LinkedListNode` object.

02-linked-list/LinkedList.js

```
33    append(value) {
34      const newNode = new LinkedListNode(value);
```

If there are no other items in our LinkedList, then we need to set `this.head` to our new node.

02-linked-list/LinkedList.js

```
35      // If there is no head yet let's make new node a head.
36      if (!this.head) {
37        this.head = newNode;
38        this.tail = newNode;
39
40        return this;
41      }
```

Then, we get the value of `this.tail` and set that objects `next` property to be our newly created `LinkedListNode` object.

02-linked-list/LinkedList.js

```
44      const currentTail = this.tail;
45      currentTail.next = newNode;
46      // Attach new node to the end of linked list.
```

Finally, we set `this.tail` to be the new `LinkedListNode` object.

02-linked-list/LinkedList.js

```
47      this.tail = newNode;
48
49      return this;
```

delete

Our `delete` method finds an item in the LinkedList and removes it. First, we check to see if our LinkedList is empty by checking if `this.head` is null. If it is, we return `null`.

02-linked-list/LinkedList.js

```
56    delete(value) {
57      if (!this.head) {
58        return null;
59      }
```

If our LinkedList contains items in it, then we create a `deletedNode` variable which will serve as a placeholder for the node that we will eventually return as the result from our delete method.

02-linked-list/LinkedList.js

```
61      let deletedNode = null;
```

Before we traverse our LinkedList, we first check to see if the value of `this.head` matches the value we are trying to delete. If so, we set our `deletedNode` variable to `this.head` and then move `this.head` to be the next item in our LinkedList.

02-linked-list/LinkedList.js

```
62      // If the head must be deleted then make next node that is different
63      // from the head to be a new head.
64      while (this.head && this.head.value === value) {
65        deletedNode = this.head;
66        this.head = this.head.next;
67      }
```

In order to traverse our LinkedList, we need to create a placeholder variable, `currentNode` to keep track of where we are in our LinkedList. We set the value of `currentNode` equal to `this.head` to ensure we start at the beginning of our list.

02-linked-list/LinkedList.js

```
70      let currentNode = this.head;
```

We then check to make sure that our `currentNode` is not null. If it isn't, we kick off our while loop. We will break out of our while loop when we reach a node where its `this.next` property is null, meaning it doesn't point to another `LinkedListNode` object.

02-linked-list/LinkedList.js

```
72      if (currentNode !== null) {
73        while (currentNode.next) {
74          if (currentNode.next.value === value) {
```

Next, we check to see if the value of `currentNode.next` object is equal to the value we want to delete. If it is, then we set our `deletedNode` variable equal to `currentNode.next`. We also set the `currentNode.next` pointer equal to the object that the `currentNode.next` object points to.

If the values don't match, we just move on to the next item in our LinkedList.

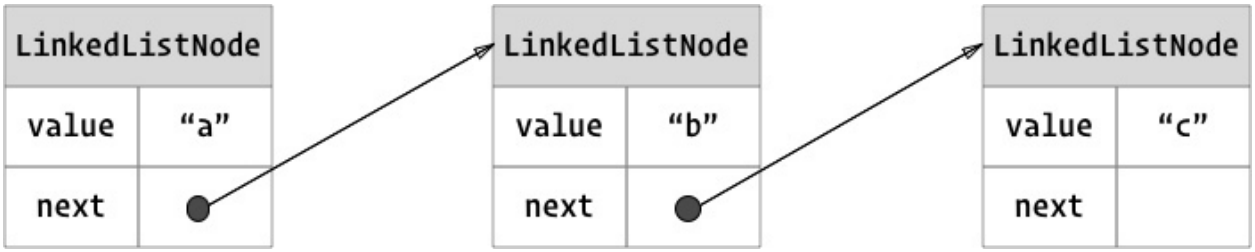
02-linked-list/LinkedList.js

```
72      if (currentNode !== null) {
73        while (currentNode.next) {
74          if (currentNode.next.value === value) {
```

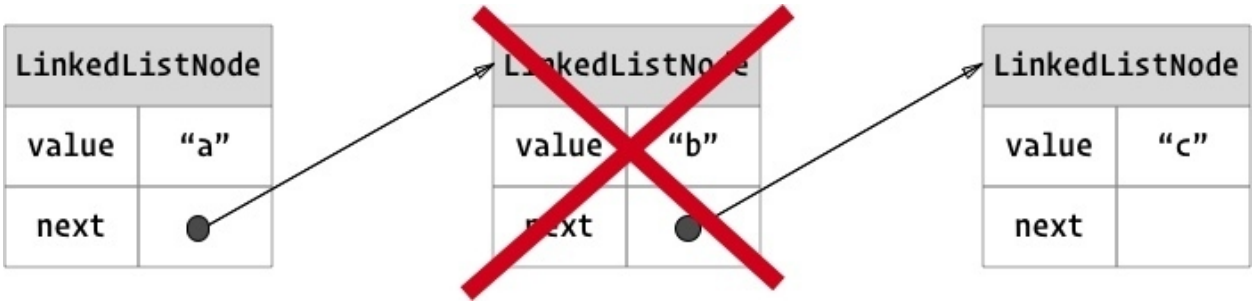
Finally, we must check if the value of the tail of our LinkedList matches the value we are trying to remove. If it does match, we'll set `this.tail` equal to `currentNode`.

See the following diagrams for a visual representation of what happens when the delete method is called:

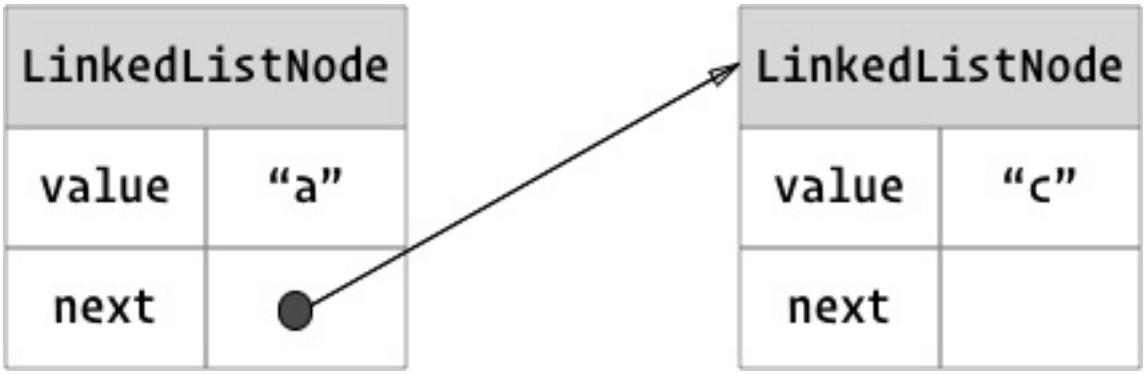
Linked list before `delete` method is called



`delete` method is called



Linked list after `delete` method is called



`deleteTail`

For our `deleteTail` method, we first create a variable `deletedTail` and set it to `this.tail` so we can return the object that is deleted when we return from this method.

02-linked-list/LinkedList.js

```
95   deleteTail() {
96     const deletedTail = this.tail;
```

We have two scenarios for which we need to write code:

1. We only have one item in our LinkedList
2. We have many items in our LinkedList

For scenario 1, `this.head.value` and `this.tail.value` are the same, so we set both `this.head` and `this.tail` to null and return `deleteTail`.

02-linked-list/LinkedList.js

```
98      if (this.head === this.tail) {
99        // There is only one node in linked list.
100       this.head = null;
101       this.tail = null;
102
103       return deletedTail;
104     }
```

For scenario 2, we need to traverse the entire list to get to the next to last `LinkedListNode` in the LinkedList. We will set the next to last `LinkedListNode` to `this.tail` by checking if the `currentNode.next.next` value is null. If `currentNode.next.next` is null, we know the `currentNode.next` is the object we want to set to `this.tail`.

02-linked-list/LinkedList.js

```
106     // If there are many nodes in linked list...
107     // Rewind to the last node and delete "next" link for the node before the last o\
108 ne.
109     let currentNode = this.head;
110     while (currentNode.next) {
111       if (!currentNode.next.next) {
112         currentNode.next = null;
113       } else {
114         currentNode = currentNode.next;
115       }
116     }
117
118     this.tail = currentNode;
119
120     return deletedTail;
121   }
```

And lastly, we return our `deletedTail` object from the method:

02-linked-list/LinkedList.js

```
121
```

deleteHead

Our `deleteHead` method removes the first item in the list. Similar to our `deleteTail` method, we have a few scenarios for which we need to write code to delete the head our LinkedList:

1. There are no items in our list
2. There is only 1 item in our list
3. There are many items in our list

For scenario 1, we check if `this.head` is null, if it is null we return null from the method

02-linked-list/LinkedList.js

```
125   deleteHead() {
126     if (!this.head) {
127       return null;
128     }
```

For scenarios 2 and 3, we must first create a variable: `deletedHead` and set it to `this.head` so we can return the object that is deleted when we return from this method.

02-linked-list/LinkedList.js

```
130     const deletedHead = this.head;
```

Also for scenarios 2 and 3, we combine the logic into an if/else statement. If `this.head.next` is not null, we know there is more than 1 item in the list and we set `this.head.next` to equal `this.head` which removes the first item in the list. If `this.head.next` is null, then we know there is only 1 item in the list and we set both `this.head` and `this.tail` to null as there are no more items in our LinkedList.

02-linked-list/LinkedList.js

```
131     if (this.head.next) {
132       this.head = this.head.next;
133     } else {
134       this.head = null;
135       this.tail = null;
136     }
```

Finally, we return our `deletedHead` object from the method:

02-linked-list/LinkedList.js

```
139     return deletedHead;
140   }
```

find

The `find` method traverses the items in the LinkedList and locates the first `LinkedListNode` object that matches the value we want. The method takes an object that has two key/value pairs, a `value` and a `callback`:

02-linked-list/LinkedList.js

```
148   find({ value = undefined, callback = undefined }) {
```

As in our previous methods, we first check if `this.head` is null. If it is null, we return from the method because the LinkedList is empty.

02-linked-list/LinkedList.js

```
149     if (!this.head) {
150       return null;
151     }
```

Next, we create a variable `currentNode` to keep track of where we are in the LinkedList and set it to `this.head` to start at the beginning of the LinkedList.

02-linked-list/LinkedList.js

```
152
```

Then we create a while loop to loop over all the nodes in the LinkedList and either call the callback function sent as a parameter or check the value of the current node to see if it matches the value we are looking for. If there is a match, we return the node.

02-linked-list/LinkedList.js

```
155     while (currentNode) {
156       // If callback is specified then try to find node by callback.
157       if (callback && callback(currentNode.value)) {
158         return currentNode;
159       }
160
161       // If value is specified then try to compare by value..
162       if (value !== undefined && currentNode.value === value) {
163         return currentNode;
164       }
165
166       currentNode = currentNode.next;
167     }
```

Finally, if we go through the entire LinkedList and do not find the value we are looking for, we return null from our method.

02-linked-list/LinkedList.js

```
169     return null;
```

At this point, we've implemented the essential LinkedList methods. We'll now implement two helper methods that will make it more convenient to test and debug our LinkedList.

toArray

The `toArray` method takes all the nodes in the LinkedList and puts them in an array. To put them in an array, we create a new array called `nodes`. Then we loop over all the nodes in the LinkedList and push each one on to the array.

02-linked-list/LinkedList.js

```
175   toArray() {
176     const nodes = [];
177
178     let currentNode = this.head;
179     while (currentNode) {
180       nodes.push(currentNode);
181       currentNode = currentNode.next;
182     }
183
184     return nodes;
185   }
```

`toString`

The `toString` method takes the LinkedList and prints out a string representation of the LinkedList. The method takes the LinkedList and first converts it to an array (using the `toArray` method described above). Then the `map` array method is called to process each LinkedListNode in the list. The `toString` method for each LinkedListNode is called and then the `toString` array method is called to print out the entire array of LinkedListNodes.

02-linked-list/LinkedList.js

```
191   toString(callback) {
192     return this.toArray().map(node => node.toString(callback)).toString();
193   }
```

Complexities

Access	Search	Insertion	Deletion
O(n)	O(n)	O(1)	O(1)

Problems Examples

Here are some related problems that you might encounter during the interview:

- Cycle in a Linked List[5]
- Intersection of Two Linked LIsts[6]

Quiz

Q1: What does `this.next` refer to in a linked list in JavaScript?

Q2: What is the difference between a deleteTail and deleteHead method in a linked list?

References

Linked List on Wikipedia https://en.wikipedia.org/wiki/Linked_list[7]

Linked List on YouTube https://www.youtube.com/watch?v=njTh_OwMljA[8]

[5] https://leetcode.com/problems/linked-list-cycle-ii/
[6] https://leetcode.com/problems/intersection-of-two-linked-lists/
[7] https://en.wikipedia.org/wiki/Linked_list
[8] https://www.youtube.com/watch?v=njTh_OwMljA

Linked List example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/linked-list[9]

Doubly Linked List implementation https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/doubly-linked-list[10]

[9] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/linked-list
[10] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/doubly-linked-list

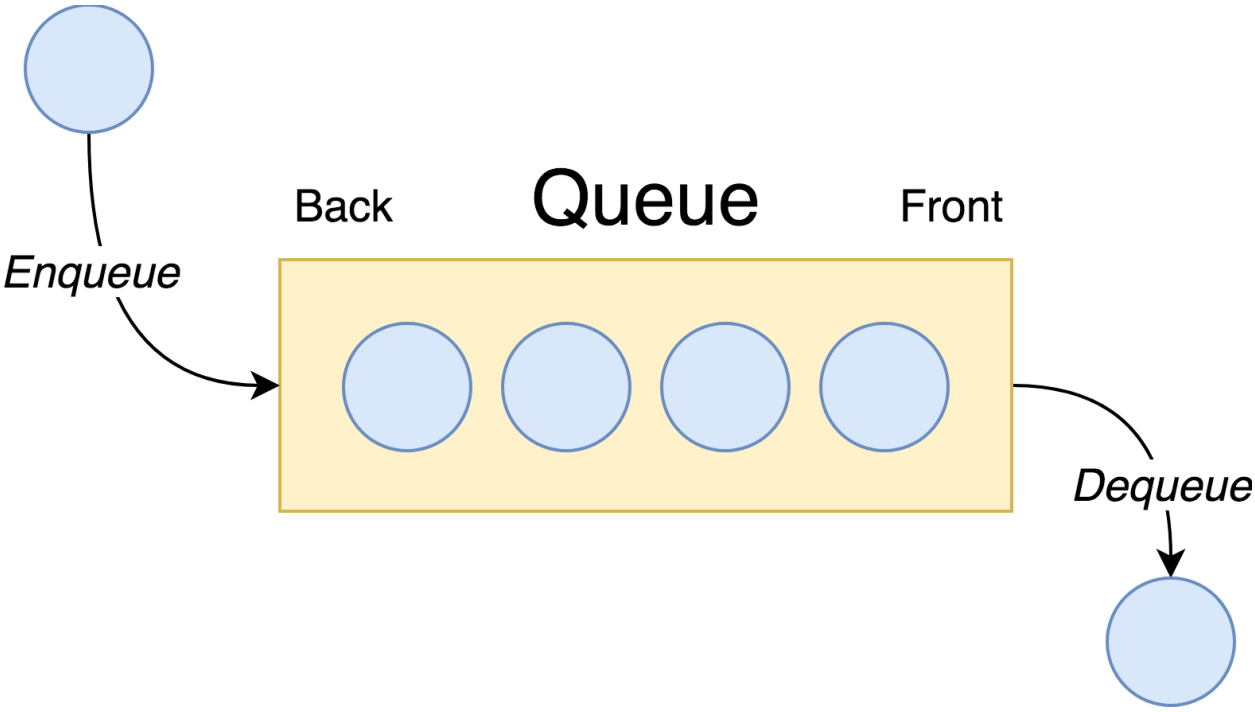
Queue

- *Difficulty: easy*

Queue and its common operations

A **queue** is a collection of entities in which the entities are kept in *First-In-First-Out (FIFO)* order. In a FIFO data structure, the first element added to the queue is the first one removed. When an element is added to the queue, all the elements that were added before it must be removed before the new element is removed. A commonly seen implementation of this in the real world are lines in a supermarket where the first shoppers in line are served first, with everyone else behind the first shopper served after.

The main operations on queues are: - *enqueue* - add an entity to the rear position (joining the line at the supermarket), - *dequeue* - remove an entity from the front position (serving the first shopper in line), - *peek* - reading the entity from the front of the queue without dequeuing it (asking who the next shopper in line will be).



Operations such as *searching* or *updating* elements are not common for queues. There are only two things we're interested in when working with queues: the beginning and the end of it.

GET THE FULL BOOK

This is the end of the preview!

Get the full version at:

https://newline.co/javascript-algorithms

Featuring:

- 27 lessons
- 280+ pages
- Complete code

GET IT NOW