Data Structures - Arrays

Data Structures

A data structure is a specific solution for organizing data that provides storage for items and capabilities for storing and retrieving them.

The data structure itself can hold store data statically or dynamically.

In a **static data structure** the size of the structure is **fixed**. The content of the data structure can be modified but without changing the memory space allocated to it such as an array.

In a **dynamic data structure** the size of the structure is **not fixed** and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time such as a linked list.

Static Data structure can provide easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

Abstract Data Types

An **abstract data type** (ADT) specifies the operations that can be performed on some data and the computational complexity of those operations. No details are provided on how data is stored or how physical memory is used.

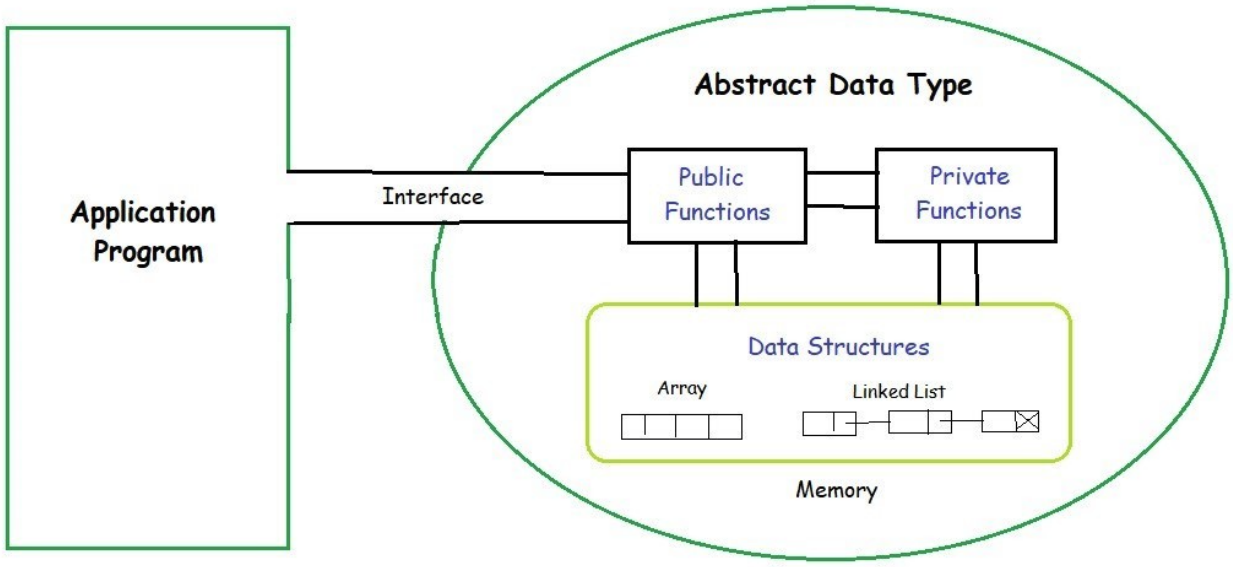
A data structure (DS) is a concrete implementation of the specification provided by an ADT.

You can think about an ADT as the blueprint, while a data structure is the translation of those specifications into real code.

An ADT is defined from the point of view of the one who uses it, by describing its behavior in terms of possible values, possible operations on it, and the output and side effects of these operations.

Common ADTs:

- Stack and Queues
- List
- Tree
- Graph



Linear vs Non-Linear Data Structures

Data structures that arrange their data linearly are where the elements are attached to its previous and next adjacent in a single level. Meaning, we can traverse all the elements in a single run only.

Examples: Array, Stack and Queue, Linked List

The opposite would be a non-linear data structure that do not have a single level and cannot traverse all the elements in a single run. Non-linear data structures are harder to implement but utilizes computer memory more efficiently. Examples of non-linear data structures would be trees and graphs.

Examples: Tree (Binary Tree, Binary Search Tree, etc.) and Graph

S.NO	Linear Data Structure	Non-linear Data Structure
1.	In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
5.	In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.

Describing a data structure

The crucial part of an ADT definition is to list the set operations that it allows. This is equivalent to defining an API.

Every time you need to describe a data structure, you can follow a few simple steps to provide a comprehensive and unambiguous specification:

1. Specifying its API first, with a focus on the methods' input and output
2. Describing its high-level behavior
3. Describing in detail the behavior of its concrete implementation
4. Analyzing the performance of its methods

Algorithms vs Data Structures

We usually use these two terms interchangeably and, for the sake of brevity, use the term data structure to mean a DS and all its relevant methods.

A good metaphor: data structures are like **nouns**, while algorithms are like **verbs**.

Data structures and algorithms are interconnected; they need each other:

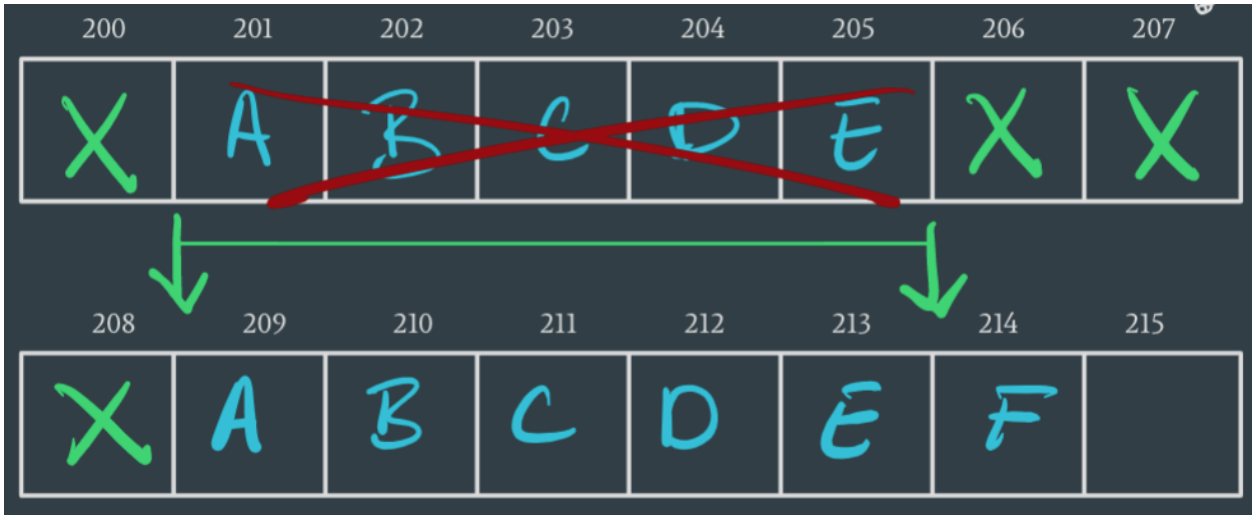
Without algorithms to transform them, data structures would just be bits stored on a memory chip; without data structures to operate on, most algorithms wouldn't even exist.

Every data structure, moreover, implicitly defines algorithms that can be performed on it—for instance, methods to add, retrieve, and remove elements to/from the data structure.

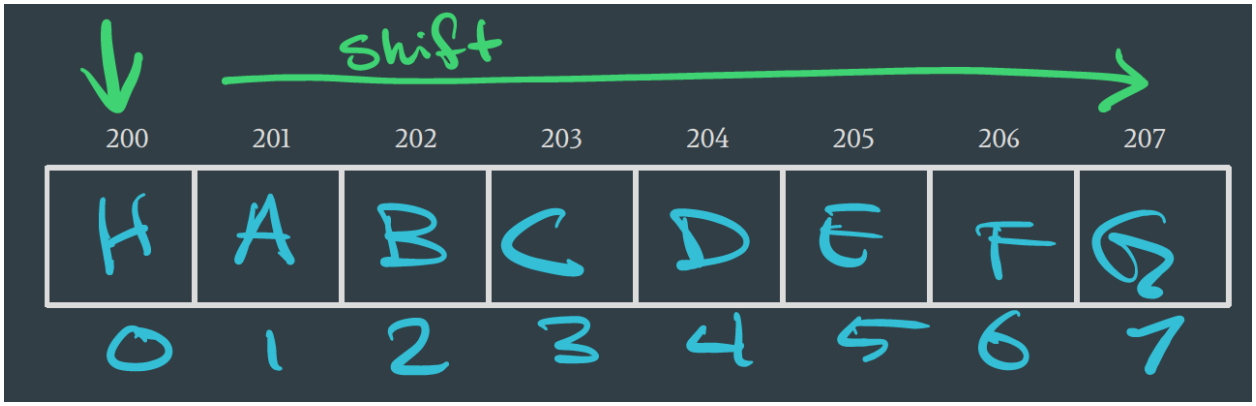
Data Structure: Array

Arrays are a series of elements that store data, differentiated by their index. Accessing the first element takes just as much time as accessing the last. Elements **must be stored contiguously** in memory.

But what happens when we run out of contiguous memory space?

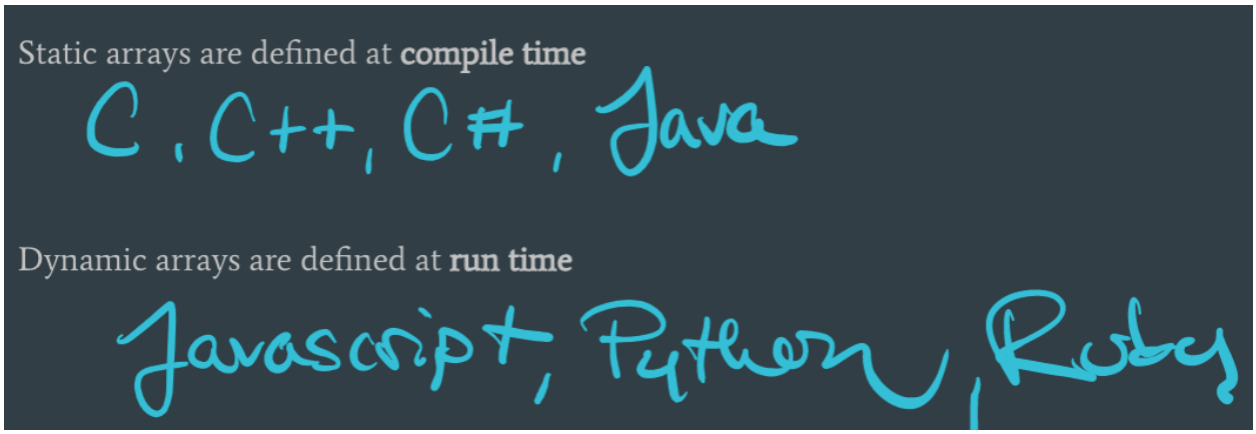


Adding or removing an element from an array requires shifting every following element forwards or backwards.



Static vs Dynamic Arrays

JavaScript directly allows arrays as dynamic. The length of an array in JavaScript can be mutated via mutator methods such as pop(), push(), shift(), unshift(), splice(), etc.



Arrays are most efficient when accessing the elements in random order or modifying only at the end of the array. Any modification to the middle of the array results in O(n) running time.

Common methods - time complexity (mutate and iterator):

- push: O(1)
- pop: O(1)
- shift: O(n)
- unshift: O(n)
- concat: O(n)
- slice: O(n)
- sort: O(n * log n)
- forEach/map/filter/etc - O(n)

Popping values

To pop a value from the end of an array is similarly simple. Rather than resize the array, you just leave an extra space which will be filled at the next push

Inserting values

What about if you want to insert a value into any point in an array, not just the middle? To do this, you need to shift all of the values after the new value back 1 position

In the best case, you are inserting the value at the back of the array: it's just the same as pushing. So insertion has a best-case performance of O(1), the same as pushing. The worst case is inserting the value at the start of the array. This requires every value to be shifted 1 memory address later; that's n copies, so it's **O(n)**.

Removing values

Removing values is very similar to inserting values, except that you are copying the values backward to fill the space where you removed the value rather than forwards to make space for a new value

Using the same logic as for insertion, the best-case performance is O(1) (the same as popping), and the average and worst cases are **O(n)**.

An array organizes items sequentially, one after another in memory.

Each position in the array has an **index**, starting at 0.

Strengths:

- **Fast lookups**. Retrieving the element at a given index takes $O(1)$ time, regardless of the length of the array.
- **Fast appends**. Adding a new element at the end of the array takes $O(1)$ time, if the array has space.

	Worst Case
space	$O(n)$
lookup	$O(1)$
append	$O(1)$
insert	$O(n)$
delete	$O(n)$

Weaknesses:

- **Fixed size**. You need to specify how many elements you're going to store in your array ahead of time. (Unless you're using a fancy dynamic array.)
- **Costly inserts and deletes.** You have to "scoot over" the other elements to fill in or close gaps, which takes worst-case $O(n)$ time.

Description	Notation	Explanation
push()	O(1)	Inserting at the end of the array
pop()	O(1)	Removing an element from the end of the array
unshift()	O(n)	Inserting at the beginning of the array
splice()	O(n)	Inserting or deleting at some arbitrary position of the array
map(), filter(), reduce()	O(n)	Requires iterating over all of the array

Data Structure: Objects

Objects are unordered data structures stored in key-value pairs.

```
let instructor = {
    firstName: "Kelly",
    isInstructor: true,
    favoriteNumbers: [1,2,3,4]
}
```

When to use objects:

- When you don't need to order
- When you need fast access/insertion/removal
    - Insertion: O(1)
    - Removal: O(1)
    - Searching: O(n)
    - Access: O(1)
- Methods:
    - Object.keys: O(n)
    - Object.values: O(n)
    - Object.entries: O(n)
    - hasOwnProperty: O(1)

Algorithm Patterns

An algorithmic pattern, or design pattern, is a method, strategy, or technique of solving a problem.

Common patterns:

- Brute Force
- Two Pointers
- Sliding Window
- Fast and Slow pointers
- Merge Intervals
- Cyclic Sort
- DFS (Depth First Search)
- BFS (Breadth First Search)
- Recursion Helpers
- Memoization
- Divide and Conquer
    - Mergesort
    - Quicksort
    - Nearest Neighbors
    - Matrix Manipulation
- Decrease and Conquer
    - Binary search
    - Selection Sort
    - Insertion Sort
    - Largest Number
- Dynamic Programming
- Backtracking
- The Greedy Method

Two Pointer Pattern

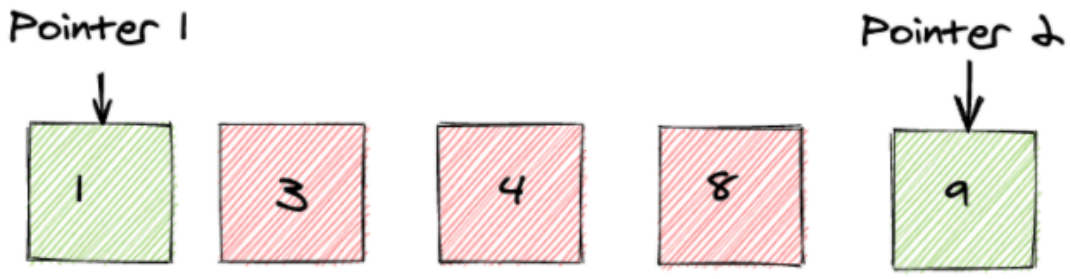
Creating pointers or values that correspond to an index or position and move towards the beginning, end or middle on a certain condition.

But what are pointers? In computer science, a pointer is a reference to an object. In many programming languages, that object stores a memory address of another value located in computer memory, or in some cases, that of memory-mapped computer hardware.

Two pointer is able to process two elements per loop instead of just one. Common patterns in the two-pointer approach entail:

- Two pointers, each starting from the beginning and the end until they both meet.
- One pointer moving at a slow pace, while the other pointer moves at twice the speed.

These patterns can be used for string or array questions. They can also be streamlined and made more efficient by iterating through two parts of an object simultaneously.



Algorithm #1: Two Sum (aka Pair with Target Sum)

Write a function that takes two inputs (array and target sum). Have the function return the **indexes** of a pair of values that when summed equal to the target sum input. The array of numbers will be **ordered**.

Input: [1,2,3,4,6], target=6
Output: [1,3]
Explanation: The numbers 2 and 4 add up to 6

Time complexity - O(n)
Space complexity - O(1)

The Brute Force pattern solution would be:

```
function twoSum(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 [-1, -1];
}
```

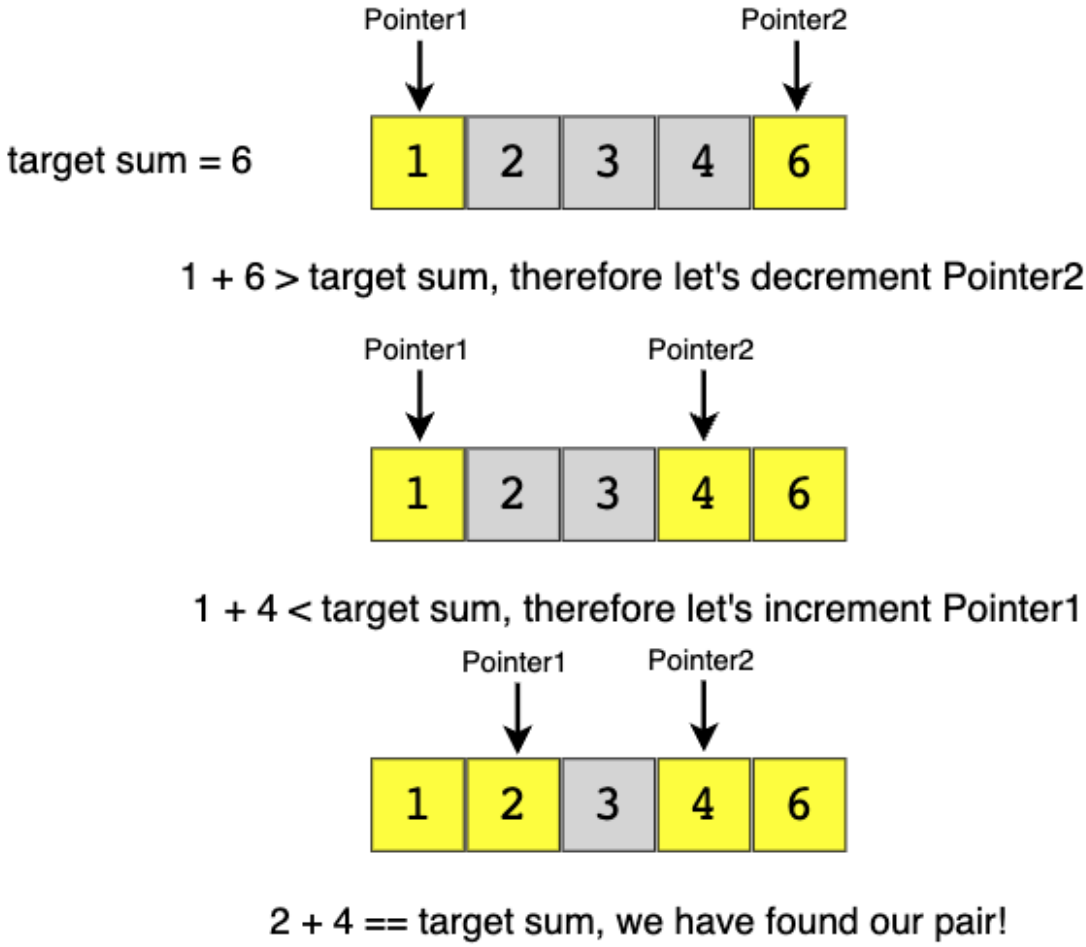
visualize code

Since we have two nested loops our time complexity would be O(n^2).

If we use the Two Pointer pattern we can get our time complexity to O(n).

We will start with one pointer pointing to the beginning of the array and another pointing at the end. At every step, we will see if the numbers pointed by the two pointers add up to the target sum. If they do, we have found our pair; otherwise, we will do one of two things:

- If the sum of the two numbers pointed by the two pointers is **greater** than the target sum, this means that we need a pair with a smaller sum. So, to try more pairs, we can **decrement the end-pointer**.
- If the sum of the two numbers pointed by the two pointers is **smaller** than the target sum, this means that we need a pair with a larger sum. So, to try more pairs, we can **increment the start-pointer.**



Note: The two-pointer approach would not work on this algorithm unless the array was sorted. If the array was not sorted, a sorting algorithm pattern would need to be applied first.

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

   if (targetSum > currentSum) {
     left += 1;
   } else {
     right -= 1;
   }
 }
 return [-1, -1];
}

console.log(twoSum([1, 2, 3, 4, 6], 6));
```

[visualize code]

Algorithm #2: Reverse a string without using array.reverse()

Time complexity - O(n)

Space complexity - O(1)

```
function reverseString(str) {
  let newString = '';
  for (let i = str.length-1; i >=0; i--) {
    newString += str[i];
  }
  return newString;
}
```

Can We Do Better Than Brute Force?

When trying to make something more efficient, it helps to think of things to cut or reduce.

- One thing to note is that we're going through the entire string-- do we truly need to iterate through every single letter?

- Let's examine the worst case scenario. What if the string is a million characters long? That would be a million operations to work through! Can we improve it? Yes, With More Pointers.

We're only working with **a single pointer** right now.

If we implement the Two Pointer Technique, we can cut the number of operations in half.

At each iteration, we can swap the letters at the pointer indices. After

swapping, we would increment the left pointer while decrementing the right one.

If n is the length of the string, we'll end up making n/2 swaps.

So despite requiring half the number operations-- a 4-character string would require 2 swaps with the two-pointer method. But an 8-character string would require 4 swaps.

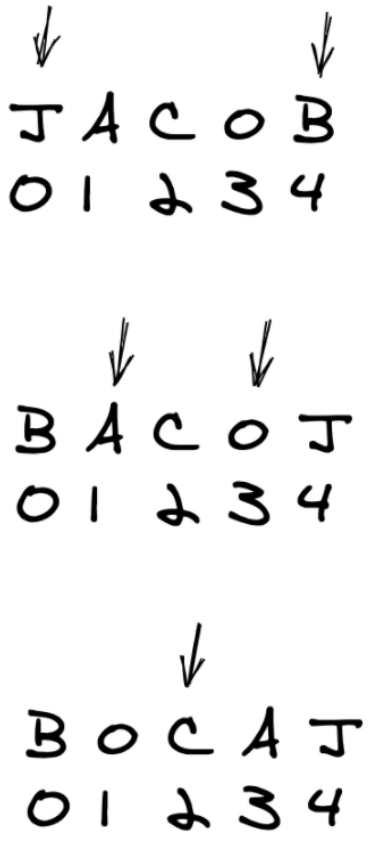
The input doubled, and so did the number of operations.

```
function reverseString(str) {

  let strArr = str.split("");
  let start = 0;
  let end = str.length - 1;

  while (start <= end) {
    const temp = strArr[start];
    strArr[start] = strArr[end];
    strArr[end] = temp;
    start++;
    end--;
  }
  return strArr.join("");
}

console.log(reverseString("JACOB"));
```



visualize code

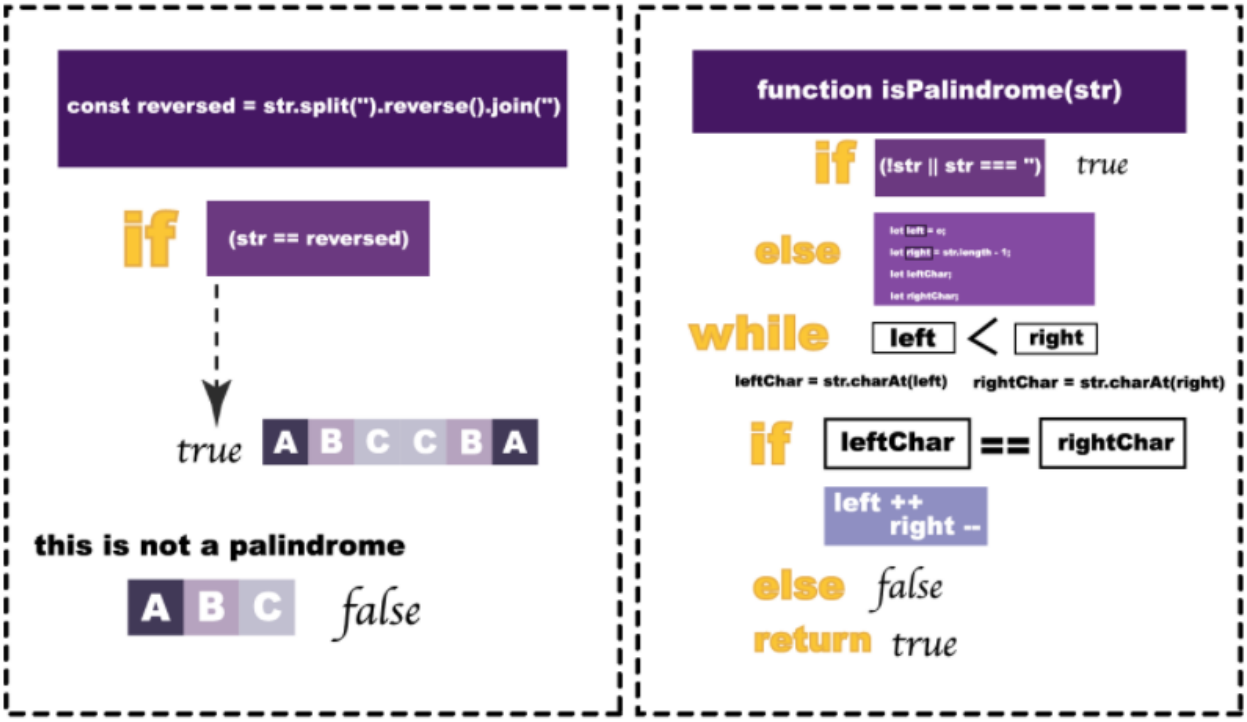
With two pointers, we've cut the number of operations in half. However, similar to the brute force, **the time complexity is still O(n).**

Algorithm #3: Validate a Palindrome

Given a string **str,** write a function that will return True if it's a palindrome and False if it is not. Assume that we **will not** have input strings that contain special characters or spaces.

Time complexity - O(n)
Space complexity - O(1)

This is a classic question, and there are multiple ways to solve this.



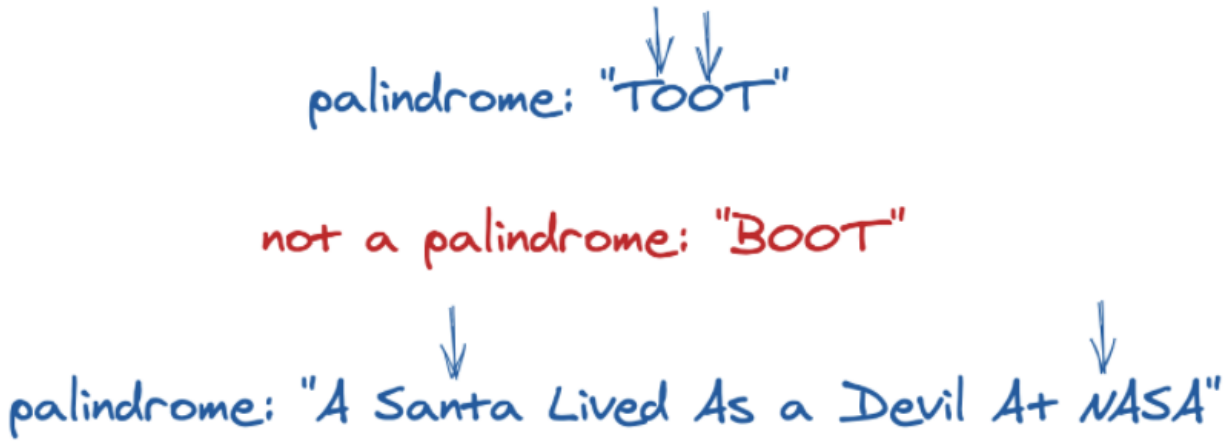
This would probably be invalid in an actual interview, but you can rely on the built-in String method to accomplish a quick reversal.

```
function isPalindrome(str) {
    // Calling reverse function
    const reversed = str.split('').reverse().join('');

    // Checking if both strings are equal or not
    if (str == reversed) {
        return true;
    }
    return false;
}
```

With a while loop:

We can cut down on the number of operations by recognizing that we don't need to do len(str)-1 iterations. Instead of using just one pointer that simply iterates through the string from its end, why not use two?



```
function isPalindrome(str) {
  let left = 0;
  let right = str.length - 1;
  let leftChar;
  let rightChar;

  while (left < right) {
    leftChar = str.charAt(left);
    rightChar = str.charAt(right);

    if (leftChar == rightChar) {
      left++;
      right--;
    } else {
      return false;
    }
  }

  return true;
}

console.log(isPalindrome('racecar'));
```

visualize code

We are specifying two pointers, start and end. start points to the beginning of the string, and end is a pointer to the last character. We iterate n/2 times

until the left pointer is greater than the right pointer for linear O(n) time complexity, and we use a constant O(1) space for the 2 pointers.

Algorithm #4: Detect Substring in a String

Write a function that takes two strings as input (str and subStr). The function will detect if the substring (second input string) is contained in the first input string. If the substring can be found in the string, return the index at which it starts, otherwise return -1.

Note: Do not use the native String class built-in substring or substr methods. But you can use the String class length property.

Time complexity - O(n)
Space complexity - O(1)

What if we wanted to know if “graph**”** was in “geography”?

Here, we can use the two-pointer technique. We use the 2nd pointer for the purpose of seeing how far we can "extend our match" when we find an initial one?

1. g matches g, so j = 1
2. e does not match r, so j = 0
3. o does not match g, so j = 0
4. g does match g, so j = 1
5. r does match r, so j = 2

6. a does match a, so j = 3 and so on...

At the same time, we also need to keep track of the actual index that the match starts. The j index will help us get the matching, but we need a **idxOfStart** variable for just the start.

Complexity of solution:

Let n be the length of the string. In the worst-case scenario, we iterate through all n characters of the string and do not find a match. Many of us will then declare that the time complexity is linear O(n). But think carefully, for each partial matching of substrings, we are traversing through the substring, and then again resetting to the initial pointer i.

So the time complexity will be **O(mn)** where m and n are the lengths of the string and substring. For memory, using the two-pointer method, we use a constant O(1) space for our solution.

```
function detectSubstring(str, subStr) {
  let idxOfStart = 0;
  j = 0;

  for (i = 0; i < str.length; i++) {
    if (str[i] == subStr[j]) {
      j++;
      if (j == subStr.length) {
        return i - (subStr.length - 1);
      }
    } else {
      i -= j;
      j = 0;
    }
  }

  return -1;
}

console.log(detectSubstring("geography", "graph"));
```

visualize code

Algorithm #5: Remove Duplicates

Write a function that gets an array of **sorted numbers**, and it removes all duplicates **in-place**. The function will **return the length** of the subarray that has no duplicates.

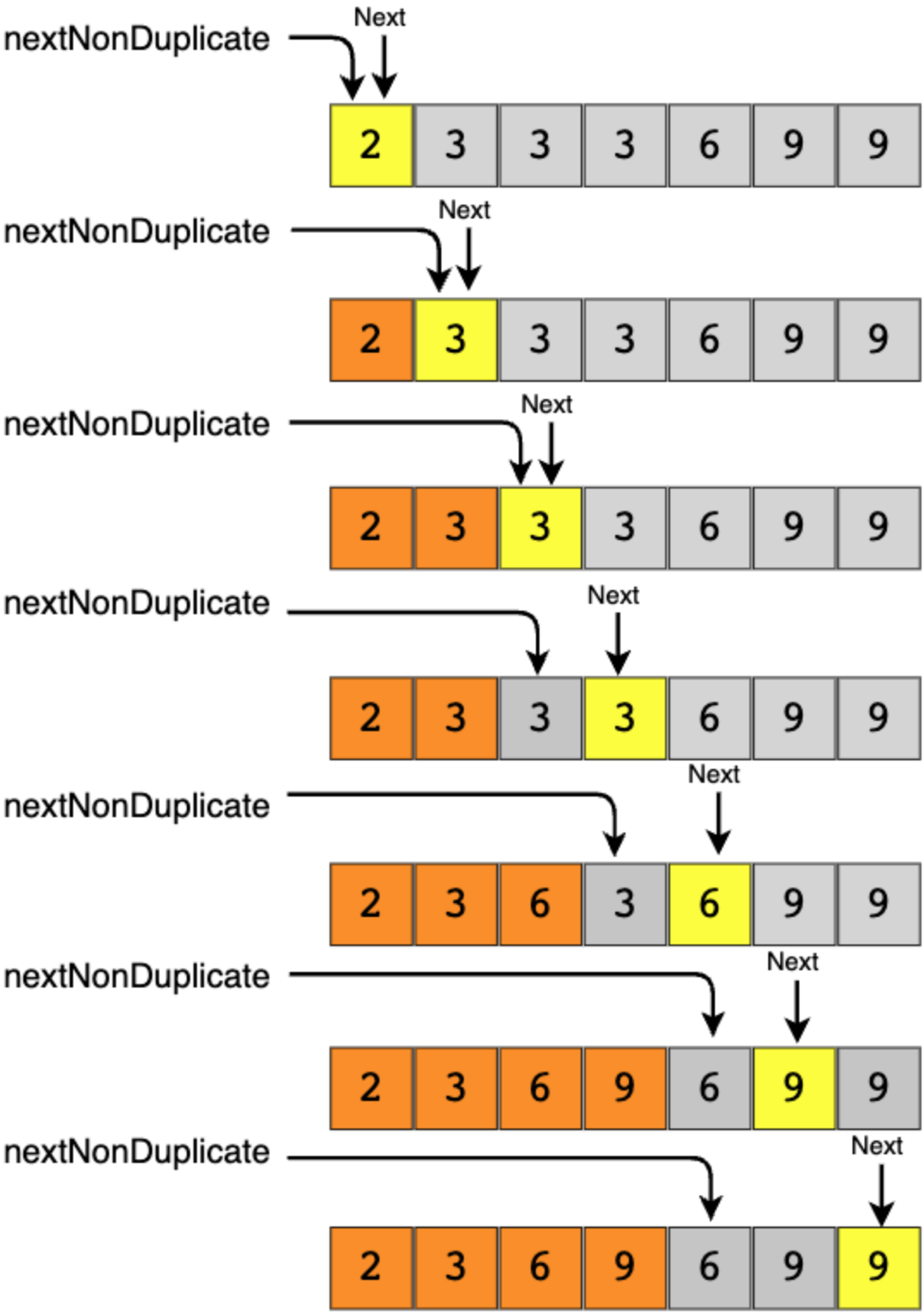
Input: [2, 3, 3, 3, 6, 9, 9]
Output: 4
Explanation: Non-duplicate array: [2, 3, 6, 9]

Input: [2, 2, 2, 11]
Output: 2
Explanation: Non-duplicate array: [2, 11]

Time complexity - O(n)
Space complexity - O(1)

We need to remove the duplicates in-place such that the resultant length of the array remains sorted.

As the **input array is sorted**, we can shift the elements left whenever we encounter duplicates. We will keep one pointer for iterating the array and one pointer for placing the next non-duplicate number. Our algorithm will be to iterate the array and whenever we see a non-duplicate number we move it next to the last non-duplicate number we've seen.


nextNonDuplicate
Next
2 3 3 3 6 9 9
nextNonDuplicate
Next
2 3 3 3 6 9 9
nextNonDuplicate
Next
2 3 3 3 6 9 9
nextNonDuplicate
Next
2 3 3 3 6 9 9
nextNonDuplicate
Next
2 3 6 3 6 9 9
nextNonDuplicate
Next
2 3 6 9 6 9 9
nextNonDuplicate
Next
2 3 6 9 6 9 9

```
function remove_duplicates(arr) {
 let nextNonDuplicate = 1;
 let i = 1;
 while (i < arr.length) {
   if (arr[nextNonDuplicate - 1] !== arr[i]) {
     arr[nextNonDuplicate] = arr[i];
     nextNonDuplicate += 1;
   }
   i += 1;
 }
 return nextNonDuplicate;
}

console.log(remove_duplicates([2, 3, 3, 3, 6, 9, 9]));
console.log(remove_duplicates([2, 2, 2, 11]));
```

visualize code