Pattern 1: Sliding Window

In many problems dealing with an array (or a LinkedList), we are asked to find or calculate something among all the contiguous subarrays (or sublists) of a given size. For example, take a look at this problem:

Find Averages of Sub Arrays

https://leetcode.com/problems/maximum-average-subarray-i/

> Given an array, find the average of all contiguous subarrays of size 'K' in it.

Let's understand this problem with a real input:

```
Array: [1, 3, 2, 6, -1, 4, 1, 8, 2], K=5
```

A **brute-force** algorithm will calculate the sum of every 5-element contiguous subarray of the given array and divide the sum by '5' to find the average.

```
function findAvgOfSubarrays(arr, K) {
  const results = []

  for(let i = 0; i < arr.length - K + 1; i++) {
    let sum = 0

    for(let j = i; j < i + K; j++) {
      sum += arr[j]
    }
    results.push(sum/K)
  }
  return results
}


findAvgOfSubarrays([1, 3, 2, 6, -1, 4, 1, 8, 2], 5)
```

Time complexity: Since for every element of the input array, we are calculating the sum of its next `'K'` elements, the time complexity of the above algorithm will be `O(N*K)` where `'N'` is the number of elements in the input array.

Can we find a better solution? Do you see any inefficiency in the above approach?

The inefficiency is that for any two consecutive subarrays of size `'5'`, the overlapping part (which will contain four elements) will be evaluated twice.

The efficient way to solve this problem would be to visualize each contiguous subarray as a sliding window of `'5'` elements. This means that we will slide the window by one element when we move

on to the next subarray. To reuse the sum from the previous subarray, we will subtract the element going out of the window and add the element now being included in the sliding window. This will save us from going through the whole subarray to find the sum and, as a result, the algorithm complexity will reduce to `O(N)`.

Here is the algorithm for the **Sliding Window** approach:

```
function findAveragesOfSubarrays(arr, k) {
  //sliding window approach

  const results = []
  let windowSum = 0
  let windowStart = 0

  for(let windowEnd = 0; windowEnd < arr.length; windowEnd++) {
    //add the next element
    windowSum += arr[windowEnd]

    //slide the window forward
    //we don't need to slide if we have not hit the required window size of k

    if (windowEnd >= k - 1) {
      //we are **AUTOMATICALLY** returning the window average once we hit the window s
      //and pushing to the output array
      results.push(windowSum/k)

      //subtracting the element going out
      windowSum -= arr[windowStart]

      //then sliding the window forward
      windowStart++

      //adding the element coming in, in the outer/previous loop
      //and repeating this process until we hit the end of the array
    }
  }
  return results
}

findAveragesOfSubarrays([1, 3, 2, 6, -1, 4, 1, 8, 2], 5)//[2.2, 2.8, 2.4, 3.6, 2.8]
```

Maximum Sum Subarray of Size K (easy)

https://leetcode.com/problems/largest-subarray-length-k/

> Given an array of positive numbers and a positive number `'k'`, find the maximum sum of any contiguous subarray of size `'k'`.

Brute Force

A basic brute force solution will be to calculate the sum of all `‘k’` sized subarrays of the given array to find the subarray with the highest sum. We can start from every index of the given array and add the next `‘k’` elements to find the subarray’s sum.

```
function maxSubarrayOfSizeK(arr, k) {
  //brute force
  let maxSum = 0
  let windowSum = 0

  //loop through array
  for(let i = 0; i < arr.length -k + 1; i++) {

    //keep track of sum in current window
    windowSum = 0
    for(let j = i; j < i + k; j++) {
      windowSum += arr[j]
    }

    //if currentWindowSum is > maxWindowSum
    //set currentWindwoSum to maxWindowSum
    maxSum = Math.max(maxSum, windowSum)
  }
  return maxSum
}

max_sub_array_of_size_k(3, [2, 1, 5, 1, 3, 2])//9
max_sub_array_of_size_k(2, [2, 3, 4, 1, 5])//7
```

- Time complexity will be `O(N*K)`, where `N` is the total number of elements in the given array

Sliding Window Approach

If you observe closely, you will realize that to calculate the sum of a contiguous subarray, we can utilize the sum of the previous subarray. For this, consider each subarray as a **Sliding Window** of size `‘k’`. To calculate the sum of the next subarray, we need to slide the window ahead by one element. So to slide the window forward and calculate the sum of the new position of the sliding window, we need to do two things:

1. Subtract the element going out of the sliding window, i.e., subtract the first element of the window.
2. Add the new element getting included in the sliding window, i.e., the element coming right after the end of the window.

This approach will save us from re-calculating the sum of the overlapping part of the sliding window.

```
function maxSubarrayOfSizeK(arr, k) {
  //sliding window
```

```
    let maxSum = 0
    let windowSum = 0
    let windowStart = 0

    //loop through array
    for(let windowEnd = 0; windowEnd < arr.length; windowEnd++) {
      //add the next element
      windowSum += arr[windowEnd]

      //slide the window, we dont need to slid if we
      //haven't hit the required window size of 'k'
      if(windowEnd >= k -1) {
        maxSum = Math.max(maxSum, windowSum)

        //subtract the element going out
        windowSum -= arr[windowStart]

        //slide the window ahead
        windowStart ++
      }
    }
    return maxSum
  }


  maxSubarrayOfSizeK([2, 1, 5, 1, 3, 2], 3)//9
  maxSubarrayOfSizeK([2, 3, 4, 1, 5], 2)//7
```

- The time complexity of the above algorithm will be `O(N)`
- The space complexity of the above algorithm will be `O(1)`

Smallest Subarray with a given sum (easy)

https://leetcode.com/problems/minimum-size-subarray-sum/

> Given an array of positive numbers and a positive number ‘S,’ find the length of the **smallest contiguous subarray whose sum is greater than or equal to ‘S’**.
>
> Return 0 if no such subarray exists.

This problem follows the Sliding Window pattern, and we can use a similar strategy as discussed in **Maximum Sum Subarray of Size K**. There is one difference though: in this problem, the sliding window size is not fixed. Here is how we will solve this problem:

1. First, we will add-up elements from the beginning of the array until their sum becomes greater than or equal to `‘S’`.
2. These elements will constitute our sliding window. We are asked to find the smallest such window having a sum greater than or equal to `‘S’`. We will remember the length of this window as the smallest window so far.

3. After this, we will keep adding one element in the sliding window (i.e., slide the window ahead) in a stepwise fashion.
4. In each step, we will also try to shrink the window from the beginning. We will shrink the window until the window's sum is smaller than `'S'` again. This is needed as we intend to find the smallest window. This shrinking will also happen in multiple steps; in each step, we will do two things:

- Check if the current window length is the smallest so far, and if so, remember its length.
- Subtract the first element of the window from the running sum to shrink the sliding window.

```
function smallestSubarrayWithGivenSum(arr, s) {
  //sliding window, BUT the window size is not fixed
  let windowSum = 0
  let minLength = Infinity
  let windowStart = 0

  //First, we will add-up elements from the beginning of the array until their sum bec
  for(windowEnd = 0; windowEnd < arr.length; windowEnd++) {

    //add the next element
    windowSum += arr[windowEnd]

    //shrink the window as small as possible
    //until windowSum is small than s
    while(windowSum >= s) {
      //These elements will constitute our sliding window. We are asked to find the sm
      //After this, we will keep adding one element in the sliding window (i.e., slide
      //In each step, we will also try to shrink the window from the beginning. We wil
      //Check if the current window length is the smallest so far, and if so, remember
      minLength = Math.min(minLength, windowEnd - windowStart + 1)

      //Subtract the first element of the window from the running sum to shrink the sl
      windowSum -= arr[windowStart]
      windowStart++
    }
  }

  if(minLength === Infinity) {
    return 0
  }
  return minLength
}


smallestSubarrayWithGivenSum([2, 1, 5, 2, 3, 2], 7)//2
smallestSubarrayWithGivenSum([2, 1, 5, 2, 8], 7)//1
smallestSubarrayWithGivenSum([3, 4, 1, 1, 6], 8)//3
```

- The time complexity of the above algorithm will be `O(N)`. The outer for loop runs for all elements, and the inner while loop processes each element only once; therefore, the time complexity of the algorithm will be `O(N+N)`), which is asymptotically equivalent to `O(N)`.
- The algorithm runs in constant space `O(1)`.

Longest Substring with K Distinct Characters (medium)

https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters/

> Given a string, find the length of the **longest substring** in it with **no more than `K` distinct characters**.
>
> You can assume that K is less than or equal to the length of the given string.

This problem follows the Sliding Window pattern, and we can use a similar dynamic sliding window strategy as discussed in **Smallest Subarray with a given sum**. We can use a HashMap to remember the frequency of each character we have processed. Here is how we will solve this problem:

1. First, we will insert characters from the beginning of the string until we have `K` distinct characters in the HashMap.
2. These characters will constitute our sliding window. We are asked to find the longest such window having no more than `K` distinct characters. We will remember the length of this window as the longest window so far.
3. After this, we will keep adding one character in the sliding window (i.e., slide the window ahead) in a stepwise fashion.
4. In each step, we will try to shrink the window from the beginning if the count of distinct characters in the HashMap is larger than `K`. We will shrink the window until we have no more than `K` distinct characters in the HashMap. This is needed as we intend to find the longest window.
5. While shrinking, we'll decrement the character's frequency going out of the window and remove it from the HashMap if its frequency becomes zero.
6. At the end of each step, we'll check if the current window length is the longest so far, and if so, remember its length.

```
function longestSubstringWithKdistinct(str, k) {
   // Given a string, find the length of the longest substring in it with no more than
  let windowStart = 0
  let maxLength = 0
  let charFrequency = {}

  //in the following loop we'll try to extend the range [windowStart, windowEnd]
  for(let windowEnd = 0; windowEnd < str.length; windowEnd++) {
    const endChar = str[windowEnd]
    if(!(endChar in charFrequency)) {
```

```
      charFrequency[endChar] = 0
    }
    charFrequency[endChar]++
    //shrink the window until we are left with k distinct characters
    //in the charFrequency Object

    while(Object.keys(charFrequency).length > k) {
      //insert characters from the beginning of the string until we have 'K' distinct
    //these characters will consitutue our sliding window.  We are asked to find the l
    //we will keep adding on character in the sliding window in a stepwise fashion
      //in each step we will try to shrink the window from the beginning if the count
      const startChar = str[windowStart]
      charFrequency[startChar]--
      //while shrinking , we will decrement the characters frequency going out of the
      if(charFrequency[startChar] === 0) {
        delete charFrequency[startChar]
      }
      windowStart++
    }
    //after each step we will check if the current window length is the longest so far
    maxLength = Math.max(maxLength, windowEnd - windowStart + 1)
  }
    return maxLength
};

longestSubstringWithKdistinct("araaci", 2)//4, The longest substring with no more than
longestSubstringWithKdistinct("araaci", 1)//2, The longest substring with no more than
longestSubstringWithKdistinct("cbbebi", 3)//5, The longest substrings with no more tha
```

- The above algorithm's time complexity will be `O(N)`, where `N` is the number of characters in the input string. The outer for loop runs for all characters, and the inner while loop processes each character only once; therefore, the time complexity of the algorithm will be `O(N+N)`, which is asymptotically equivalent to `O(N)`
- The algorithm's space complexity is `O(K)`, as we will be storing a maximum of `K+1` characters in the HashMap.

Fruits into Baskets (medium)

https://leetcode.com/problems/fruit-into-baskets/

> Given an array of characters where each character represents a fruit tree, you are given **two baskets**, and your goal is to put the **maximum number of fruits in each basket**. The only restriction is that **each basket can have only one type of fruit**.
>
> You can start with any tree, but you can't skip a tree once you have started. You will pick one fruit from each tree until you cannot, i.e., you will stop when you have to pick from a third fruit type.
>
> Write a function to return the maximum number of fruits in both baskets.

This problem follows the Sliding Window pattern and is quite similar to **Longest Substring with K Distinct Characters**.

In this problem, we need to find the length of the longest subarray with no more than two distinct characters (or fruit types!).

This transforms the current problem into Longest Substring with **K Distinct Characters** where `K=2`.

```
function fruitsInBaskets(fruits) {
  let windowStart = 0;
  let maxLength = 0;
  let fruitFrequency = {};

  //try to extend the range
  for(let windowEnd = 0; windowEnd < fruits.length; window++) {
    const endFruit = fruits[windowEnd]
    if(!(endFruit in fruitFrequency)) {
      fruitFrequency[endFruit] = 0
    }
    fruitFrequency[endFruit]++

    //shrink the sliding window, until we are left with '2' fruits in the fruitFrequen
    while(Object.keys(fruitFrequency).length > 2) {
      const startFruit = fruits[windowStart];
      fruitFrequency[startFruit]--
      if(fruitFrequency[startFruit] === 0) {
        delete fruitFrequency[startFruit]
      }
      windowStart++
    }
    maxLength = Math.max(maxLength, windowEnd - windowStart + 1)
  }
  return maxLength
}

fruitsInBaskets(['A', 'B', 'C', 'A', 'C'])//3 , We can put 2 'C' in one basket and one
fruitsInBaskets(['A', 'B', 'C', 'B', 'B', 'C'])//5 , We can put 3 'B' in one basket an
```

- The above algorithm's time complexity will be `O(N)`, where `'N'` is the number of characters in the input array. The outer `for` loop runs for all characters, and the inner `while` loop processes each character only once; therefore, the time complexity of the algorithm will be `O(N+N)`, which is asymptotically equivalent to `O(N)`.
- The algorithm runs in constant space `O(1)` as there can be a maximum of three types of fruits stored in the frequency map.

Longest Substring with at most 2 distinct characters

https://leetcode.com/problems/longest-substring-with-at-most-two-distinct-characters/

> Given a string, find the length of the longest substring in it with at most two distinct characters.

```
function lengthOfLongestSubstringTwoDistinct(s) {
    let windowStart = 0
    let maxLength = 0
    let charFreq = {}

    //try to extend the range
    for(let windowEnd = 0; windowEnd < s.length; windowEnd++) {
        const endChar = s[windowEnd]

        if(!(endChar in charFreq)) {
            charFreq[endChar] = 0
        }
        charFreq[endChar]++

      //shrink the sliding window, until we are left
      //with 2 chars in charFreq hashMap

      while(Object.keys(charFreq).length > 2) {
        const startChar = s[windowStart]
        charFreq[startChar]--
        if(charFreq[startChar] === 0) {
          delete charFreq[startChar]
        }
        windowStart++
      }
      maxLength = Math.max(maxLength, windowEnd - windowStart + 1)
    }

    return maxLength
};

lengthOfLongestSubstringTwoDistinct('eceba')//3
lengthOfLongestSubstringTwoDistinct('ccaabbb')//5
```

No-repeat Substring (hard)

https://leetcode.com/problems/longest-substring-without-repeating-characters/

> Given a string, find the **length of the longest substring**, which has **no repeating characters**.

This problem follows the **Sliding Window pattern**, and we can use a similar dynamic sliding window strategy as discussed in **Longest Substring with K Distinct Characters**. We can use a HashMap to remember the last index of each character we have processed. Whenever we get a repeating character, we will shrink our sliding window to ensure that we always have distinct characters in the sliding window.

```
function nonRepeatSubstring(str) {
  // sliding window with hashmap

  let windowStart = 0
  let maxLength = 0
  let charIndexMap = {}

  //try to extend the range [windowStart, windowEnd]
  for(let windowEnd = 0; windowEnd < str.length; windowEnd++) {
    const endChar = str[windowEnd]

    //if the map already contains the endChar,
    //shrink the window from the beginning
    //so that we only have on occurance of endChar
    if(endChar in charIndexMap) {

      //this is tricky; in the current window,
      //we will not have any endChar after
      //it's previous index. and if windowStart
      //is already ahead of the last index of
      //endChar, we'll keep windowStart
      windowStart = Math.max(windowStart, charIndexMap[endChar] + 1)
    }

    //insert the endChar into the map
    charIndexMap[endChar] = windowEnd

    //remember the maximum length so far
    maxLength = Math.max(maxLength, windowEnd - windowStart+1)
  }
  return maxLength
};

nonRepeatSubstring("aabccbb")//3
nonRepeatSubstring("abbbb")//2
nonRepeatSubstring("abccde")//3
```

- The above algorithm's time complexity will be `O(N)`, where `'N'` is the number of characters in the input string.
- The algorithm's space complexity will be `O(K)`, where `K` is the number of distinct characters in the input string. This also means `K<=N`, because in the worst case, the whole string might not have any repeating character, so the entire string will be added to the HashMap. Having said that, since we can expect a fixed set of characters in the input string (e.g., 26 for English letters), we can say that the algorithm runs in fixed space `O(1)`; in this case, we can use a fixed-size array instead of the HashMap.

Longest Substring with Same Letters after Replacement (hard)

https://leetcode.com/problems/longest-repeating-character-replacement/

> Given a string with lowercase letters only, if you are allowed to **replace no more than 'k' letters** with any letter, find the **length of the longest substring having the same letters** after replacement.

This problem follows the **Sliding Window pattern**, and we can use a similar dynamic sliding window strategy as discussed in **No-repeat Substring**. We can use a HashMap to count the frequency of each letter.

- We will iterate through the string to add one letter at a time in the window.
- We will also keep track of the count of the maximum repeating letter in any window (let's call it `maxRepeatLetterCount`).
- So, at any time, we know that we do have a window with one letter repeating `maxRepeatLetterCount` times; this means we should try to replace the remaining letters.
  - If the remaining letters are less than or equal to `'k'`, we can replace them all.
  - If we have more than `'k'` remaining letters, we should shrink the window as we cannot replace more than `'k'` letters.

While shrinking the window, we don't need to update `maxRepeatLetterCount` (hence, it represents the maximum repeating count of ANY letter for ANY window). Why don't we need to update this count when we shrink the window? Since we have to replace all the remaining letters to get the longest substring having the same letter in any window, we can't get a better answer from any other window even though all occurrences of the letter with frequency `maxRepeatLetterCount` is not in the current window.

```
function lengthOfLongestSubstring(str, k) {
  let windowStart = 0
  let maxLength = 0
  let maxRepeatLetterCount = 0
  let charFrequency = {}

  //Try to extend the range [windowStart, windowEnd]
  for(let windowEnd = 0; windowEnd < str.length; windowEnd++) {
    const endChar = str[windowEnd]
    if(!(endChar in charFrequency)) {
      charFrequency[endChar] = 0
    }
    charFrequency[endChar]++
    //*REVIEW THIS LINE*
    maxRepeatLetterCount = Math.max(maxRepeatLetterCount, charFrequency[endChar])

    //current window size is from windowStart to windowEnd, overall we have a letter w
    //repeating maxRepeatLetterCount times, this mean we can have a window which has o
    //repeating maxRepeatLetterCount times and the remaining letters we should replace
    //if the remaining letters are more than k, it is the time to shrink the window as
    //are not allowed to replace more than k letters
    if((windowEnd - windowStart + 1 - maxRepeatLetterCount) > k) {
      const startChar = str[windowStart]
      charFrequency[startChar]--
      windowStart++
```

```
    }
    maxLength = Math.max(maxLength, windowEnd - windowStart + 1)
  }
  return maxLength
}

lengthOfLongestSubstring("aabccbb", 2)//5, Replace the two 'c' with 'b' to have a long
lengthOfLongestSubstring("abbcb", 1)//4, Replace the 'c' with 'b' to have a longest re
lengthOfLongestSubstring("abccde", 1)//3, Replace the 'b' or 'd' with 'c' to have the
```

- The above algorithm's time complexity will be `O(N)`, where `'N'` is the number of letters in the input string.
- As we expect only the lower case letters in the input string, we can conclude that the space complexity will be `O(26)` to store each letter's frequency in the HashMap, which is asymptotically equal to `O(1)`.

Longest Subarray with Ones after Replacement (hard)

https://leetcode.com/problems/max-consecutive-ones-iii/

> Given an array containing 0s and 1s, if you are allowed to **replace no more than 'k' 0s with 1s**, find the length of the **longest contiguous subarray having all 1s**.

This problem follows the **Sliding Window** pattern and is quite similar to **Longest Substring with same Letters after Replacement**. The only difference is that, in the problem, we only have two characters (1s and 0s) in the input arrays.

Following a similar approach, we'll iterate through the array to add one number at a time in the window. We'll also keep track of the maximum number of repeating 1s in the current window (let's call it `maxOnesCount`). So at any time, we know that we can have a window with 1s repeating `maxOnesCount` time, so we should try to replace the remaining 0s. If we have more than `'k'` remaining 0s, we should shrink the window as we are not allowed to replace more than `'k'` 0s.

```
function lengthOfLongestSubstring (arr, k) {
  let windowStart = 0
  let maxLength = 0
  let maxOnesCount = 0

  //Try to extend the range [windowStart, windowEnd]
  for(let windowEnd = 0; windowEnd < arr.length; windowEnd++) {
    if(arr[windowEnd] === 1) {
      maxOnesCount++
    }

    //current window size is from windowStart to windowEnd, overall we have a
    //maximum of 1s repeating maxOnesCount times, this means we can have a window
    //with maxOnesCount 1s and the remaining are 0s which should replace with 1s
    //now, if the remaining 0s are more that k, it is the time to shrink the
```

```
      //window as we are not allowed to replace more than k 0s
      if((windowEnd - windowStart + 1 - maxOnesCount) > k) {
        if(arr[windowStart] === 1) {
          maxOnesCount--
        }
        windowStart++
      }

      maxLength = Math.max(maxLength, windowEnd - windowStart + 1)
    }

    return maxLength
  }

  lengthOfLongestSubstring ([0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1], 2)//6, Replace the '0' at
  lengthOfLongestSubstring ([0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1], 3)//9, Replace the
```

- The above algorithm's time complexity will be `O(N)`, where `'N'` is the count of numbers in the input array.
- The algorithm runs in constant space `O(1)`.

🌟 Permutation in a String (hard)

https://leetcode.com/problems/permutation-in-string/

> Given a string and a pattern, find out if the **string contains any permutation of the pattern**.

Permutation is defined as the re-arranging of the characters of the string. For example, `"abc"` has the following six permutations:

- abc
- acb
- bac
- bca
- cab
- cba

If a string has `'n'` distinct characters, it will have `n!` permutations.

This problem follows the **Sliding Window** pattern, and we can use a similar sliding window strategy as discussed in **Longest Substring with K Distinct Characters**. We can use a HashMap to remember the frequencies of all characters in the given pattern. Our goal will be to match all the characters from this HashMap with a sliding window in the given string. Here are the steps of our algorithm:

- Create a HashMap to calculate the frequencies of all characters in the pattern.
- Iterate through the string, adding one character at a time in the sliding window.

- If the character being added matches a character in the HashMap, decrement its frequency in the map. If the character frequency becomes zero, we got a complete match.
- If at any time, the number of characters matched is equal to the number of distinct characters in the pattern (i.e., total characters in the HashMap), we have gotten our required permutation.
- If the window size is greater than the length of the pattern, shrink the window to make it equal to the pattern's size. At the same time, if the character going out was part of the pattern, put it back in the frequency HashMap.

```
function findPermutation(str, pattern) {
  //sliding window
  let windowStart = 0
  let isMatch = 0
  let charFrequency = {}

 for(i = 0; i < pattern.length; i++) {
   const char = pattern[i]
   if(!(char in charFrequency)) {
     charFrequency[char] = 0
   }
   charFrequency[char]++
 }

  //our goal is to math all the characters from charFrequency with the current window
  //try to extend the range [windowStart, windowEnd]
  for(windowEnd = 0; windowEnd < str.length; windowEnd++) {
    const endChar = str[windowEnd]
    if(endChar in charFrequency) {
      //decrement the frequency of the matched character
      charFrequency[endChar]--
      if(charFrequency[endChar] === 0) {
        isMatch++
      }
    }
    if(isMatch === Object.keys(charFrequency).length) {
      return true
    }

    //shrink the sliding window
    if(windowEnd >= pattern.length - 1) {
      let startChar = str[windowStart]
      windowStart++
      if(startChar in charFrequency) {
        if(charFrequency[startChar] === 0) {
          isMatch--
        }
        charFrequency[startChar]++
      }
    }
  }
  return false
}
```

```
findPermutation("oidbcaf", "abc")//true, The string contains "bca" which is a permutat
findPermutation("odicf", "dc")//false
findPermutation("bcdxabcdy", "bcdxabcdy")//true
findPermutation("aaacb", "abc")//true, The string contains "acb" which is a permutatio
```

- The above algorithm's time complexity will be `O(N + M)`, where `'N'` and `'M'` are the number of characters in the input string and the pattern, respectively.
- The algorithm's space complexity is `O(M)` since, in the worst case, the whole pattern can have distinct characters that will go into the HashMap.

🌟 String Anagrams (hard)

https://leetcode.com/problems/find-all-anagrams-in-a-string/

> Given a string and a pattern, **find all anagrams of the pattern in the given string**.

Every **anagram** is a **permutation** of a string.

As we know, when we are not allowed to repeat characters while finding permutations of a string, we get `N!` permutations (or anagrams) of a string having `N` characters. For example, here are the six anagrams of the string `"abc"`:

- abc
- acb
- bac
- bca
- cab
- cba

> Write a function to return a list of starting indices of the anagrams of the pattern in the given string.

This problem follows the **Sliding Window** pattern and is very similar to **Permutation in a String**. In this problem, we need to find every occurrence of any permutation of the pattern in the string. We will use a list to store the starting indices of the anagrams of the pattern in the string.

```
function findStringAnagrams(str, pattern){
  let windowStart = 0, matched = 0, charFreq = {}

  for(let i = 0; i < pattern.length; i++){
    const char = pattern[i]
    if(!(char in charFreq)) {
      charFreq[char] = 0
    }
    charFreq[char]++
  }
```

```
  const resultIndex = []

  //our goal is to match all the characters from the charFreq
  //with the current window try to
  //extend the range [windowStart, windowEnd]
  for(let windowEnd = 0; windowEnd < str.length; windowEnd++) {
    const endChar = str[windowEnd]
    if(endChar in charFreq) {
      //decrement the frequency of matched character
      charFreq[endChar]--
      if(charFreq[endChar] === 0) {
        matched++
      }
    }

    if(matched === Object.keys(charFreq).length){
      //have we found an anagram
      resultIndex.push(windowStart)
    }

    //shrink the sliding window
    if(windowEnd >= pattern.length -1) {
      const startChar = str[windowStart]
      windowStart++
      if(endChar in charFreq) {
        if(charFreq[startChar] === 0) {
          //before putting the character back
          //decrement the matched count
          matched--
        }
        //put the character back
        charFreq[startChar]++
      }
    }
  }

  return resultIndex
}

findStringAnagrams('ppqp', 'pq')//[1,2], The two anagrams of the pattern in the given
findStringAnagrams('abbcabc', 'abc')//[2,3,4], The three anagrams of the pattern in th
```

- The time complexity of the above algorithm will be `O(N + M)` where `'N'` and `'M'` are the number of characters in the input string and the pattern respectively.
- The space complexity of the algorithm is `O(M)` since in the worst case, the whole pattern can have distinct characters which will go into the `HashMap` . In the worst case, we also need `O(N)` space for the result list, this will happen when the pattern has only one character and the string contains only that character.

🌟 Smallest Window containing Substring (hard)

https://leetcode.com/problems/minimum-window-substring/

> Given a string and a pattern, find the **smallest substring** in the given string which has **all the characters of the given pattern**.

This problem follows the **Sliding Window** pattern and has a lot of similarities with **Permutation in a String** with one difference. In this problem, we need to find a substring having all characters of the pattern which means that the required substring can have some additional characters and doesn't need to be a permutation of the pattern. Here is how we will manage these differences:

1. We will keep a running count of every matching instance of a character.
2. Whenever we have matched all the characters, we will try to shrink the window from the beginning, keeping track of the smallest substring that has all the matching characters.
3. We will stop the shrinking process as soon as we remove a matched character from the sliding window. One thing to note here is that we could have redundant matching characters, e.g., we might have two 'a' in the sliding window when we only need one 'a'. In that case, when we encounter the first 'a', we will simply shrink the window without decrementing the matched count. We will decrement the matched count when the second 'a' goes out of the window.

```
function findSubstring(str, pattern) {
  let windowStart = 0
  let matched = 0
  let substrStart = 0
  let minLength = str.length + 1
  let charFreq = {}

  for(let i = 0; i < pattern.length; i++) {
    const char = pattern[i]
    if(!(char in charFreq)) {
      charFreq[char] = 0
    }
    charFreq[char]++
  }

  //try to extend the range [windowStart, windowEnd]
  for(let windowEnd = 0; windowEnd < str.length; windowEnd++) {
    const endChar = str[windowEnd]
    if(endChar in charFreq) {
      charFreq[endChar]--
      if(charFreq[endChar] >= 0) {
        //count every matching of a character
        matched++
      }
    }

    //Shrink the window if we can, finish as soon as we remove a
    //matched character
    while(matched === pattern.length) {
      if(minLength > windowEnd - windowStart + 1) {
        minLength = windowEnd - windowStart + 1
```

```
          substrStart = windowStart
        }

        const startChar = str[windowStart]
        windowStart++
        if(startChar in charFreq) {
          if(charFreq[startChar] === 0) {
          matched--
        }
        charFreq[startChar]++

        }
      }
    }
    if(minLength > str.length) {
    return ''
  }
  return str.substring(substrStart, substrStart + minLength)
  }

  findSubstring("aabdec", "abc")//"abdec", The smallest substring having all characters
  findSubstring("abdbca", "abc")//"bca", The smallest substring having all characters of
  findSubstring("adcad", "abc")//"", No substring in the given string has all characters
```

- The time complexity of the above algorithm will be `O(N + M)` where `'N'` and `'M'` are the number of characters in the input string and the pattern respectively.
- The space complexity of the algorithm is `O(M)` since in the worst case, the whole pattern can have distinct characters which will go into the `HashMap` . In the worst case, we also need `O(N)` space for the resulting substring, which will happen when the input string is a permutation of the pattern.

🌟 Words Concatenation (hard)

https://leetcode.com/problems/concatenated-words/

Given a string and a list of words, find all the starting indices of substrings in the given string that are a **concatenation of all the given words** exactly once without any **overlapping of words**. It is given that all words are of the same length.

This problem follows the **Sliding Window** pattern and has a lot of similarities with **Maximum Sum Subarray of Size K**. We will keep track of all the words in a **HashMap** and try to match them in the given string. Here are the set of steps for our algorithm:

1. Keep the frequency of every word in a **HashMap**.
2. Starting from every index in the string, try to match all the words.
3. In each iteration, keep track of all the words that we have already seen in another **HashMap**.
4. If a word is not found or has a higher frequency than required, we can move on to the next character in the string.

5. Store the index if we have found all the words.

```
function findWordConcatenation(str, words) {
  if(words.length === 0 || words[0].length === 0) {
    return []
  }

  let wordFreq = {}

  words.forEach((word) => {
    if(!(word in wordFreq)) {
      wordFreq[word] = 0
    }
    wordFreq[word]++
  })

  const resultIndex = []
  let wordCount = words.length
  let wordLength = words[0].length

  for(let i = 0; i < (str.length - wordCount * wordLength) + 1; i++) {
    const wordsSeen = {}
    for(let j = 0; j < wordCount; j++) {
      let nextWordIndex = i + j * wordLength
      //get the next word from the string
      const word = str.substring(nextWordIndex, nextWordIndex + wordLength)
      if(!(word in wordFreq)){
        //break if we don't need this word
        break
      }

      //add the word ot the wordsSeen ma
      if(!(word in wordsSeen)){
        wordsSeen[word] = 0
      }
      wordsSeen[word]++

      //no need to process furrther if the word
      //has higher frequency than required
      if(wordsSeen[word] > (wordFreq[word] || 0)){
        break
      }

      if(j + 1 === wordCount){
        //store index if we have found all the words
        resultIndex.push(i)
      }
    }
  }
  return resultIndex
}

findWordConcatenation("catfoxcat", ["cat", "fox"])//[0, 3], The two substring containi
findWordConcatenation("catcatfoxfox", ["cat", "fox"])//[3], The only substring contain
```

- The time complexity of the above algorithm will be `O(N * M * Len)` where `'N'` is the number of characters in the given string, `'M'` is the total number of words, and `'Len'` is the length of a word.
- The space complexity of the algorithm is `O(M)` since at most, we will be storing all the words in the two **HashMaps**. In the worst case, we also need `O(N)` space for the resulting list. So, the overall space complexity of the algorithm will be `O(M+N)`.

Pattern 2: Two Pointers

In problems where we deal with sorted arrays (or LinkedLists) and need to find a set of elements that fulfill certain constraints, the Two Pointers approach becomes quite useful. The set of elements could be a pair, a triplet or even a subarray. For example, take a look at the following problem:

> Given an array of sorted numbers and a target sum, find a pair in the array whose sum is equal to the given target.

To solve this problem, we can consider each element one by one (pointed out by the first pointer) and iterate through the remaining elements (pointed out by the second pointer) to find a pair with the given sum. The time complexity of this algorithm will be `O(N^2)` where `'N'` is the number of elements in the input array.

Given that the input array is sorted, an efficient way would be to start with one pointer in the beginning and another pointer 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 not, we will do one of two things:

1. 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.
2. 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.

Pair with Target Sum aka "Two Sum" (easy) 🌴

https://leetcode.com/problems/two-sum/

> Given an array of sorted numbers and a target sum, find a pair in the array whose sum is equal to the given target.

Write a function to return the indices of the two numbers (i.e. the pair) such that they add up to the given target.

Since the given array is sorted, a brute-force solution could be to iterate through the array, taking one number at a time and searching for the second number through **Binary Search**. The time complexity of this algorithm will be `O(N*logN)`. Can we do better than this?

We can follow the **Two Pointers** approach. 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:

1. 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.
2. 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.

Brute Force Solution

```
function pair_with_targetsum(nums, target) {
  for (let i = 0; i < nums.length; i++) {
    for(let j = 1; j < nums.length; j++) {
      if((nums[i] + nums[j]) === target) {
        //they cannot be at the same index
        if(i !== j) {
          return [i, j]
        }
      }
    }
  }
}
```

Two pointer Solution

- Assume the input is sorted

```
function pairWithTargetSum(arr, target) {
  let start = 0
  let end = arr.length-1

  while(start < end) {
    let currentSum = arr[start] + arr[end]

    if(currentSum === target) {
      return [start, end]
    }

    if(currentSum < target) {
      start++
    } else {
      end--
    }
  }
  return 0
}

pairWithTargetSum([1, 2, 3, 4, 6], 6)//[1,3]
pairWithTargetSum([2, 5, 9, 11], 11)//[0,2]
```

- The time complexity of the above algorithm will be `O(N)`, where `'N'` is the total number of elements in the given array.
- The algorithm runs in constant space `O(1)`.

❗ Hash Table Solution

Instead of using a two-pointer or a binary search approach, we can utilize a **HashTable** to search for the required pair. We can iterate through the array one number at a time. Let's say during our iteration we are at number `'X'`, so we need to find `'Y'` such that `"X + Y == Target"`. We will do two things here:

1. Search for `'Y'` (which is equivalent to `"Target - X"`) in the HashTable. If it is there, we have found the required pair.
2. Otherwise, insert `"X"` in the HashTable, so that we can search it for the later numbers.

```
function pair_with_targetsum(nums, target) {
  //Instead of using a two-pointer or a binary search approach,
  //we can utilize a HashTable to search for the required pair.
  //We can iterate through the array one number at a time.
  //Let's say during our iteration we are at number 'X',
  //so we need to find 'Y' such that "X + Y == Target".

  //We will do two things here:
  const arr = {}
  for(let i = 0; i < nums.length; i ++){
    let item = nums[i]

    if(target - item in arr) {
      //1. Search for 'Y' (which is equivalent to "Target - X") in the HashTable.
      //If it is there, we have found the required pair
      return [arr[target - item], i]
    }
    arr[nums[i]] = i
    //2. Otherwise, insert "X" in the HashTable, so that we can search it for the late
  }
  return [-1, -1]
}

pair_with_targetsum([1, 2, 3, 4, 6], 6)//[1, 3]
pair_with_targetsum([2, 5, 9, 11], 11)//[0, 2]
pair_with_targetsum([2, 7, 11, 15], 9)//[0, 1]
pair_with_targetsum([3, 2, 4], 6)//[1, 2]
pair_with_targetsum([3, 3], 6)//[0, 1]
```

- The time complexity of the above algorithm will be `O(N)`, where `'N'` is the total number of elements in the given array.
- The space complexity will also be `O(N)`, as, in the worst case, we will be pushing `'N'` numbers in the HashTable.

Remove Duplicates (easy)

https://leetcode.com/problems/remove-duplicates-from-sorted-array/

> Given an array of sorted numbers, **remove all duplicates** from it. You should **not use any extra space**; after removing the duplicates in-place return the length of the subarray that has no duplicate in it.

In this problem, we need to remove the duplicates in-place such that the resultant length of the array remains sorted. As the input array is sorted, therefore, one way to do this is to shift the elements left whenever we encounter duplicates. In other words, we will keep one pointer for iterating the array and one pointer for placing the next non-duplicate number. So 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.

- Assume the input is sorted

```
function removeDuplicates(arr) {

  //shift the elements left when we encounter duplicates

  //one pointer for iterating
  let i = 1

  //one pointer for placing this next non-duplicate
  let nextNonDupe = 1

  while(i < arr.length) {
    if(arr[nextNonDupe -1] !== arr[i]) {
      arr[nextNonDupe] = arr[i]
      nextNonDupe++
    }
    i++
  }
  return nextNonDupe
}

removeDuplicates([2, 3, 3, 3, 6, 9, 9])//4, The first four elements after removing the
removeDuplicates([2, 2, 2, 11])//2, The first two elements after removing the duplicat
```

- The time complexity of the above algorithm will be `O(N)`, where `'N'` is the total number of elements in the given array.
- The algorithm runs in constant space `O(1)`.

Remove Element

https://leetcode.com/problems/remove-element/

> Given an unsorted array of numbers and a target 'key', remove all instances of 'key' in-place and return the new length of the array.

```
function removeElement(arr, key) {
  //pointed for index of the next element which is not the key
  let nextElement = 0

  for(i = 0; i < arr.length; i++) {
    if(arr[i] !== key) {
      arr[nextElement] = arr[i]
      nextElement++
    }
  }
  return nextElement
}

removeElement([3, 2, 3, 6, 3, 10, 9, 3], 3)//4, The first four elements after removing
removeElement([2, 11, 2, 2, 1], 2)//2, The first four elements after removing every 'K
```

- The time complexity of the above algorithm will be `O(N)`, where `'N'` is the total number of elements in the given array.
- The algorithm runs in constant space `O(1)`.

Squaring a Sorted Array (easy)

https://leetcode.com/problems/squares-of-a-sorted-array/

This is a straightforward question. The only trick is that we can have negative numbers in the input array, which will make it a bit difficult to generate the output array with squares in sorted order.

An easier approach could be to first find the index of the first non-negative number in the array. After that, we can use **Two Pointers** to iterate the array. One pointer will move forward to iterate the non-negative numbers, and the other pointer will move backward to iterate the negative numbers. At any step, whichever number gives us a bigger square will be added to the output array.

Since the numbers at both ends can give us the largest square, an alternate approach could be to use two pointers starting at both ends of the input array. At any step, whichever pointer gives us the bigger square, we add it to the result array and move to the next/previous number according to the pointer.

```
function makeSquares(arr) {
  //The only trick is that we can have negative numbers in the input array, which will
  //An easier approach could be to first find the index of the first non-negative numb
  //After that, we can use Two Pointers to iterate the array.
  //One pointer will move forward to iterate the non-negative numbers
  //the other pointer will move backward to iterate the negative numbers. At any step,
  //Since the numbers at both ends can give us the largest square, an alternate approa
```

```
    const n = arr.length;
    let squares = Array(n).fill(0)
    let highestSquareIndex = n - 1
    let start = 0
    let end = n-1
    while(start<= end) {
      let startSquare = arr[start] * arr[start]
      let endSquare = arr[end] * arr[end]

      if(startSquare > endSquare) {
        squares[highestSquareIndex]  = startSquare
        start++
      } else {
        squares[highestSquareIndex]  = endSquare
        end--
      }
      highestSquareIndex--
    }
    return squares
  }

  makeSquares([-2, -1, 0, 2, 3])//[0, 1, 4, 4, 9]
  makeSquares([-3, -1, 0, 1, 2])//[0, 1, 1, 4, 9]
```

- The above algorithm's time complexity will be `O(N)` as we are iterating the input array only once.
- The above algorithm's space complexity will also be `O(N)`; this space will be used for the output array.

🌟 Triplet Sum to Zero (medium)

https://leetcode.com/problems/3sum/

> Given an array of unsorted numbers, find all unique triplets in it that add up to zero.

This problem follows the **Two Pointers** pattern and shares similarities with **Pair with Target Sum**. A couple of differences are that the input array is not sorted and instead of a pair we need to find triplets with a target sum of zero.

To follow a similar approach, first, we will sort the array and then iterate through it taking one number at a time. Let's say during our iteration we are at number `'X'`, so we need to find `'Y'` and `'Z'` such that `X + Y + Z == 0`. At this stage, our problem translates into finding a pair whose sum is equal to "`-X`" (as from the above equation `Y + Z == -X`).

Another difference from Pair with Target Sum is that we need to find all the unique triplets. To handle this, we have to skip any duplicate number. Since we will be sorting the array, so all the duplicate numbers will be next to each other and are easier to skip.

```
function searchTriplets(arr) {
  arr.sort((a, b) => a-b)
  const triplets = [];

  for(i = 0; i< arr.length;i++) {

    if(i>0 && arr[i] === arr[i -1]){
      //skip the same element to avoid dupes
      continue
    }
    searchPair(arr, -arr[i], i+1, triplets)
  }
  return triplets;
};

function searchTriplets(arr) {
  arr.sort((a, b) => a -b)
  const triplets = []

  for(i = 0; i < arr.length; i++) {
    if(i > 0 && arr[i] === arr[i -1]){
      //skip same element to avoid duplicates
      continue
    }
    searchPair(arr, -arr[i], i + 1, triplets)
  }
  return triplets
};

function searchPair(arr, targetSum, start, triplets) {
  let end = arr.length -1
  while(start < end) {
    const currentSum = arr[start] + arr[end]
    if(currentSum === targetSum) {
      //found the triplet
      triplets.push([-targetSum, arr[start], arr[end]])
      start++
      end--
      while(start < end && arr[start] === arr[start-1]) {
        //skip same element to avoid duplicates
        start++
      }
      while(start < end && arr[end] === arr[end + 1]) {
        //skip same element to avoid duplicates
        end--
      }
    } else if(targetSum > currentSum) {
      //we need a pair with a bigger sum
      start++
    } else {
      //we need a pair with a smaller sum
      end--
    }
  }
```

```
}

searchTriplets([-3, 0, 1, 2, -1, 1, -2]) //[[-3, 1, 2], [-2, 0, 2], [-2, 1, 1], [-1, 0
searchTriplets([-5, 2, -1, -2, 3]) //[[-5, 2, 3], [-2, -1, 3]]
```

- Sorting the array will take `O(N * logN)`. The `searchPair()` function will take `O(N)`. As we are calling `searchPair()` for every number in the input array, this means that overall `searchTriplets()` will take `O(N * logN + N^2)`, which is asymptotically equivalent to `O(N^2)`.
- Ignoring the space required for the output array, the space complexity of the above algorithm will be `O(N)` which is required for sorting.

Triplet Sum Close to Target (medium)

https://leetcode.com/problems/3sum-closest/

> Given an array of unsorted numbers and a target number, find a **triplet in the array whose sum is as close to the target number as possible**, return the sum of the triplet. If there are more than one such triplet, return the sum of the triplet with the smallest sum.

This problem follows the **Two Pointers** pattern and is quite similar to **Triplet Sum to Zero**.

We can follow a similar approach to iterate through the array, taking one number at a time. At every step, we will save the difference between the triplet and the target number, so that in the end, we can return the triplet with the closest sum.

```
function tripletSumCloseToTarget(arr, targetSum){
  arr.sort((a, b) => a - b)

  let smallestDifference = Infinity

  for(let i = 0; i < arr.length - 2; i++) {
    let start = i + 1
    let end = arr.length - 1

    while(start < end) {
      const targetDifference = targetSum - arr[i] - arr[start] - arr[end]

      if(targetDifference === 0) {
        //we've found a triplet with an exact sum
        //so return the sum of all the numbers
        return targetSum - targetDifference
      }

      if(Math.abs(targetDifference) < Math.abs(smallestDifference)) {
        //save the closet difference
        smallestDifference = targetDifference
      }
      //the second part of the followinf 'if' is to handle the smallest sum
```

```
        //when we have more than one solution
        if(Math.abs(targetDifference) < Math.abs(smallestDifference) || (Math.abs(target
          //save the closest and the biggest difference
          smallestDifference = targetDifference
        }

        if(targetDifference > 0) {
          //we need a triplet with a bigger sum
          start++
        } else {
          //we need a triplet with a smaller sum
          end--
        }
      }
    }
    return targetSum - smallestDifference
  }

  tripletSumCloseToTarget([-2, 0, 1, 2], 2)//1, he triplet [-2, 1, 2] has the closest su
  tripletSumCloseToTarget([-3, -1, 1, 2], 1)//0, The triplet [-3, 1, 2] has the closest
  tripletSumCloseToTarget([1, 0, 1, 1], 100)//3, The triplet [1, 1, 1] has the closest s
  tripletSumCloseToTarget([-1,2,1,-4], 1)//2, The sum that is closest to the target is 2
```

- Sorting the array will take `O(N* logN)`. Overall, the function will take `O(N * logN + N^2)`, which is asymptotically equivalent to `O(N^2)`
- The above algorithm's space complexity will be `O(N)`, which is required for sorting.

Triplets with Smaller Sum (medium)

https://leetcode.com/problems/3sum-smaller/

> Given an array `arr` of unsorted numbers and a `target` sum, count all triplets in it such that `arr[i] + arr[j] + arr[k] < target` where `i`, `j`, and `k` are three different indices. Write a function to return the count of such triplets.

This problem follows the **Two Pointers pattern** and shares similarities with **Triplet Sum to Zero**. The only difference is that, in this problem, we need to find the triplets whose sum is less than the given target. To meet the condition `i != j != k` we need to make sure that each number is not used more than once.

Following a similar approach, first, we can sort the array and then iterate through it, taking one number at a time. Let's say during our iteration we are at number `'X'`, so we need to find `'Y'` and `'Z'` such that `X + Y + Z < target`. At this stage, our problem translates into finding a pair whose sum is less than `"target - X"` (as from the above equation `Y + Z == target - X`). We can use a similar approach as discussed in **Triplet Sum to Zero**.

```
  function tripletWithSmallerSum (arr, target) {
    arr.sort((a, b) => a -b)
```

```
  let count = 0;

  for(let i = 0; i < arr.length - 2; i++){
    count += searchPair(arr, target - arr[i], i)
  }
  return count;
};

function searchPair(arr, targetSum, first){
  let count = 0
  let start = first + 1
  let end = arr.length -1

  while(start < end) {
    if(arr[start] + arr[end] < targetSum) {
      //we found the a triplet
      //since arr[end] >= arr[start], therefore, we can replace arr[end]
      //by any number between start and end to get a sum less than the targetSum
      count += end - start
      start++
    } else {
      //we need a pair with a smaller sum
      end--
    }
  }
  return count

}

tripletWithSmallerSum ([-1, 0, 2, 3], 3)//2, There are two triplets whose sum is less
tripletWithSmallerSum ([-1, 4, 2, 1, 3], 5)//4, There are four triplets whose sum is l
tripletWithSmallerSum ([-2,0,1,3], 2)//2, Because there are two triplets which sums ar
tripletWithSmallerSum ([], 0)//0
tripletWithSmallerSum ([0], 0)//0
```

- Sorting the array will take `O(N * logN)`. The `searchPair()` will take `O(N)`. So, overall `searchTriplets()` will take `O(N * logN + N^2)`, which is asymptotically equivalent to `O(N^2)`.
- The space complexity of the above algorithm will be `O(N)` which is required for sorting if we are not using an in-place sorting algorithm.

> Write a function to return the list of all such triplets instead of the count. How will the time complexity change in this case?

```
function tripletWithSmallerSum (arr, target) {
  arr.sort((a, b) => a -b)
  const triplets = []

  for(let i = 0; i < arr.length - 2; i++){
    searchPair(arr, target - arr[i], i, triplets)
  }
```

```
    return triplets;
  };

  function searchPair(arr, targetSum, first, triplets){

    let start = first + 1
    let end = arr.length -1

    while(start < end) {
      if(arr[start] + arr[end] < targetSum) {
        //we found the a triplet
        //since arr[end] >= arr[start], therefore, we can replace arr[end]
        //by any number between start and end to get a sum less than the targetSum
        for(let i = end; i > start; i--){
          triplets.push(arr[first], arr[start], arr[end])
        }
        start++
      } else {
        //we need a pair with a smaller sum
        end--
      }
    }
  }
```

- Sorting the array will take `O(N * logN)`. The `searchPair()`, in this case, will take `O(N^2)`; the main while loop will run in `O(N)` but the nested for loop can also take `O(N)` - this will happen when the target sum is bigger than every triplet in the array. So, overall `searchTriplets()` will take `O(N * logN + N^3)`, which is asymptotically equivalent to `O(N^3)`.
- Ignoring the space required for the output array, the space complexity of the above algorithm will be `O(N)` which is required for sorting.

🌟 Subarrays with Product Less than a Target (medium)

https://leetcode.com/problems/subarray-product-less-than-k/

> Given an array with positive numbers and a target number, find all of its contiguous subarrays whose **product is less than the target number**.

This problem follows the **Sliding Window** and the **Two Pointers** pattern and shares similarities with **Triplets with Smaller Sum** with two differences:

1. In this problem, the input array is not sorted.
2. Instead of finding triplets with sum less than a target, we need to find all subarrays having a product less than the target. The implementation will be quite similar to **Triplets with Smaller Sum**.

```
function findSubarrays(arr, target) {
  let result = []
  let product = 1
  let start = 0

  for(let end = 0; end < arr.length; end++) {
    product *= arr[end]
    while(product >= target && start < arr.length) {
      product /= arr[start]
      start++
    }
    //since the product of all numbers from start to end is less than the target.
    //Therefore, all subarrays from start to end will have a product less than the tar
    //to avoid duplicates, we will start with a subarray containing only arr[end] and
    const tempList = []
    for(let i = end; i > start -1; i--) {
      tempList.unshift(arr[i])
      result.push(tempList)
    }
  }
  return result
}

findSubarrays([2, 5, 3, 10], 30)//[2], [5], [2, 5], [3], [5, 3], [10] There are six co
findSubarrays([8, 2, 6, 5], 50)//[8], [2], [8, 2], [6], [2, 6], [5], [6, 5] There are
findSubarrays([10, 5, 2, 6], 100)//The 8 subarrays that have product less than 100 are
```

- The main for-loop managing the sliding window takes `O(N)` but creating subarrays can take up to `O(N^2)` in the worst case. Therefore overall, our algorithm will take `O(N^3)` .
- Ignoring the space required for the output list, the algorithm runs in `O(N)` space which is used for the temp list.

Dutch National Flag Problem (medium)

https://leetcode.com/problems/sort-colors/

> Given an array containing 0s, 1s and 2s, sort the array in-place. You should treat numbers of the array as objects, hence, we can't count 0s, 1s, and 2s to recreate the array.

The flag of the Netherlands consists of three colors: red, white and blue; and since our input array also consists of three different numbers that is why it is called **Dutch National Flag problem**.

The brute force solution will be to use an in-place sorting algorithm like *Heapsort* which will take `O(N*logN)` . Can we do better than this? Is it possible to sort the array in one iteration?

We can use a **Two Pointers** approach while iterating through the array. Let's say the two pointers are called `low` and `high` which are pointing to the first and the last element of the array

respectively. So while iterating, we will move all `0` s before `low` and all `2s` after `high` so that in the end, all `1s` will be between `low` and `high` .

```
function dutchFlagSort(arr) {
  //all elements < low are 0, and all elements > high are 2
  //all elements >= low < i are 1
  let low = 0
  let high = arr.length - 1
  let i = 0

  while(i <= high) {
    if(arr[i] === 0) {
      //swap
      //[arr[i], arr[low]] = [arr[low], arr[i]]
      let temp = arr[i]
      arr[i] = arr[low]
      arr[low] = temp
      //increment i and low
      i++
      low++
    } else if(arr[i] === 1){
      i++
    } else{
      //the case for arr[i] === 2
      //swap
      // [arr[i], arr[high]] = [arr[high], arr[i]]
      temp = arr[i]
      arr[i] = arr[high]
      arr[high] = temp
      //decrement high only, after the swap the number
      //at index i could be 0, 1, or 2
      high--
    }
  }
  return arr
}

console.log(dutchFlagSort([1, 0, 2, 1, 0]))//[0 0 1 1 2]
console.log(dutchFlagSort([2, 2, 0, 1, 2, 0]))//[0 0 1 2 2 2 ]
```

- The time complexity of the above algorithm will be `O(N)` as we are iterating the input array only once.
- The algorithm runs in constant space `O(1)` .

🌟 Quadruple Sum to Target (medium)

https://leetcode.com/problems/4sum/

> Given an array of unsorted numbers and a target number, find all **unique quadruplets** in it, whose **sum is equal to the target number**.

This problem follows the **Two Pointers** pattern and shares similarities with **Triplet Sum to Zero**.

We can follow a similar approach to iterate through the array, taking one number at a time. At every step during the iteration, we will search for the quadruplets similar to **Triplet Sum to Zero** whose sum is equal to the given target.

```
function searchQuads (arr, target) {
  //sort the array
  arr.sort((a,b) => a -b)

  let quads = []

  for(let i = 0; i < arr.length-3; i++) {
    //skip the same element to avoid duplicate quadruplets
    if(i > 0 && arr[i] === arr[i-1]) {
      continue
    }
    for(let j = i +1; j < arr.length-2; j++) {
      //skip the same element to avoid duplicate quadruplets
      if(j > i + 1 && arr[j] === arr[j-1]){
        continue
      }
      searchPairs(arr, target, i, j, quads)
    }
  }
  return quads
}

function searchPairs(arr, targetSum, first, second, quads) {
  let start = second + 1
  let end = arr.length -1

  while(start < end) {
    const sum = arr[first] + arr[second] + arr[start] + arr[end]
    if(sum === targetSum) {
      //found the quadruplet
      quads.push([arr[first], arr[second], arr[start], arr[end]])
      start++
      end--
      while(start < end && arr[start] === arr[start -1]){
        //skip the same element to avoid duplicate quadruplets
        start++
      }
      while(start < end && arr[end] === arr[end -1]){
        //skip the same element to avoid duplicate quadruplets
        end--
      }
    } else if(sum < targetSum) {
      //we need a pair with a bigger sum
      start++
    } else {
      //we need a pair with a smaller sum
      end--
```

```
      }
    }
  }

  searchQuads([4,1,2,-1,1,-3], 1)//-3, -1, 1, 4], [-3, 1, 1, 2]
  searchQuads([2,0,-1,1,-2,2], 2)//[-2, 0, 2, 2], [-1, 0, 1, 2]
```

- Sorting the array will take `O(N*logN)`. Overall `searchQuads()` will take `O(N * logN + N³)`, which is asymptotically equivalent to `O(N³)`.
- The space complexity of the above algorithm will be `O(N)` which is required for sorting.

🌟 Comparing Strings containing Backspaces (medium)

https://leetcode.com/problems/backspace-string-compare/

> Given two strings containing backspaces (identified by the character `'#'`), check if the two strings are equal.

To compare the given strings, first, we need to apply the backspaces. An efficient way to do this would be from the end of both the strings. We can have separate pointers, pointing to the last element of the given strings. We can start comparing the characters pointed out by both the pointers to see if the strings are equal. If, at any stage, the character pointed out by any of the pointers is a backspace (`'#'`), we will skip and apply the backspace until we have a valid character available for comparison.

```
function backspaceCompare(str1, str2) {
  //use two pointer approach to compare the strings
  let pointerOne = str1.length -1
  let pointerTwo = str2.length -1

  while(pointerOne >= 0 || pointerTwo >= 0){
    let i = getNextChar(str1, pointerOne)
    let j = getNextChar(str2, pointerTwo)

    if(i < 0 && j < 0){
      //reached the end of both strings
      return true
    }
     if(i < 0 || j < 0){
      //reached the end of both strings
      return false
    }
    if(str1[i] !== str2[j]){
      //check if the characters are equal
      return false
    }
    pointerOne = i - 1
    pointerTwo = j -1
  }
```

```
    return true
  }


  function getNextChar(str, index) {
    let backspaceCount = 0
    while(index >= 0) {
      if(str[index] === '#'){
        //found a backspace character
        backspaceCount++
      } else if(backspaceCount > 0) {
        //a non-backspace character
        backspaceCount--
      } else {
        break
      }
      //skip a backspace or valid character
      index--
    }
    return index
  }

  backspaceCompare("xy#z", "xzz#")//true, After applying backspaces the strings become "
  backspaceCompare("xy#z", "xyz#")//false, After applying backspaces the strings become
  backspaceCompare("xp#", "xyz##")//true, After applying backspaces the strings become "
  backspaceCompare("xywrrmp", "xywrrmu#p")//true, After applying backspaces the strings
```

- The time complexity of the above algorithm will be `O(M+N)` where `'M'` and `'N'` are the lengths of the two input strings respectively.
- The algorithm runs in constant space `O(1)`.

🌟 Minimum Window Sort (medium)

https://leetcode.com/problems/shortest-subarray-to-be-removed-to-make-array-sorted/

> Given an array, find the length of the smallest subarray in it which when sorted will sort the whole array.

As we know, once an array is sorted (in ascending order), the smallest number is at the beginning and the largest number is at the end of the array. So if we start from the beginning of the array to find the first element which is out of sorting order i.e., which is smaller than its previous element, and similarly from the end of array to find the first element which is bigger than its previous element, will sorting the subarray between these two numbers result in the whole array being sorted?

Let's try to understand this with Example-2 mentioned above. In the following array, what are the first numbers out of sorting order from the beginning and the end of the array:

```
  [1, 3, 2, 0, -1, 7, 10]
```

Starting from the beginning of the array the first number out of the sorting order is `'2'` as it is smaller than its previous element which is `'3'`. Starting from the end of the array the first number out of the sorting order is `'0'` as it is bigger than its previous element which is `'-1'`

As you can see, sorting the numbers between `'3'` and `'-1'` will not sort the whole array. To see this, the following will be our original array after the sorted subarray:

```
[1, -1, 0, 2, 3, 7, 10]
```

The problem here is that the smallest number of our subarray is `'-1'` which dictates that we need to include more numbers from the beginning of the array to make the whole array sorted. We will have a similar problem if the maximum of the subarray is bigger than some elements at the end of the array. To sort the whole array we need to include all such elements that are smaller than the biggest element of the subarray. So our final algorithm will look like:

1. From the beginning and end of the array, find the first elements that are out of the sorting order. The two elements will be our candidate subarray.
2. Find the maximum and minimum of this subarray.
3. Extend the subarray from beginning to include any number which is bigger than the minimum of the subarray.
4. Similarly, extend the subarray from the end to include any number which is smaller than the maximum of the subarray.

```
function shortestWindowSort(arr) {
  let low = 0
  let high = arr.length - 1

  //find the first number out of sorting order from the beginning
  while(low < arr.length -1 && arr[low] <= arr[low+1]){
    low++
  }
  if(low === arr.length - 1) {
    // if the array is already sorted
    return 0
  }
  //find the first number out of sorting order from the end
  while(high > 0 && arr[high] >= arr[high - 1]){
    high--
  }

  //find the max and min of the subarray
  let subArrayMax = -Infinity
  let subArrayMin = Infinity

  for(let k = low; k < high + 1; k++) {
    subArrayMax = Math.max(subArrayMax, arr[k])
```

```
      subArrayMin = Math.min(subArrayMin, arr[k])
    }

    //extend the subarray to include any number which is bigger than the minumum of the
    while(low > 0 && arr[low -1] > subArrayMin) {
      low--
    }
    //extend the subarray to include any number which is small than the maximum of the s
    while(high < arr.length - 1 && arr[high + 1] < subArrayMax){
      high++
    }

    return high - low + 1
  }

  shortestWindowSort([1,2,5,3,7,10,9,12])//5, We need to sort only the subarray [5, 3, 7
  shortestWindowSort([1,3,2,0,-1,7,10])//5, We need to sort only the subarray [1, 3, 2,
  shortestWindowSort([1,2,3])//0, The array is already sorted
  shortestWindowSort([3,2,1])// 3, The whole array needs to be sorted.
```

- The time complexity of the above algorithm will be `O(N)`.
- The algorithm runs in constant space `O(1)`.

Pattern 3: Fast & Slow pointers

The **Fast & Slow** pointer approach, also known as the **Hare & Tortoise algorithm**, is a pointer algorithm that uses two pointers which move through the array (or sequence/LinkedList) at different speeds. This approach is quite useful when dealing with cyclic LinkedLists or arrays.

By moving at different speeds (say, in a cyclic LinkedList), the algorithm proves that the two pointers are bound to meet. The fast pointer should catch the slow pointer once both the pointers are in a cyclic loop.

One of the famous problems solved using this technique was **Finding a cycle in a LinkedList**. Let's jump onto this problem to understand the **Fast & Slow** pattern.

LinkedList Cycle (easy)

https://leetcode.com/problems/linked-list-cycle/

> Given the head of a **Singly LinkedList**, write a function to determine if the LinkedList has a cycle in it or not.

Imagine two racers running in a circular racing track. If one racer is faster than the other, the faster racer is bound to catch up and cross the slower racer from behind. We can use this fact to devise an algorithm to determine if a LinkedList has a cycle in it or not.

Imagine we have a slow and a fast pointer to traverse the LinkedList. In each iteration, the slow pointer moves one step and the fast pointer moves two steps. This gives us two conclusions:

1. If the LinkedList doesn't have a cycle in it, the fast pointer will reach the end of the LinkedList before the slow pointer to reveal that there is no cycle in the LinkedList.
2. The slow pointer will never be able to catch up to the fast pointer if there is no cycle in the LinkedList.

If the LinkedList has a cycle, the fast pointer enters the cycle first, followed by the slow pointer. After this, both pointers will keep moving in the cycle infinitely. If at any stage both of these pointers meet, we can conclude that the LinkedList has a cycle in it. Let's analyze if it is possible for the two pointers to meet. When the fast pointer is approaching the slow pointer from behind we have two possibilities:

1. The fast pointer is one step behind the slow pointer.
2. The fast pointer is two steps behind the slow pointer.

All other distances between the fast and slow pointers will reduce to one of these two possibilities. Let's analyze these scenarios, considering the fast pointer always moves first:

1. If the fast pointer is one step behind the slow pointer: The fast pointer moves two steps and the slow pointer moves one step, and they both meet.
2. If the fast pointer is two steps behind the slow pointer: The fast pointer moves two steps and the slow pointer moves one step. After the moves, the fast pointer will be one step behind the slow pointer, which reduces this scenario to the first scenario. This means that the two pointers will meet in the next iteration.

This concludes that the two pointers will definitely meet if the LinkedList has a cycle.

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

function hasCycle(head) {
  let slow = head
  let fast = head
  while(fast !== null && fast.next !== null) {
    fast = fast.next.next;
    slow = slow.next

    if(slow === fast) {
      //found the cycle
      return true
    }
  }
  return false
}

head = new Node(1)
head.next = new Node(2)
head.next.next = new Node(3)
head.next.next.next = new Node(4)
head.next.next.next.next = new Node(5)
head.next.next.next.next.next = new Node(6)
console.log(`LinkedList has cycle: ${hasCycle(head)}`)

head.next.next.next.next.next.next = head.next.next
console.log(`LinkedList has cycle: ${hasCycle(head)}`)

head.next.next.next.next.next.next = head.next.next.next
console.log(`LinkedList has cycle: ${hasCycle(head)}`)
```

- Once the slow pointer enters the cycle, the fast pointer will meet the slow pointer in the same loop. Therefore, the time complexity of our algorithm will be `O(N)` where `'N'` is the total number of nodes in the LinkedList.
- The algorithm runs in constant space `O(1)`.

> Given the head of a LinkedList with a cycle, find the length of the cycle.

Once the fast and slow pointers meet, we can save the slow pointer and iterate the whole cycle with another pointer until we see the slow pointer again to find the length of the cycle.

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

function findCycleLength(head) {
  let slow = head
  let fast = head

  while(fast !== null && fast.next !== null) {
    fast = fast.next.next;
    slow = slow.next

    if(slow === fast) {
      //found the cycle
      return calculateCycleLength(slow)
    }
  }
  return 0
}

function calculateCycleLength(slow) {
  let current = slow
  let cycleLength = 0

  while(true) {
    current = current.next
    cycleLength++
    if(current === slow) {
      break
    }
  }
  return cycleLength
}

head = new Node(1)
head.next = new Node(2)
head.next.next = new Node(3)
head.next.next.next = new Node(4)
head.next.next.next.next = new Node(5)
head.next.next.next.next.next = new Node(6)
console.log(`LinkedList has cycle length of: ${findCycleLength(head)}`)

head.next.next.next.next.next.next = head.next.next
console.log(`LinkedList has cycle length of: ${findCycleLength(head)}`)
```

```
head.next.next.next.next.next.next = head.next.next.next
console.log(`LinkedList has cycle length of: ${findCycleLength(head)}`)
```

- The above algorithm runs in `O(N)` time complexity and `O(1)` space complexity.

Start of LinkedList Cycle (medium)

https://leetcode.com/problems/linked-list-cycle-ii/

> Given the head of a **Singly LinkedList** that contains a cycle, write a function to find the **starting node of the cycle**.

If we know the length of the **LinkedList** cycle, we can find the start of the cycle through the following steps:

1. Take two pointers. Let's call them `pointer1` and `pointer2`.
2. Initialize both pointers to point to the start of the LinkedList.
3. We can find the length of the LinkedList cycle using the approach discussed in **LinkedList Cycle**. Let's assume that the length of the cycle is 'K' nodes.
4. Move `pointer2` ahead by 'K' nodes.
5. Now, keep incrementing `pointer1` and `pointer2` until they both meet.
6. As `pointer2` is 'K' nodes ahead of `pointer1`, which means, `pointer2` must have completed one loop in the cycle when both pointers meet. Their meeting point will be the start of the cycle.

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

function findCycleStart(head) {
  let cycleLength = 0
  let slow = head
  let fast = head
   while((fast !== null && fast.next !== null)){
     fast = fast.next.next
     slow = slow.next

     if(slow === fast) {
       //found the cycle
       cycleLength = calculateCycleLength(slow)
       break
     }
   }

   return findStart(head, cycleLength)
```

```
};


function calculateCycleLength(slow) {
  let current = slow
  let cycleLength = 0

  while(true) {
    current = current.next
    cycleLength++
    if(current === slow) {
      break
    }
  }
  return cycleLength
}

function findStart(head, cycleLength) {
  let pointer1 = head
  let pointer2 = head
  //move pointer2 ahead by cycleLength nodes
  while(cycleLength > 0) {
    pointer2 = pointer2.next
    cycleLength--
  }

  //increment both pointers until they meet at the start
  //of the cycle
  while(pointer1 !== pointer2) {
    pointer1 = pointer1.next
    pointer2 = pointer2.next
  }
  return pointer1
}

head = new Node(1)
head.next = new Node(2)
head.next.next = new Node(3)
head.next.next.next = new Node(4)
head.next.next.next.next = new Node(5)
head.next.next.next.next.next = new Node(6)

head.next.next.next.next.next.next = head.next.next
console.log(`LinkedList cycle start: ${findCycleStart(head).value}`)

head.next.next.next.next.next.next = head.next.next.next
console.log(`LinkedList cycle start: ${findCycleStart(head).value}`)

head.next.next.next.next.next.next = head
console.log(`LinkedList cycle start: ${findCycleStart(head).value}`)
```

- As we know, finding the cycle in a LinkedList with `'N'` nodes and also finding the length of the cycle requires `O(N)` . Also, as we saw in the above algorithm, we will need `O(N)` to find the

start of the cycle. Therefore, the overall time complexity of our algorithm will be `O(N)`.

- The algorithm runs in constant space `O(1)`.

Happy Number (medium)

https://leetcode.com/problems/happy-number/

Any number will be called a happy number if, after repeatedly replacing it with a number equal to the **sum of the square of all of its digits, leads us to number '1'**. All other (not-happy) numbers will never reach '1'. Instead, they will be stuck in a cycle of numbers which does not include '1'.

The process, defined above, to find out if a number is a happy number or not, always ends in a cycle. If the number is a happy number, the process will be stuck in a cycle on number '1,' and if the number is not a happy number then the process will be stuck in a cycle with a set of numbers. As we saw in Example-2 while determining if '12' is a happy number or not, our process will get stuck in a cycle with the following numbers: 89 -> 145 -> 42 -> 20 -> 4 -> 16 -> 37 -> 58 -> 89

We saw in the **LinkedList Cycle** problem that we can use the **Fast & Slow** pointers method to find a cycle among a set of elements. As we have described above, each number will definitely have a cycle. Therefore, we will use the same fast & slow pointer strategy to find the cycle and once the cycle is found, we will see if the cycle is stuck on number '1' to find out if the number is happy or not.

```
function findHappyNumber(num) {
  let slow = num
  let fast = num

  while(true) {
    //move one step
    slow = findSquareSum(slow)
    //move two steps
    fast = findSquareSum(findSquareSum(fast))

    if(slow === fast) {
      //found the cycle
      break
    }
  }
  //see if the cycle is stuck on the number 1
  return slow === 1
}

function findSquareSum(num) {
  let sum = 0
  while(num > 0) {
    let digit = num % 10
    sum += digit * digit
    num = Math.floor(num / 10)
  }
```

```
    return sum

  }


findHappyNumber(23)//true
```

`23` is a happy number, Here are the steps to find out that `23` is a happy number:

1. $2^2 + 3^2 = 4 + 9 = 13$
2. $1^2 + 3^2 = 1 + 9 = 10$
3. $1^2 + 0^2 = 1 + 0 = 1$

```
findHappyNumber(12)//false
```

`12` is not a happy number, Here are the steps to find out that `12` is not a happy number:

1. $1^2+2^2= 1 + 4 = 5$
2. $5^2 = 25$
3. $2^2 + 5^2 = 4 + 25 = 29$
4. $2^2 + 9^2 = 4 + 81 = 85$
5. $8^2 + 5^2 = 64 + 25 = 89$
6. $8^2 + 9^2 = 64 + 81 = 145$
7. $1^2 + 4^2 + 5^2= 1 + 16 + 25 = 42$
8. $4^2 + 2^2 = 16 + 4 = 20$
9. $2^2 + 0^2 = 4 + 0 = 4$
10. $4^2= 16$
11. $1^2 + 6^2 = 1 + 36 = 37$
12. $3^2 + 7^2 = 9 + 49 = 58$
13. $5^2 + 8^2= 25 + 64 = 89$ Step ‘13’ leads us back to step ‘5’ as the number becomes equal to ‘89’, this means that we can never reach ‘1’, therefore, ‘12’ is not a happy number.

```
findHappyNumber(19)//true
```

`19` is a happy number, Here are the steps to find out that 19 is a happy number:

1. $1^2 + 9^2 = 82$
2. $8^2 + 2^2 = 68$
3. $6^2 + 8^2 = 100$
4. $1^2 + 0^2 + 0^2 = 1$

```
findHappyNumber(2)//false
```

`2` is not a happy number

- The time complexity of the algorithm is difficult to determine. However we know the fact that all unhappy numbers eventually get stuck in the cycle: 4 -> 16 -> 37 -> 58 -> 89 -> 145 -> 42 -> 20 -> 4

This sequence behavior tells us two things:

1. If the number `N` is less than or equal to 1000, then we reach the cycle or '1' in at most 1001 steps.
2. For `N > 1000`, suppose the number has `'M'` digits and the next number is `'N1'`. From the above Wikipedia link, we know that the sum of the squares of the digits of `'N'` is at most 9^2M, or `81M` (this will happen when all digits of `'N'` are `'9'`).

This means:

1. `N1 < 81M`
2. As we know `M = log(N+1)`
3. Therefore: `N1 < 81 * log(N+1) => N1 = O(logN)`

- This concludes that the above algorithm will have a time complexity of `O(logN)`.
- The algorithm runs in constant space `O(1)`.

Middle of the LinkedList (easy)

https://leetcode.com/problems/middle-of-the-linked-list/

> Given the head of a **Singly LinkedList**, write a method to return the **middle node** of the LinkedList.
>
> If the total number of nodes in the LinkedList is even, return the second middle node.

One brute force strategy could be to first count the number of nodes in the LinkedList and then find the middle node in the second iteration. Can we do this in one iteration?

We can use the **Fast & Slow** pointers method such that the fast pointer is always twice the nodes ahead of the slow pointer. This way, when the fast pointer reaches the end of the LinkedList, the slow pointer will be pointing at the middle node.

```
class Node {
  constructor(value, next = null) {
    this.value = value
    this.next = next
  }
}

function findMiddleOfLinkedList(head) {
  let slow = head
  let fast = head
```

```
  while(fast !== null && fast.next !== null) {
    slow = slow.next
    fast = fast.next.next
  }
  return slow
}

head = new Node(1)
head.next = new Node(2)
head.next.next = new Node(3)
head.next.next.next = new Node(4)
head.next.next.next.next = new Node(5)

console.log(`Middle Node: ${findMiddleOfLinkedList(head).value}`)

head.next.next.next.next.next = new Node(6)
console.log(`Middle Node: ${findMiddleOfLinkedList(head).value}`)

head.next.next.next.next.next.next = new Node(7)
console.log(`Middle Node: ${findMiddleOfLinkedList(head).value}`)
```

- The above algorithm will have a time complexity of `O(N)` where `'N'` is the number of nodes in the LinkedList.
- The algorithm runs in constant space `O(1)`.

🌟 Palindrome LinkedList (medium)

https://leetcode.com/problems/palindrome-linked-list/

> Given the head of a **Singly LinkedList**, write a method to check if the **LinkedList is a palindrome** or not.
>
> Your algorithm should use **constant space** and the input LinkedList should be in the original form once the algorithm is finished. The algorithm should have `O(N)` time complexity where `'N'` is the number of nodes in the LinkedList.

Example 1:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> null
Output: true
```

Example 2:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> 2 -> null
Output: false
```

As we know, a palindrome LinkedList will have nodes values that read the same backward or forward. This means that if we divide the LinkedList into two halves, the node values of the first half in the forward direction should be similar to the node values of the second half in the backward direction. As we have been given a Singly LinkedList, we can't move in the backward direction. To handle this, we will perform the following steps:

1. We can use the **Fast & Slow pointers** method similar to **Middle of the LinkedList** to find the middle node of the LinkedList.
2. Once we have the middle of the LinkedList, we will reverse the second half.
3. Then, we will compare the first half with the reversed second half to see if the LinkedList represents a palindrome.
4. Finally, we will reverse the second half of the LinkedList again to revert and bring the LinkedList back to its original form.

```
class Node {
  constructor(value, next = null) {
    this.value = value
    this.next = next
  }
}

function isPalindromicLinkedList(head) {
  if(head === null || head.next === null) {
    return true
  }

  //find the middle of the LinkedList
  let slow = head
  let fast = head

  while((fast !== null && fast.next !== null)) {
    slow = slow.next
    fast = fast.next.next
  }

  //reverse the second half
  let headSecondHalf = reverse(slow)

  //store the head of reversed part to revert back later
  let copyHeadSecondHalf = headSecondHalf

  //compare first and second half
  while((head !== null && headSecondHalf !== null)){
    if(head.value !== headSecondHalf.value) {
      //not a palindrome
      break
    }

    head = head.next
    headSecondHalf = headSecondHalf.next
```

```
    }

    //revert the reverse of the second half
    reverse(copyHeadSecondHalf)

    //if both halves match
    if(head === null || headSecondHalf === null) {
      return true
    }

    return false
  }


  function reverse(head) {
    let prev = null

    while (head !== null) {
      let next = head.next
      head.next = prev
      prev = head
      head = next
    }
    return prev
  }
  head = new Node(2)
  head.next = new Node(4)
  head.next.next = new Node(6)
  head.next.next.next = new Node(4)
  head.next.next.next.next = new Node(2)

  console.log(`Is palindrome: ${isPalindromicLinkedList(head)}`)

  head.next.next.next.next.next = new Node(2)
  console.log(`Is palindrome: ${isPalindromicLinkedList(head)}`)
```

- The above algorithm will have a time complexity of `O(N)` where `'N'` is the number of nodes in the LinkedList.
- The algorithm runs in constant space `O(1)`.

🌟 Rearrange a LinkedList (medium)

https://leetcode.com/problems/reorder-list/

> Given the head of a Singly LinkedList, write a method to modify the LinkedList such that the **nodes from the second half of the LinkedList are inserted alternately to the nodes from the first half in reverse order**. So if the LinkedList has nodes 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> null, your method should return 1 -> 6 -> 2 -> 5 -> 3 -> 4 -> null.

> Your algorithm should not use any extra space and the input LinkedList should be modified in-place.

Example 1:

```
Input: 2 -> 4 -> 6 -> 8 -> 10 -> 12 -> null
Output: 2 -> 12 -> 4 -> 10 -> 6 -> 8 -> null
```

Example 2:

```
Input: 2 -> 4 -> 6 -> 8 -> 10 -> null
Output: 2 -> 10 -> 4 -> 8 -> 6 -> null
```

This problem shares similarities with **Palindrome LinkedList**. To rearrange the given LinkedList we will follow the following steps:

1. We can use the **Fast & Slow pointers** method similar to **Middle of the LinkedList** to find the middle node of the LinkedList.
2. Once we have the middle of the LinkedList, we will reverse the second half of the LinkedList.
3. Finally, we'll iterate through the first half and the reversed second half to produce a LinkedList in the required order.

```
class Node {
  constructor (val, next = null) {
    this.val = val
    this.next = next
  }

  printList() {
    let result = "";
    let temp = this;
    while (temp !== null) {
      result += temp.val + " ";
      temp = temp.next;
    }
    console.log(result);
  }
}
```

```
function reorder (head) {
  if(head === null || head.next === null) {
    return true
  }

  //find the middle of the LinkedList
  let slow = head
```

```
    let fast = head

    while (fast !== null && fast.next !== null) {
      slow = slow.next
      fast = fast.next.next
    }

    //slow is now pointing to the middle node
    headSecondHalf = reverse(slow)
    //reverse thesecond half
    headFirstHalf = head

    //rearrange to produce the LinkList in the required order
    while(headFirstHalf !== null && headSecondHalf !== null) {
      let temp = headFirstHalf.next
      headFirstHalf.next = headSecondHalf
      headFirstHalf = temp

      temp = headSecondHalf.next
      headSecondHalf.next = headFirstHalf
      headSecondHalf = temp
    }

    //set the next of the last node to 'null'
    if(headFirstHalf!== null) {
      headFirstHalf.next = null
    }
  }


  function reverse(head) {
    let prev = null

    while(head !== null) {
      let next = head.next
      head.next = prev
      prev = head
      head = next
    }

    return prev

  }
  head = new Node(2)
  head.next = new Node(4)
  head.next.next = new Node(6)
  head.next.next.next = new Node(8)
  head.next.next.next.next = new Node(10)
  head.next.next.next.next.next = new Node(12)
  reorder(head)
  head.printList()
```

- The above algorithm will have a time complexity of `O(N)` where `'N'` is the number of nodes in the LinkedList.
- The algorithm runs in constant space `O(1)`.

🌟 Cycle in a Circular Array (hard)

https://leetcode.com/problems/circular-array-loop/

We are given an array containing positive and negative numbers. Suppose the array contains a number 'M' at a particular index. Now, if 'M' is positive we will move forward 'M' indices and if 'M' is negative move backwards 'M' indices. You should assume that the **array is circular** which means two things:

1. If, while moving forward, we reach the end of the array, we will jump to the first element to continue the movement.
2. If, while moving backward, we reach the beginning of the array, we will jump to the last element to continue the movement. Write a method to determine **if the array has a cycle**. The cycle should have more than one element and should follow one direction which means the cycle should not contain both forward and backward movements.

Example 1:

```
Input: [1, 2, -1, 2, 2]
Output: true
Explanation: The array has a cycle among indices: 0 -> 1 -> 3 -> 0
```

Example 2:

```
Input: [2, 2, -1, 2]
Output: true
Explanation: The array has a cycle among indices: 1 -> 3 -> 1
```

Example 3:

```
Input: [2, 1, -1, -2]
Output: false
Explanation: The array does not have any cycle.
```

This problem involves finding a cycle in the array and, as we know, the **Fast & Slow pointer** method is an efficient way to do that. We can start from each index of the array to find the cycle. If a number does not have a cycle we will move forward to the next element. There are a couple of additional things we need to take care of:

1. As mentioned in the problem, the cycle should have more than one element. This means that when we move a pointer forward, if the pointer points to the same element after the move, we have a one-element cycle. Therefore, we can finish our cycle search for the current element.

2. The other requirement mentioned in the problem is that the cycle should not contain both forward and backward movements. We will handle this by remembering the direction of each element while searching for the cycle. If the number is positive, the direction will be forward and if the number is negative, the direction will be backward. So whenever we move a pointer forward, if there is a change in the direction, we will finish our cycle search right there for the current element.

```
function circularArrayLoopExists(arr) {
  for(let i = 0; i < arr.length; i++) {
    //if we are moving forward or not
    let isForward = arr[i] >= 0
    let slow = i
    let fast = i

    //if slow or fast becomes -1 this means we can't find cycle for this number
    while(true) {
      // move one step for slow pointer
      slow = findNextIndex(arr, isForward, slow)
      //move one step for fast pointer
      fast = findNextIndex(arr, isForward, fast)
      if(fast !== -1){
        //move another step for the fast pointer
        fast = findNextIndex(arr, isForward, fast)
      }
      if(slow === -1 || fast === -1 || slow === fast){
        break
      }
    }

    if(slow !== -1 && slow === fast){
      return true
    }
  }
  return false
}

function findNextIndex(arr, isForward, currentIndex) {
  let direction = arr[currentIndex] >= 0

  if(isForward !== direction){
    //change indirection, return -1
    return -1
  }

  nextIndex = (currentIndex + arr[currentIndex]) % arr.length
  if(nextIndex < 0) {
    //wrap around for negative numbers
    nextIndex += arr.length
```

```
    }

    //one element cycle, return -1
    if(nextIndex === currentIndex){
      nextIndex = -1
    }

    return nextIndex
  }

  circularArrayLoopExists([1, 2, -1, 2, 2])
  circularArrayLoopExists([2, 2, -1, 2])
  circularArrayLoopExists([2, 1, -1, -2])
```

- The above algorithm will have a time complexity of `O(N²)` where `'N'` is the number of elements in the array. This complexity is due to the fact that we are iterating all elements of the array and trying to find a cycle for each element.
- The algorithm runs in constant space `O(1)`.

An Alternate Approach

In our algorithm, we don't keep a record of all the numbers that have been evaluated for cycles. We know that all such numbers will not produce a cycle for any other instance as well. If we can remember all the numbers that have been visited, our algorithm will improve to `O(N)` as, then, each number will be evaluated for cycles only once. We can keep track of this by creating a separate array, however, in this case, the space complexity of our algorithm will increase to `O(N)`.

Pattern 4 : Merge Intervals

This pattern describes an efficient technique to deal with overlapping intervals. In a lot of problems involving intervals, we either need to find overlapping intervals or merge intervals if they overlap.

Given two intervals (`a` and `b`), there will be six different ways the two intervals can relate to each other:

1. `a` and `b` do not overlap
2. `a` and `b` overlap, `b` ends after `a`
3. `a` completely overlaps `b`
4. `a` and `b` overlap, `a` ends after `b`
5. `b` completly overlaps `a`
6. `a` and `b` do not overlap

Understanding the above six cases will help us in solving all intervals related problems.

Merge Intervals (medium)

https://leetcode.com/problems/merge-intervals/

> Given a list of intervals, **merge all the overlapping intervals** to produce a list that has only mutually exclusive intervals.

Our goal is to merge the intervals whenever they overlap. The diagram above clearly shows a merging approach. Our algorithm will look like this:

1. Sort the intervals on the start time to ensure `a.start <= b.start`
2. If `a` overlaps `b` (i.e. `b.start <= a.end`), we need to merge them into a new interval `c` such that:

```
c.start = a.start
c.end = max(a.end, b.end)
```

We will keep repeating the above two steps to merge `c` with the next interval if it overlaps with `c` .

```
class Interval {
  constructor(start, end) {
    this.start = start;
    this.end = end;
  }

  get_interval() {
```

```
    return "[" + this.start + ", " + this.end + "]";
  }
}

function merge (intervals) {
  if(intervals.length < 2) {
    return intervals
  }

  //sort the intervals on the start time
  intervals.sort((a,b) => a.start - b.start)
  const mergedIntervals = []

  let start = intervals[0].start
  let end = intervals[0].end

  for(let i = 1; i < intervals.length; i++) {
    const interval = intervals[i]
    if(interval.start <= end) {
      //overlapping intervals, adjust the end
      end = Math.max(interval.end, end)
    } else {
      //non-overlapping intercal, add the precious interval and reset
      mergedIntervals.push(new Interval(start, end))
      start = interval.start
      end = interval.end
    }
  }
  //add the last interval
  mergedIntervals.push(new Interval(start, end))
  return mergedIntervals;
};

merged_intervals = merge([new Interval(1, 4), new Interval(2, 5), new Interval(7, 9)])
result = "";
for(i=0; i < merged_intervals.length; i++) {
  result += merged_intervals[i].get_interval() + " ";
}
console.log(`Merged intervals: ${result}`)
//Output: [[1,5], [7,9]]
//Explanation: Since the first two intervals [1,4] and [2,5] overlap, we merged them i


merged_intervals = merge([new Interval(6, 7), new Interval(2, 4), new Interval(5, 9)])
result = "";
for(i=0; i < merged_intervals.length; i++) {
  result += merged_intervals[i].get_interval() + " ";
}
console.log(`Merged intervals: ${result}`)
//Output: [[2,4], [5,9]]
//Explanation: Since the intervals [6,7] and [5,9] overlap, we merged them into one [5


merged_intervals = merge([new Interval(1, 4), new Interval(2, 6), new Interval(3, 5)])
result = "";
```

```
for(i=0; i < merged_intervals.length; i++) {
  result += merged_intervals[i].get_interval() + " ";
}
console.log(`Merged intervals: ${result}`)
//Output: [[1,6]]
//Explanation: Since all the given intervals overlap, we merged them into one.
```

OR

```
function merge(intervals) {
  if(intervals.length < 2) return intervals

  //sort
  intervals.sort((a,b) => a[0]-b[0])

  for(let i = 1; i < intervals.length; i++) {
    let current = intervals[i]
    let previous = intervals[i-1]

    if(current[0] <= previous[1]) {
      intervals[i] = [previous[0], Math.max(previous[1], current[1])]
      intervals.splice(i-1, 1)
      i--
    }
  }

  return intervals
}

merge([[1,4], [2,5], [7,9]])//[[1,5], [7,9]], Since the first two intervals [1,4] and
merge([[6,7], [2,4], [5,9]])//[[2,4], [5,9]], Since the intervals [6,7] and [5,9] over
merge([[1,4], [2,6], [3,5]])//[[1,6]], Since all the given intervals overlap, we merge
merge([[2,5]])
```

- The time complexity of the above algorithm is `O(N * logN)`, where `N` is the total number of intervals. We are iterating the intervals only once which will take `O(N)`, in the beginning though, since we need to sort the intervals, our algorithm will take `O(N * logN)`.
- The space complexity of the above algorithm will be `O(N)` as we need to return a list containing all the merged intervals. We will also need `O(N)` space for sorting

> Given a set of intervals, find out if any two intervals overlap.

```
Intervals: [[1,4], [2,5], [7,9]]
Output: true
Explanation: Intervals [1,4] and [2,5] overlap
```

We can follow the same approach as discussed above to find if any two intervals overlap.

```
function anyOverlap(intervals) {
 //edge cases?
 if(intervals.length < 2) return false

 //already sorted?
 intervals.sort((a, b) => a[0] - b[0])

 for(let i = 1; i < intervals.length; i++) {
   let current = intervals[i]
   let previous = intervals[i-1]
  if(current[0] <= previous[1]) return true
 }

 return false
}

anyOverlap([[1,4], [2,5], [7,9]])//true, Intervals [1,4] and [2,5] overlap
anyOverlap([[1,2], [3,4], [5,6]])
anyOverlap([[1,2]])
```

Insert Interval (medium)

https://leetcode.com/problems/insert-interval/

> Given a list of non-overlapping intervals sorted by their start time, **insert a given interval at the correct position** and merge all necessary intervals to produce a list that has only mutually exclusive intervals.

If the given list was not sorted, we could have simply appended the new interval to it and used the `merge()` function from **Merge Intervals**. But since the given list is sorted, we should try to come up with a solution better than `O(N * logN)`

When inserting a new interval in a sorted list, we need to first find the correct index where the new interval can be placed. In other words, we need to skip all the intervals which end before the start of the new interval. So we can iterate through the given sorted listed of intervals and skip all the intervals with the following condition: `intervals[i].end < newInterval.start` Once we have found the correct place, we can follow an approach similar to **Merge Intervals** to insert and/or merge the new interval. Let's call the new interval `a` and the first interval with the above condition `b`. There are five possibilities:

The diagram above clearly shows the merging approach. To handle all four merging scenarios, we need to do something like this:

```
    c.start = min(a.start, b.start)
    c.end = max(a.end, b.end)
```

Our overall algorithm will look like this:

1. Skip all intervals which end before the start of the new interval, i.e., skip all `intervals` with the following condition:

```
intervals[i].end < newInterval.start
```

2. Let's call the last interval `b` that does not satisfy the above condition. If `b` overlaps with the new interval (a) (i.e. `b.start <= a.end`), we need to merge them into a new interval `c`:

```
c.start = min(a.start, b.start)
c.end = max(a.end, b.end)
```

3. We will repeat the above two steps to merge `c` with the next overlapping interval.

```
class Interval {
  constructor(start, end) {
    this.start = start;
    this.end = end;
  }

  print_interval() {
    process.stdout.write(`[${this.start}, ${this.end}]`);
  }
}

function insert (intervals, newInterval) {
  let merged = [];
  let i = 0

  //skip and add to output all intervals that come before the newInterval
  while(i < intervals.length && intervals[i].end < newInterval.start) {
    merged.push(intervals[i])
    i++
  }

  // merge all intervals that overlap with newInterval
  while(i < intervals.length && intervals[i].start <= newInterval.end) {
    newInterval.start = Math.min(intervals[i].start, newInterval.start)
    newInterval.end = Math.max(intervals[i].end, newInterval.end)
    i++
  }

  //insert the newInterval
  merged.push(newInterval)

  //add all the remaining intervals to the output
  while(i < intervals.length) {
    merged.push(intervals[i])
    i++
  }
  return merged;
```

```
};

//Input: Intervals=[[1,3], [5,7], [8,12]], New Interval=[4,6]
// Output: [[1,3], [4,7], [8,12]]
// Explanation: After insertion, since [4,6] overlaps with [5,7], we merged them into
process.stdout.write('Intervals after inserting the new interval: ');
let result = insert([
  new Interval(1, 3),
  new Interval(5, 7),
  new Interval(8, 12),
], new Interval(4, 6));
for (i = 0; i < result.length; i++) {
  result[i].print_interval();
}
console.log();

// Input: Intervals=[[1,3], [5,7], [8,12]], New Interval=[4,10]
// Output: [[1,3], [4,12]]
// Explanation: After insertion, since [4,10] overlaps with [5,7] & [8,12], we merged
process.stdout.write('Intervals after inserting the new interval: ');
result = insert([
  new Interval(1, 3),
  new Interval(5, 7),
  new Interval(8, 12),
], new Interval(4, 10));
for (i = 0; i < result.length; i++) {
  result[i].print_interval();
}
console.log();

// Input: Intervals=[[2,3],[5,7]], New Interval=[1,4]
// Output: [[1,4], [5,7]]
// Explanation: After insertion, since [1,4] overlaps with [2,3], we merged them into
process.stdout.write('Intervals after inserting the new interval: ');
result = insert([new Interval(2, 3),
  new Interval(5, 7),
], new Interval(1, 4));
for (i = 0; i < result.length; i++) {
  result[i].print_interval();
}
console.log();
```

OR

```
function insert(intervals, newInterval) {
  let merged = []

  let i = 0

  //skip and add to output all intervals that come before the newInterval
  while(i < intervals.length && intervals[i][1] < newInterval[0]) {
    merged.push(intervals[i])
    i++
```

```
    }

    //merge all intervals that overlap with newInterval
    while(i < intervals.length && intervals[i][0] <= newInterval[1]) {
      newInterval[0] = Math.min(intervals[i][0], newInterval[0])
      newInterval[1] = Math.max(intervals[i][1], newInterval[1])
      i++
    }

    //insert the newInterval
    merged.push(newInterval)

    //add the remaining intervals to the output
    while(i < intervals.length) {
      merged.push(intervals[i])
      i++
    }

    return merged
  }

  insert([[1,3], [5,7], [8,12]], [4,6])//[[1,3], [4,7], [8,12]], After insertion, since
  insert([[1,3], [5,7], [8,12]], [4,10])// [[1,3], [4,12]], After insertion, since [4,10
  insert([[2,3],[5,7]], [1,4])//[[1,4], [5,7]], After insertion, since [1,4] overlaps wi
```

- As we are iterating through all the intervals only once, the time complexity of the above algorithm is `O(N)`, where `N` is the total number of intervals.
- The space complexity of the above algorithm will be `O(N)` as we need to return a list containing all the merged intervals.

Intervals Intersection (medium)

https://leetcode.com/problems/interval-list-intersections/

> Given two lists of intervals, find the **intersection of these two lists**. Each list consists of **disjoint intervals sorted on their start time**.

This problem follows the **Merge Intervals** pattern. As we have discussed under **Insert Interval**, there are five overlapping possibilities between two intervals ‘a’ and ‘b’. A close observation will tell us that whenever the two intervals overlap, one of the interval’s start time lies within the other interval. This rule can help us identify if any two intervals overlap or not.

Now, if we have found that the two intervals overlap, how can we find the overlapped part?

Again from the above diagram, the overlapping interval will be equal to:

```
start = max(a.start, b.start)
end = min(a.end, b.end)
```

That is, the highest start time and the lowest end time will be the overlapping interval.

So our algorithm will be to iterate through both the lists together to see if any two intervals overlap. If two intervals overlap, we will insert the overlapped part into a result list and move on to the next interval which is finishing early.

```
function findIntersection(firstIntervals, secondIntervals) {
  let result = []

  let i = 0
  let j = 0

  while(i < firstIntervals.length && j < secondIntervals.length) {
    //check if intervals overlap and firstIntervals[i] start time
    //lies within the other secondIntervals[j]
    let firstOverlapsSecond = firstIntervals[i][0] >= secondIntervals[j][0] && firstIn

    //check if intervals overlap and firstIntervals[j]'s start time
    //lies within the other secondInterval[i]
    let secondOverlapsFirst = secondIntervals[j][0] >= firstIntervals[i][0] && secondI

    //store the intersection part
    if(firstOverlapsSecond || secondOverlapsFirst) {
      result.push([Math.max(firstIntervals[i][0], secondIntervals[j][0]), Math.min(fir
    }

    //move next from the interval which is finishing first
    if(firstIntervals[i][1] < secondIntervals[j][1]) {
      i++
    } else {
      j++
    }
  }

  return result
}


findIntersection([[1, 3], [5, 6], [7, 9]], [[2, 3], [5, 7]])//[2, 3], [5, 6], [7, 7],
findIntersection([[1, 3], [5, 7], [9, 12]], [[5, 10]])// [5, 7], [9, 10], The output l
```

- As we are iterating through both the lists once, the time complexity of the above algorithm is `O(N + M)`, where `'N'` and `'M'` are the total number of intervals in the input arrays respectively.
- Ignoring the space needed for the result list, the algorithm runs in constant space `O(1)`.

Conflicting Appointments (medium)

https://leetcode.com/problems/meeting-rooms/

> Given an array of intervals representing 'N' appointments, find out if a person can **attend all the appointments**.

The problem follows the **Merge Intervals** pattern. We can sort all the intervals by start time and then check if any two intervals overlap. A person will not be able to attend all appointments if any two appointments overlap.

```
function canAttendAllAppointments(appointmentTimes) {
  //sort intervals by start time
  appointmentTimes.sort((a,b) => a[0] -b[0])

  //check if any two intervals overlap
  for(let i = 1; i < appointmentTimes.length; i++) {
    if(appointmentTimes[i][0] < appointmentTimes[i-1][1]) {
      //note that in the comparison above, it is < and not <=
      //while merging we needed <= comparison, as we will be merging the two
      //intervals have conditions appointmentTimes[i][0] === appointmentTimes[i-1][1]
      //but such intervals don't represent conflicting appointments
      //as one starts right after the other
      return false
    }
  }
  return true
}

canAttendAllAppointments([[1,4], [2,5], [7,9]])//false, Since [1,4] and [2,5] overlap,
canAttendAllAppointments([[6,7], [2,4], [8,12]])//true, None of the appointments overl
canAttendAllAppointments([[4,5], [2,3], [3,6]])//false, Since [4,5] and [3,6] overlap,
```

- The time complexity of the above algorithm is `O(N*logN)`, where `'N'` is the total number of appointments. Though we are iterating the intervals only once, our algorithm will take `O(N * logN)` since we need to sort them in the beginning.
- The space complexity of the above algorithm will be `O(N)`, which we need for sorting.

> 🌟 Given a list of appointments, find all the conflicting appointments.

```
Appointments: [[4,5], [2,3], [3,6], [5,7], [7,8]]
Output:
[4,5] and [3,6] conflict.
[3,6] and [5,7] conflict.
```

REVIEW

```
function whatAreTheConflicts(appointmentTimes)  {

  // appointmentTimes.sort((a,b) => a[0]-b[0])

  let conflicts = []

  for(let i = 0; i < appointmentTimes.length -1; i++) {
    for(let j = 1; j < appointmentTimes.length; j++) {
      if((j!==i) && (appointmentTimes[i][1] > appointmentTimes[j][0])) {
        conflicts.push([appointmentTimes[j], appointmentTimes[i]])
      }
    }
 }
 // console.log(appointmentTimes)
 return conflicts
}


 whatAreTheConflicts([[4,5], [2,3], [3,6], [5,7], [7,8]])
//[4,5] and [3,6] conflict.
//[3,6] and [5,7] conflict.
```

🌟 Minimum Meeting Rooms (hard)

https://leetcode.com/problems/meeting-rooms-ii/

> Given a list of intervals representing the start and end time of 'N' meetings, find the **minimum number of rooms** required to **hold all the meetings**.

Example :

```
Meetings: [[4,5], [2,3], [2,4], [3,5]]
Output: 2
Explanation: We will need one room for [2,3] and [3,5], and another room for [2,4] and
```

Let's take the above-mentioned example (4) and try to follow our **Merge Intervals** approach:

1. Sorting these meetings on their start time will give us: `[[2,3], [2,4], [3,5], [4,5]]`
2. Merging overlapping meetings:

- `[2,3]` overlaps with `[2,4]`, so after merging we'll have => `[[2,4], [3,5], [4,5]]`
- `[2,4]` overlaps with `[3,5]`, so after merging we'll have => `[[2,5], [4,5]]`
- `[2,5]` overlaps `[4,5]`, so after merging we'll have => `[2,5]`

Since all the given meetings have merged into one big meeting (`[2,5]`), does this mean that they all are overlapping and we need a minimum of four rooms to hold these meetings? You might have already guessed that the answer is NO! As we can clearly see, some meetings are mutually

exclusive. For example, `[2,3]` and `[3,5]` do not overlap and can happen in one room. So, to correctly solve our problem, we need to keep track of the mutual exclusiveness of the overlapping meetings.

Here is what our strategy will look like:

1. We will sort the meetings based on start time.
2. We will schedule the first meeting (let's call it `m1`) in one room (let's call it `r1`).
3. If the next meeting `m2` is not overlapping with `m1`, we can safely schedule it in the same room `r1`.
4. If the next meeting `m3` is overlapping with `m2` we can't use `r1`, so we will schedule it in another room (let's call it `r2`).
5. Now if the next meeting `m4` is overlapping with `m3`, we need to see if the room `r1` has become free. For this, we need to keep track of the end time of the meeting happening in it. If the end time of `m2` is before the start time of `m4`, we can use that room `r1`, otherwise, we need to schedule `m4` in another room `r3`.

We can conclude that we need to **keep track of the ending time of all the meetings currently happening** so that when we try to schedule a new meeting, we can see what meetings have already ended. We need to put this information in a data structure that can easily give us the smallest ending time. A **Min Heap** would fit our requirements best.

So our algorithm will look like this:

1. Sort all the meetings on their start time.
2. Create a min-heap to store all the active meetings. This min-heap will also be used to find the active meeting with the smallest end time.
3. Iterate through all the meetings one by one to add them in the min-heap. Let's say we are trying to schedule the meeting `m1`.
4. Since the min-heap contains all the active meetings, so before scheduling `m1` we can remove all meetings from the heap that have ended before `m1`, i.e., remove all meetings from the heap that have an end time smaller than or equal to the start time of `m1`.
5. Now add `m1` to the heap.
6. The heap will always have all the overlapping meetings, so we will need rooms for all of them. Keep a counter to remember the maximum size of the heap at any time which will be the minimum number of rooms needed.

```
function minMeetingRooms(meetings) {
  //JavaScript does not come with built in Heap, so I used an array to keep track of r
  if(meetings == null) return 0
  if(meetings.length <= 1) return meetings.length

  //helper that returns the meeting room with the earliest end time
  function getEarliest(room) {
    room.sort((a,b) => a[1]-b[1])
```

```
    return rooms[0]
  }

  //sort meetings on start time
  meetings.sort((a,b) => a[0]-b[0])

  let rooms = [meetings[0]]

  for(let i = 1; i < meetings.length; i++) {
    let earliestRoom = getEarliest(rooms)
    let currentTime = meetings[i]

    //if the room time ends before the currentTime interval starts
    //then use the room and update the room end time to currentTime
    if(earliestRoom[1] <= currentTime[0]) {
      earliestRoom[1] = currentTime[1]
    } else {
      //create room
      rooms.push(currentTime)
    }
  }
  return rooms.length
}

minMeetingRooms()
minMeetingRooms([[1,4]])
minMeetingRooms([[1,4], [2,5], [7,9]])//2, Since [1,4] and [2,5] overlap, we need two
minMeetingRooms([[6,7], [2,4], [8,12]])//1, None of the meetings overlap, therefore we
minMeetingRooms([[1,4], [2,3], [3,6]])//2, Since [1,4] overlaps with the other two mee
minMeetingRooms([[4,5], [2,3], [2,4], [3,5]])//2, We will need one room for [2,3] and
```

- The time complexity of the above algorithm is `O(N*logN)`, where `'N'` is the total number of meetings. This is due to the sorting that we did in the beginning. Also, while iterating the meetings we might need to poll/offer meeting to the priority queue. Each of these operations can take `O(logN)`. Overall our algorithm will take `O(NlogN)`.
- The space complexity of the above algorithm will be `O(N)` which is required for sorting. Also, in the worst case scenario, we'll have to insert all the meetings into the Min Heap (when all meetings overlap) which will also take `O(N)` space. The overall space complexity of our algorithm is `O(N)`.

Similar Problems

> Given a list of intervals, find the point where the maximum number of intervals overlap.

> Given a list of intervals representing the arrival and departure times of trains to a train station, our goal is to find the minimum number of platforms required for the train station so that no train has to wait.

Both of these problems can be solved using the approach discussed above.

🌟 Maximum CPU Load (hard)

https://leetcode.com/problems/car-pooling/

We are given a list of Jobs. Each job has a Start time, an End time, and a CPU load when it is running. Our goal is to find the **maximum CPU load** at any time if all the jobs are **running on the same machine**.

Example 1:

```
Jobs: [[1,4,3], [2,5,4], [7,9,6]]
Output: 7
Explanation: Since [1,4,3] and [2,5,4] overlap, their maximum CPU load (3+4=7) will be
jobs are running at the same time i.e., during the time interval (2,4).
```

Example 2:

```
Jobs: [[6,7,10], [2,4,11], [8,12,15]]
Output: 15
Explanation: None of the jobs overlap, therefore we will take the maximum load of any
```

Example 3:

```
Jobs: [[1,4,2], [2,4,1], [3,6,5]]
Output: 8
Explanation: Maximum CPU load will be 8 as all jobs overlap during the time interval [
```

The problem follows the **Merge Intervals** pattern and can easily be converted to **Minimum Meeting Rooms**. Similar to 'Minimum Meeting Rooms' where we were trying to find the maximum number of meetings happening at any time, for 'Maximum CPU Load' we need to find the maximum number of jobs running at any time. We will need to keep a running count of the maximum CPU load at any time to find the overall maximum load.

```
function findMaxCPULoad(jobs) {
  //sort the jobs by start time
  jobs.sort((a, b) => a[0]-b[0])

  let maxCPULoad = 0

 //consolidate jobs that overlap
  for(let i = 1; i < jobs.length; i++) {
    let current = jobs[i]
    let previous = jobs[i-1]

    if(current[0] < previous[1]){
```

```
        jobs[i] = [previous[0], current[1], previous[2] + current[2]]
        jobs.splice(i-1, 1)
        i--
      }
    }

    //set maximum load
    for(let i = 0; i < jobs.length; i++) {
      maxCPULoad = Math.max(maxCPULoad, jobs[i][2])
    }

    return maxCPULoad;
  };

  findMaxCPULoad([[1,4,3], [2,5,4], [7,9,6]])//7, Since [1,4,3] and [2,5,4] overlap, th
  findMaxCPULoad([[6,7,10], [2,4,11], [8,12,15]])//15, None of the jobs overlap, theref
  findMaxCPULoad([[1,4,2], [2,4,1], [3,6,5]])//8, Maximum CPU load will be 8 as all job
```

- The time complexity of the above algorithm is `O(N*logN)`, where `N` is the total number of jobs. This is due to the sorting that we did in the beginning. Also, while iterating the jobs, we might need to poll/offer jobs to the priority queue. Each of these operations can take `O(logN)`. Overall our algorithm will take `O(NlogN)`.
- The space complexity of the above algorithm will be `O(N)`, which is required for sorting. Also, in the worst case, we have to insert all the jobs into the priority queue (when all jobs overlap) which will also take `O(N)` space. The overall space complexity of our algorithm is `O(N)`.

🌟 Employee Free Time (hard)

https://leetcode.com/problems/employee-free-time/

> For 'K' employees, we are given a list of intervals representing the working hours of each employee. Our goal is to find out if there is a **free interval that is common to all employees**. You can assume that each list of employee working hours is sorted on the start time.

This problem follows the Merge Intervals pattern. Let's take the an example:

```
Input: Employee Working Hours=[[[1,3], [9,12]], [[2,4]], [[6,8]]]
Output: [4,6], [8,9]
```

One simple solution can be to put all employees' working hours in a list and sort them on the start time. Then we can iterate through the list to find the gaps. Let's dig deeper. Sorting the intervals of the above example will give us:

```
[1,3], [2,4], [6,8], [9,12]
```

We can now iterate through these intervals, and whenever we find non-overlapping intervals (e.g., `[2,4]` and `[6,8]`), we can calculate a free interval (e.g., `[4,6]`).

```
function findEmployeeFreeTime (schedules) {
  let freeTime = [];

  //combine all schedules
  let allTime = []

  for(let i = 0; i < schedules.length; i++) {
    for(let j = 0; j <  schedules[i].length; j++) {
      allTime.push(schedules[i][j])
    }
  }
  allTime.sort((a,b) => a[0]-b[0])

  //merge overlap
  for(let i = 1; i < allTime.length; i++) {
    let current = allTime[i]
    let previous = allTime[i-1]

    if(current[0] <= previous[1]) {
      allTime[i] = [previous[0], current[1]]
      allTime.splice(i-1, 1)
      i--
    }
  }
  //whatever is not accounted for is free time
  for(let i = 1; i < allTime.length; i++) {
    freeTime.push([allTime[i-1][1], allTime[i][0]])
  }
    return freeTime;
};

findEmployeeFreeTime ([[[1,3], [5,6]], [[2,3], [6,8]]])//[3,5], Both the employees are
findEmployeeFreeTime ([[[1,3], [9,12]], [[2,4]], [[6,8]]])//[4,6], [8,9], All employee
findEmployeeFreeTime ([[[1,3]], [[2,4]], [[3,5], [7,9]]])//[5,7], ll employees are fre
```

- This algorithm will take `O(N * logN)` time, where `'N'` is the total number of intervals. This time is needed because we need to sort all the intervals.
- The space complexity will be `O(N)`, which is needed for sorting.

Can we find a better solution?

One fact that we are not utilizing is that each employee list is individually sorted!

How about we take the first interval of each employee and insert it in a `Min Heap`. This `Min Heap` can always give us the interval with the smallest start time. Once we have the smallest start-time interval, we can then compare it with the next smallest start-time interval (again from the `Heap`) to find the gap. This interval comparison is similar to what we suggested in the previous approach.

Whenever we take an interval out of the `Min Heap`, we can insert the same employee's next interval. This also means that we need to know which interval belongs to which employee.

- The above algorithm's time complexity is `O(N*logK)`, where `'N'` is the total number of intervals, and `'K'` is the total number of employees. This is because we are iterating through the intervals only once (which will take `O(N)`), and every time we process an interval, we remove (and can insert) one interval in the `Min Heap`, (which will take `O(logK)`. At any time, the heap will not have more than `'K'` elements.
- The space complexity of the above algorithm will be `O(K)` as at any time, the heap will not have more than `'K'` elements.

Pattern 5: Cyclic Sort

This pattern describes an interesting approach to deal with problems involving arrays containing numbers in a given range. For example, take the following problem:

> You are given an unsorted array containing numbers taken from the range 1 to 'n'. The array can have duplicates, which means that some numbers will be missing. Find all the missing numbers.

To efficiently solve this problem, we can use the fact that the input array contains numbers in the range of `1` to `'n'`. For example, to efficiently sort the array, we can try placing each number in its correct place, i.e., placing `'1'` at index `'0'`, placing `'2'` at index `'1'`, and so on. Once we are done with the sorting, we can iterate the array to find all indices that are missing the correct numbers. These will be our required numbers.

Cyclic Sort (easy)

We are given an array containing `'n'` objects. Each object, when created, was assigned a unique number from `1` to `'n'` based on their creation sequence. This means that the object with sequence number `'3'` was created just before the object with sequence number `'4'`.

> Write a function to sort the objects in-place on their creation sequence number in `O(n)` and without any extra space.

For simplicity, let's assume we are passed an integer array containing only the sequence numbers, though each number is actually an object. As we know, the input array contains numbers in the range of `1` to `'n'`. We can use this fact to devise an efficient way to sort the numbers. Since all numbers are unique, we can try placing each number at its correct place, i.e., placing `'1'` at index `'0'`, placing `'2'` at index `'1'`, and so on.

To place a number (or an object in general) at its correct index, we first need to find that number. If we first find a number and then place it at its correct place, it will take us $O(N^2)$, which is not acceptable.

Instead, what if we iterate the array one number at a time, and if the current number we are iterating is not at the correct index, we swap it with the number at its correct index. This way we will go through all numbers and place them in their correct indices, hence, sorting the whole array.

```
function cyclicSort (nums) {
  let i = 0;

  while(i < nums.length) {
```

```
      const j = nums[i] - 1//nums[i] = 3, 3-1 = 2
      if(nums[i] !== nums[j]) {//3 !== 2
        //swap
        // [nums[i], nums[j]] = [nums[j], nums[i]]
        let temp = nums[i]
        nums[i] = nums[j]
        nums[j] = temp
      } else {
        i++
      }
    }
    return nums;
  }


  cyclicSort ([3, 1, 5, 4, 2])
  cyclicSort ([2, 6, 4, 3, 1, 5])
  cyclicSort ([1, 5, 6, 4, 3, 2])
```

- The time complexity of the above algorithm is `O(n)`. Although we are not incrementing the index `i` when swapping the numbers, this will result in more than `'n'` iterations of the loop, but in the worst-case scenario, the while loop will swap a total of `'n-1'` numbers and once a number is at its correct index, we will move on to the next number by incrementing `i`. So overall, our algorithm will take `O(n) + O(n-1)` which is asymptotically equivalent to `O(n)`. - The algorithm runs in constant space `O(1)`.

Find the Missing Number (easy)

https://leetcode.com/problems/missing-number/

> We are given an array containing `'n'` distinct numbers taken from the range `0` to `'n'`. Since the array has only `'n'` numbers out of the total `'n+1'` numbers, find the missing number.

This problem follows the **Cyclic Sort** pattern. Since the input array contains unique numbers from the range `0` to `'n'`, we can use a similar strategy as discussed in **Cyclic Sort** to place the numbers on their correct index. Once we have every number in its correct place, we can iterate the array to find the index which does not have the correct number, and that index will be our missing number.

However, there are two differences with Cyclic Sort:

1. In this problem, the numbers are ranged from `'0'` to `'n'`, compared to `'1'` to `'n'` in the Cyclic Sort. This will make two changes in our algorithm:
    - In this problem, each number should be equal to its index, compared to `index - 1` in the **Cyclic Sort**. Therefore => `nums[i] == nums[nums[i]]`

    - Since the array will have `'n'` numbers, which means array indices will range from `0` to `'n-1'`. Therefore, we will ignore the number `'n'` as we can't place it in the array, so => `nums[i] < nums.length`

2. Say we are at index `i`. If we swap the number at index `i` to place it at the correct index, we can still have the wrong number at index `i`. This was true in **Cyclic Sort** too. It didn't cause any problems in **Cyclic Sort** as over there, we made sure to place one number at its correct place in each step, but that wouldn't be enough in this problem as we have one extra number due to the larger range. Therefore, we will not move to the next number after the swap until we have a correct number at the index `i`.

```
function findMissingNumber(nums) {
  let i = 0
  const n = nums.length

  //sort first
  while(i < n) {
    let j = nums[i]
    if(nums[i] < n && nums[i] !== nums[j]) {//0 < 4 && 0 !== 4
      //swap
      [nums[i], nums[j]] = [nums[j], nums[i]]
    } else {
      i++
    }
  }

  //find the first number missing from it's index
  //that will be our required number
  for(i = 0; i < n; i++) {
    if(nums[i] !== i) {
      return i
    }
  }
  return n
}

findMissingNumber([4, 0, 3, 1])//2
findMissingNumber([8, 3, 5, 2, 4, 6, 0, 1])//7
```

- The time complexity of the above algorithm is `O(n)`. In the while loop, although we are not incrementing the index `i` when swapping the numbers, this will result in more than `n` iterations of the loop, but in the worst-case scenario, the while loop will swap a total of `n-1` numbers and once a number is at its correct index, we will move on to the next number by incrementing `i`. In the end, we iterate the input array again to find the first number missing from its index, so overall, our algorithm will take `O(n) + O(n-1) + O(n)` which is asymptotically equivalent to `O(n)`.
- The algorithm runs in constant space `O(1)`.

Find all Missing Numbers (easy)

https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/

> We are given an unsorted array containing numbers taken from the range `1 to 'n'`. The array can have duplicates, which means some numbers will be missing. Find all those missing numbers.

This problem follows the **Cyclic Sort** pattern and shares similarities with **Find the Missing Number** with one difference. In this problem, there can be many duplicates whereas in **Find the Missing Number** there were no duplicates and the range was greater than the length of the array.

However, we will follow a similar approach though as discussed in **Find the Missing Number** to place the numbers on their correct indices. Once we are done with the cyclic sort we will iterate the array to find all indices that are missing the correct numbers.

```
function findMissingNumbers(nums) {
  let i = 0

  while(i < nums.length) {
    const j = nums[i] - 1;

    if(nums[i] !== nums[j]) {
      //swap
      [nums[i], nums[j]] = [nums[j], nums[i]]
    } else {
      i++
    }
  }

  let missingNumbers = []

  for( i = 0; i < nums.length; i++) {
    if(nums[i] !== i + 1) {
      missingNumbers.push(i + 1)
    }
  }

  return missingNumbers
}

findMissingNumbers([2, 3, 1, 8, 2, 3, 5, 1])//[4, 6, 7], The array should have all num
findMissingNumbers([2, 4, 1, 2])//3
findMissingNumbers([2, 3, 2, 1])//4
```

- The time complexity of the above algorithm is `O(n)`.
- Ignoring the space required for the output array, the algorithm runs in constant space `O(1)`.

Find the Duplicate Number (easy)

https://leetcode.com/problems/find-the-duplicate-number/

> We are given an unsorted array containing 'n+1' numbers taken from the range 1 to 'n'. The array has only one duplicate but it can be repeated multiple times. **Find that duplicate number without using any extra space**. You are, however, allowed to modify the input array.

This problem follows the **Cyclic Sort** pattern and shares similarities with **Find the Missing Number**. Following a similar approach, we will try to place each number on its correct index. Since there is only one duplicate, if while swapping the number with its index both the numbers being swapped are same, we have found our duplicate!

```
function findDuplicate(nums) {
  let i = 0

  while(i < nums.length) {
    if(nums[i] !== i + 1) {
      let j = nums[i] - 1
      if(nums[i] !== nums[j]) {
        //swap
        [nums[i], nums[j]] = [nums[j], nums[i]]
      } else {
        //we have found the duplicate
        return nums[i]
      }
    } else {
      i++
    }
  }
  return -1
}

findDuplicate([1, 4, 4, 3, 2])//4
findDuplicate([2, 1, 3, 3, 5, 4])//3
findDuplicate([2, 4, 1, 4, 4])//4
```

- The time complexity of the above algorithm is `O(n)`.
- The algorithm runs in constant space `O(1)` but modifies the input array.

> Can we solve the above problem in `O(1)` space and without modifying the input array?

While doing the cyclic sort, we realized that the array will have a cycle due to the duplicate number and that the start of the cycle will always point to the duplicate number. This means that we can use the **fast & the slow** pointer method to find the duplicate number or the start of the cycle similar to Start of LinkedList Cycle.

```
function findDuplicate(nums) {
  //using fast & slow pointer method
  let slow = nums[0]
  let fast = nums[nums[0]]
```

```
    while(slow !== fast) {
      slow = nums[slow]
      fast = nums[nums[fast]]
    }
    //find the cycle length
    let current = nums[nums[slow]]
    let cycleLength = 1
    while(current !== nums[slow]) {
      current = nums[current]
      cycleLength++
    }

    return findStart(nums, cycleLength)
  }

  function findStart(nums, cycleLength){
    let pointer1 = nums[0]
    let pointer2 = nums[0]
    //move pointer2 ahead by cycleLength steps
    while(cycleLength > 0) {
      pointer2 = nums[pointer2]
      cycleLength--
    }
    //increment both pointers until they meet at the start of the cycle
    while(pointer1 !== pointer2) {
      pointer1 = nums[pointer1]
      pointer2 = nums[pointer2]
    }

    return pointer1
  }

  findDuplicate([1, 4, 4, 3, 2])//4
  findDuplicate([2, 1, 3, 3, 5, 4])//3
  findDuplicate([2, 4, 1, 4, 4])//4
```

- The time complexity of the above algorithm is `O(n)` and the space complexity is `O(1)`.

Find all Duplicate Numbers (easy)

https://leetcode.com/problems/find-all-duplicates-in-an-array/

> We are given an unsorted array containing `'n'` numbers taken from the range `1` to `'n'`. The array has some numbers appearing twice, **find all these duplicate numbers without using any extra space**.

This problem follows the **Cyclic Sort** pattern and shares similarities with **Find the Duplicate Number**. Following a similar approach, we will place each number at its correct index. After that, we will iterate through the array to find all numbers that are not at the correct indices. All these numbers are duplicates.

```
function findAllDuplicates(nums) {
  let i = 0

  while(i < nums.length) {
    //??
    let j = nums[i] - 1

    if(nums[i] !== nums[j]) {
      //swap
      [nums[i], nums[j]] = [nums[j], nums[i]]
    } else {
        i++
      }
    }

  let duplicateNumbers = []

  for(i = 0; i < nums.length; i++) {
    if(nums[i] !== i + 1) {
      //we have found the duplicate
        duplicateNumbers.push(nums[i])
       }
  }

  return duplicateNumbers
}

findAllDuplicates([3, 4, 4, 5, 5])//[4, 5]
findAllDuplicates([5, 4, 7, 2, 3, 5, 3])//[3, 5]
```

- The time complexity of the above algorithm is `O(n)`.
- Ignoring the space required for storing the duplicates, the algorithm runs in constant space `O(1)`.

🌟 Find the Corrupt Pair (easy)

> We are given an unsorted array containing `‘n’` numbers taken from the range `1` to `‘n’`. The array originally contained all the numbers from `1` to `‘n’`, but due to a data error, one of the numbers got duplicated which also resulted in one number going missing. Find both these numbers.

```
function findCorruptNumbers(nums) {
  let i = 0

  while(i < nums.length) {
    const j = nums[i] - 1
    if(nums[i] !== nums[j]) {
      //swap
      [nums[i], nums[j]] = [nums[j], nums[i]]
```

```
    } else {
      i++
    }
  }

  //output => duplicate number(nums[i]) and the missing number(i+1)
  for(let i = 0; i < nums.length; i++) {
    if(nums[i] !== i + 1) {
      return [nums[i], i + 1]
    }
  }
  return [-1, -1]
};


findCorruptNumbers([3, 1, 2, 5, 2])//[2, 4], '2' is duplicated and '4' is missing.
findCorruptNumbers([3, 1, 2, 3, 6, 4])// [3, 5], '3' is duplicated and '5' is missing.
```

- The time complexity of the above algorithm is `O(n)`.
- The algorithm runs in constant space `O(1)`.

🌟 Find the Smallest Missing Positive Number (medium)

https://leetcode.com/problems/first-missing-positive/

> Given an unsorted array containing numbers, find the **smallest missing positive number** in it.

This problem follows the **Cyclic Sort** pattern and shares similarities with **Find the Missing Number** with one big difference. In this problem, the numbers are not bound by any range so we can have any number in the input array.

However, we will follow a similar approach though as discussed in **Find the Missing Number** to place the numbers on their correct indices and ignore all numbers that are out of the range of the array (i.e., all negative numbers and all numbers greater than or equal to the length of the array). Once we are done with the cyclic sort we will iterate the array and the first index that does not have the correct number will be the smallest missing positive number!

```
function findFirstSmallestMissingPositive(nums) {
  //try to sort the array
  let i = 0
  let n = nums.length

  while(i < n) {
    const j = nums[i] - 1
    if(nums[i] !== nums[j] && nums[i] > 0 && nums[i] <= n){
      [nums[i], nums[j]] = [nums[j], nums[i]]
    } else {
      i++
    }
```

```
    }


    for(let i = 0; i < n; i++) {
      if(nums[i] !== i + 1) {
        return i + 1
      }
    }

    return nums.length + 1;
  };

  findFirstSmallestMissingPositive([-3, 1, 5, 4, 2])//3, The smallest missing positive n
  findFirstSmallestMissingPositive([3, -2, 0, 1, 2])//4
  findFirstSmallestMissingPositive([3, 2, 5, 1])//4
```

- The time complexity of the above algorithm is `O(n)`.
- The algorithm runs in constant space `O(1)`.

🌟 Find the First K Missing Positive Numbers (hard)

https://leetcode.com/problems/kth-missing-positive-number/

> Given an unsorted array containing numbers and a number `‘k’`, find the first `‘k’` missing positive numbers in the array.

This problem follows the **Cyclic Sort** pattern and shares similarities with **Find the Smallest Missing Positive Number**. The only difference is that, in this problem, we need to find the first `‘k’` missing numbers compared to only the first missing number.

We will follow a similar approach as discussed in **Find the Smallest Missing Positive Number** to place the numbers on their correct indices and ignore all numbers that are out of the range of the array. Once we are done with the cyclic sort we will iterate through the array to find indices that do not have the correct numbers.

If we are not able to find `‘k’` missing numbers from the array, we need to add additional numbers to the output array. To find these additional numbers we will use the length of the array. For example, if the length of the array is `4`, the next missing numbers will be `4, 5, 6` and so on. One tricky aspect is that any of these additional numbers could be part of the array. Remember, while sorting, we ignored all numbers that are greater than or equal to the length of the array. So all indices that have the missing numbers could possibly have these additional numbers. To handle this, we must keep track of all numbers from those indices that have missing numbers. Let’s understand this with an example:

```
nums: [2, 1, 3, 6, 5], k =2
```

After the cyclic sort our array will look like:

```
nums: [1, 2, 3, 6, 5]
```

From the sorted array we can see that the first missing number is `'4'` (as we have `'6'` on the fourth index) but to find the second missing number we need to remember that the array does contain `'6'`. Hence, the next missing number is `'7'`.

```
function findFirstKMissingPositive(nums, k) {
  //sort? the input array
  let i = 0
  const n = nums.length

  while(i < n) {
    const j = nums[i] - 1
    if(nums[i] !== nums[j] && nums[i] <= n && nums[i] > 0) {
      //swap
      [nums[i], nums[j]] = [nums[j], nums[i]]
    } else {
      i++
    }
  }

  const missingNumbers = []
  const extraNumbers = new Set()

    for(let i = 0; i < n; i++) {
      if(missingNumbers.length < k) {
        if(nums[i] !== i + 1) {
          missingNumbers.push(i+1)
          extraNumbers.add(nums[i])
        }
      }
    }

  //add the remaining missing numbers
  let j = 1

  while(missingNumbers.length < k) {
    const currentNumber = j + n
    //ignore if the array contains the current number
    if(!extraNumbers.has(currentNumber)) {
      missingNumbers.push(currentNumber)
      }
    j++
  }

  return missingNumbers;
};

findFirstKMissingPositive([3, -1, 4, 5, 5], 3)//[1, 2, 6], The smallest missing positi
findFirstKMissingPositive([2, 3, 4], 3)//[1, 5, 6], The smallest missing positive numb
```

```
findFirstKMissingPositive([-2, -3, 4], 2)//[1, 2], The smallest missing positive numbe
findFirstKMissingPositive([2, 1, 3, 6, 5],2)//[4, 7]
```

- The time complexity of the above algorithm is `O(n + k)`, as the last two for loops will run for `O(n)` and `O(k)` times respectively.
- The algorithm needs `O(k)` space to store the `extraNumbers`.

Pattern 6: In-place Reversal of a LinkedList

In a lot of problems, we are asked to reverse the links between a set of nodes of a **LinkedList**. Often, the constraint is that we need to do this in-place, i.e., using the existing node objects and without using extra memory.

In-place Reversal of a LinkedList pattern describes an efficient way to solve the above problem.

Reverse a LinkedList (easy)

https://leetcode.com/problems/reverse-linked-list/

> Given the head of a Singly LinkedList, reverse the LinkedList. Write a function to return the new head of the reversed LinkedList.

To reverse a **LinkedList**, we need to reverse one node at a time. We will start with a variable `current` which will initially point to the head of the LinkedList and a variable `previous` which will point to the previous node that we have processed; initially `previous` will point to `null`.

In a stepwise manner, we will reverse the `current` node by pointing it to the `previous` before moving on to the next node. Also, we will update the `previous` to always point to the previous node that we have processed.

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

  printList() {
    let result = ""
    let temp = this
    while(temp !== null) {
      result += temp.value + " "
      temp = temp.next
    }
    return result
  }
}

function reverse(head) {
  let current = head
  let previous = null

  while(current !== null) {
    //temporarily store the next node
    next = current.next
```

```
    //reverse the current node
    current.next = previous

    //before we move to the next node,
    //point previous to the current node
    previous = current

    //move on to the next node
    current = next
  }

  return previous
}

head = new Node(2)
head.next = new Node(4);
head.next.next = new Node(6);
head.next.next.next = new Node(8);
head.next.next.next.next = new Node(10);

console.log(`Nodes of original LinkedList are: ${head.printList()}`)
console.log(`Nodes of reversed LinkedList are: ${reverse(head).printList()}`)
```

- The time complexity of our algorithm will be `O(N)` where `'N'` is the total number of nodes in the LinkedList.
- We only used constant space, therefore, the space complexity of our algorithm is `O(1)`.

Reverse a Sub-list (medium)

https://leetcode.com/problems/reverse-linked-list-ii/

> Given the head of a LinkedList and two positions `p` and `q`, reverse the LinkedList from position `p` to `q`.

The problem follows the **In-place Reversal** of a LinkedList pattern. We can use a similar approach as discussed in **Reverse a LinkedList**. Here are the steps we need to follow:

1. Skip the first `p-1` nodes, to reach the node at position `p`.
2. Remember the node at position `p-1` to be used later to connect with the reversed sub-list.
3. Next, reverse the nodes from `p` to `q` using the same approach discussed in **Reverse a LinkedList**.
4. Connect the `p-1` and `q+1` nodes to the reversed sub-list.

```
class Node {
  constructor(value, next = null) {
    this.value = value
    this.next = next
```

```
    }

    getList() {
      let result = ""
      let temp = this
      while(temp !== null) {
        result += temp.value + " "
        temp = temp.next
      }
      return result
    }
  }

  function reverseSubList(head, p, q) {
    if(p === q) {
      return head
    }

    //after skipping p-1 nodes, current will
    //point to the p th node

    let current = head
    let previous = null

    let i = 0

    while(current !== null && i < p - 1) {
      previous = current
      current = current.next
      i++
    }

    //we are interested in three parts ofthe LL,
    //1. the part before index p
    //2. the part between p and q
    //3. and the part after index q

    const lastNodeOfFirstPart = previous

    //after reversing the LL current will
    //become the last node of the subList
    const lastNodeOfSubList = current

    //will be used to temporarily store the next node
    let next = null

    i = 0
    //reverse nodes between p and q

    while (current !== null && i < q - p + 1) {
      next = current.next
      current.next = previous
      previous = current
      current = next
      i++
```

```
    }

    //connect with the first part
    if(lastNodeOfFirstPart !== null) {
      //previous is now the first node of the sub list
      lastNodeOfFirstPart.next = previous
      //this means p === 1 i.e., we are changing
      //the first node(head) of the LL
    } else {
      head = previous
    }

    //connect with the last part
    lastNodeOfSubList.next = current
    return head
  }

  head = new Node(1)
  head.next = new Node(2);
  head.next.next = new Node(3);
  head.next.next.next = new Node(4);
  head.next.next.next.next = new Node(5);

  console.log(`Nodes of original LinkedList are: ${head.getList()}`)
  console.log(`Nodes of reversed LinkedList are: ${reverseSubList(head, 2, 4).getList()}
```

- The time complexity of our algorithm will be `O(N)` where `N` is the total number of nodes in the LinkedList.
- We only used constant space, therefore, the space complexity of our algorithm is `O(1)`.

> 🌟 Reverse the first `k` elements of a given LinkedList.

This problem can be easily converted to our parent problem; to reverse the first `k` nodes of the list, we need to pass `p=1` and `q=k`.

> 🌟 Given a LinkedList with `n` nodes, reverse it based on its size in the following way:
>
> 1. If `n` is even, reverse the list in a group of `n/2` nodes.
> 2. If `n` is odd, keep the middle node as it is, reverse the first `n/2` nodes and reverse the last `n/2` nodes.

When `n` is even we can perform the following steps:

1. Reverse first `n/2` nodes: `head = reverse(head, 1, n/2)`
2. Reverse last `n/2` nodes: `head = reverse(head, n/2 + 1, n)`

When `n` is odd, our algorithm will look like:

1. `head = reverse(head, 1, n/2)`

2. `head = reverse(head, n/2 + 2, n)` Please note the function call in the second step. We're skipping two elements as we will be skipping the middle element.

Reverse every K-element Sub-list (medium)

https://leetcode.com/problems/reverse-nodes-in-k-group/

> Given the head of a LinkedList and a number 'k', **reverse every 'k' sized sub-list** starting from the head. If, in the end, you are left with a sub-list with less than 'k' elements, reverse it too.

The problem follows the **In-place Reversal of a LinkedList** pattern and is quite similar to **Reverse a Sub-list**. The only difference is that we have to reverse all the sub-lists. We can use the same approach, starting with the first sub-list (i.e. `p=1, q=k`) and keep reversing all the sublists of size 'k'.

```
class Node {
  constructor(value, next=null) {
    this.value = value
    this.next = next
  }

  getList() {
    let result = ""
    let temp = this
    while(temp !== null) {
      result += temp.value + " "
      temp = temp.next
    }
    return result
  }
}

function reverseEveryKElements(head, k) {
  //edge cases
  if(k <= 1 || head === null) {
    return head
  }

  let current = head
  let previous = null

  while(true) {
    const lastNodeOfPreviousPart = previous

    //after reversing the LL current will
    //become the last node of the sublist
    const lastNodeOfSubList = current

    //will be used to temporaily store the next node
    let next = null
```

```
    let i = 0;

    //reverse k nodes
    while(current !== null && i < k) {
      next = current.next
      current.next = previous
      previous = current
      current = next
      i++
    }

    //connect with the previous part
    if(lastNodeOfPreviousPart !== null) {
      lastNodeOfPreviousPart.next = previous
    } else {
      head = previous
    }

    //connect with the next part
    lastNodeOfSubList.next = current

    if(current === null) {
      break
    }
    previous = lastNodeOfSubList
  }
  return head
}

head = new Node(1)
head.next = new Node(2)
head.next.next = new Node(3)
head.next.next.next = new Node(4)
head.next.next.next.next = new Node(5)
head.next.next.next.next.next = new Node(6)
head.next.next.next.next.next.next = new Node(7)
head.next.next.next.next.next.next.next = new Node(8)

console.log(`Nodes of original LinkedList are: ${head.getList()}`)
console.log(`Nodes of reversed LinkedList are: ${reverseEveryKElements(head, 3).getLis
```

- The time complexity of our algorithm will be `O(N)` where `N` is the total number of nodes in the LinkedList.
- We only used constant space, therefore, the space complexity of our algorithm is `O(1)`.

🌟 Reverse alternating K-element Sub-list (medium)

> Given the head of a LinkedList and a number `'k'`, **reverse every alternating `'k'` sized sub-list** starting from the head.

> If, in the end, you are left with a sub-list with less than `'k'` elements, reverse it too.

The problem follows the **In-place Reversal of a LinkedList** pattern and is quite similar to **Reverse every K-element Sub-list**. The only difference is that we have to skip `'k'` alternating elements. We can follow a similar approach, and in each iteration after reversing `'k'` elements, we will skip the next `'k'` elements.

```
    constructor(value, next = null) {
      this.value = value
      this.next = next
  }

    printList() {
      let temp = this
      while(temp !== null) {
        process.stdout.write(`${temp.value} `);
        temp = temp.next
      }
      console.log()
    }
  }

  function reverseAlternateKElements(head, k) {
    if(head === null || k <= 1) return head

    let current = head
    let previous = null

    while (current !== null) {
      //break if we've reached the end of the list
      const lastNodeOfPreviousPart = previous

      //after reversing the LinkedList current will become the last node of the sub-list
      const lastNodeOfSubList = current

      //will be used to temporarily store the next node
      let next = null

      //reverse k nodes
      let i = 0
      while(current !== null && i < k) {
        next = current.next
        current.next = previous
        previous = current
        current = next
        i++
      }

      //connect with the previous part
      if(lastNodeOfPreviousPart !== null) {
        lastNodeOfPreviousPart.next = previous
      } else {
        head = previous
```

```
      }

      //connect with the next part
      lastNodeOfSubList.next = current


      //skip k nodes
      i = 0
      while (current !== null && i < k){
        previous = current
        current = current.next
        i++
       }
    }
    return head
};



let head = new Node(1);
head.next = new Node(2);
head.next.next = new Node(3);
head.next.next.next = new Node(4);
head.next.next.next.next = new Node(5);
head.next.next.next.next.next = new Node(6);
head.next.next.next.next.next.next = new Node(7);
head.next.next.next.next.next.next.next = new Node(8);

process.stdout.write('Nodes of original LinkedList are: ');
head.printList();
result =  reverseAlternateKElements(head, 2);
process.stdout.write('Nodes of reversed LinkedList are: ');
result.printList();
```

- The time complexity of our algorithm will be `O(N)` where `'N'` is the total number of nodes in the LinkedList.
- We only used constant space, therefore, the space complexity of our algorithm is `O(1)`.

🌟 Rotate a LinkedList (medium)

https://leetcode.com/problems/rotate-list/

> Given the head of a Singly LinkedList and a number `'k'`, rotate the LinkedList to the right by `'k'` nodes.

Another way of defining the rotation is to take the sub-list of `'k'` ending nodes of the LinkedList and connect them to the beginning. Other than that we have to do three more things:

1. Connect the last node of the LinkedList to the head, because the list will have a different tail after the rotation.

2. The new head of the LinkedList will be the node at the beginning of the sublist.
3. The node right before the start of sub-list will be the new tail of the rotated LinkedList.

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

  getList() {
    let result = "";
    let temp = this;
    while (temp !== null) {
      result += temp.value + " ";
      temp = temp.next;
    }
    return result;
  }
};


function rotate(head, rotations) {
  if(head === null || head.next === null || rotations <= 0) return head

  //find the length and the last node of the list
  let lastNode = head
  let listLength = 1

  while(lastNode.next !== null) {
    lastNode = lastNode.next
    listLength++
  }

  //connect the last node with the head to make it a circular list
  lastNode.next = head

  //no need to do rotations more than the length of the list
  rotations %= listLength
  let skipLength = listLength - rotations
  let lastNodeOfRotatedList = head

  for(let i = 0; i < skipLength - 1; i++) {
    lastNodeOfRotatedList = lastNodeOfRotatedList.next
  }

  //lastNodeOfRotatedList.next is pointing to the sub-list of k ending nodes
  head = lastNodeOfRotatedList.next
  lastNodeOfRotatedList.next = null

  return head
};
```

```
  head = new Node(1)
  head.next = new Node(2)
  head.next.next = new Node(3)
  head.next.next.next = new Node(4)
  head.next.next.next.next = new Node(5)
  head.next.next.next.next.next = new Node(6)

  console.log(`Nodes of original LinkedList are: ${head.getList()}`)
  console.log(`Nodes of rotated LinkedList are: ${rotate(head, 3).getList()}`)
```

- The time complexity of our algorithm will be `O(N)` where `'N'` is the total number of nodes in the LinkedList.
- We only used constant space, therefore, the space complexity of our algorithm is `O(1)`.

Pattern 7: Tree Breadth First Search

This pattern is based on the **Breadth First Search (BFS)** technique to traverse a tree.

Any problem involving the traversal of a tree in a level-by-level order can be efficiently solved using this approach. We will use a **Queue** to keep track of all the nodes of a level before we jump onto the next level. This also means that the space complexity of the algorithm will be `O(W)`, where `W` is the maximum number of nodes on any level.

Binary Tree Level Order Traversal (easy) 😕

https://leetcode.com/problems/binary-tree-level-order-traversal/

> Given a binary tree, populate an array to represent its level-by-level traversal. You should populate the values of all **nodes of each level from left to right** in separate sub-arrays.

Since we need to traverse all nodes of each level before moving onto the next level, we can use the **Breadth First Search (BFS)** technique to solve this problem.

We can use a Queue to efficiently traverse in BFS fashion. Here are the steps of our algorithm:

1. Start by pushing the `root` node to the queue.
2. Keep iterating until the queue is empty.
3. In each iteration, first count the elements in the queue (let's call it `levelSize`). We will have these many nodes in the current level.
4. Next, remove `levelSize` nodes from the queue and push their `value` in an array to represent the current level.
5. After removing each node from the queue, insert both of its children into the queue.
6. If the queue is not empty, repeat from step 3 for the next level.

```
class Deque {
    constructor() {
        this.front = this.back = undefined;
    }
    addFront(value) {
        if (!this.front) this.front = this.back = { value };
        else this.front = this.front.next = { value, prev: this.front };
    }
    removeFront() {
        let value = this.peekFront();
        if (this.front === this.back) this.front = this.back = undefined;
        else (this.front = this.front.prev).next = undefined;
        return value;
    }
    peekFront() {
```

```
        return this.front && this.front.value;
    }
    addBack(value) {
        if (!this.front) this.front = this.back = { value };
        else this.back = this.back.prev = { value, next: this.back };
    }
    removeBack() {
        let value = this.peekBack();
        if (this.front === this.back) this.front = this.back = undefined;
        else (this.back = this.back.next).back = undefined;
        return value;
    }
    peekBack() {
        return this.back && this.back.value;
    }
}

class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
};


function traverse (root) {
  result = [];
  if(root === null ) {
    return result
  }

  const queue = new Deque()
  //Start by pushing the root node to the queue.
  queue.addFront(root)
  //Keep iterating until the queue is empty.
  let currentLevel = []
  while (queue.length > 0) {
    const levelSize = queue.length
    //In each iteration, first count the elements in the queue (let's call it levelSiz

    for(i = 0; i < levelSize; i++) {
      TreeNode = queue.removeFront()
      //add the node to the current level
      currentLevel.push(TreeNode.val)
      //insert the children of current node in the queue
      if(TreeNode.left !== null) {
        queue.addBack(TreeNode.left)
      }
    }
    if(TreeNode.right !== null) {
      queue.addBack(TreeNode.right)
    }
  }
```

```
    result.push(currentLevel)

    //Next, remove levelSize nodes from the queue and push their value in an array to re
    //After removing each node from the queue, insert both of its children into the queu
    //If the queue is not empty, repeat from step 3 for the next level.
    return result;
};



var root = new TreeNode(12);
root.left = new TreeNode(7);
root.right = new TreeNode(1);
root.left.left = new TreeNode(9);
root.right.left = new TreeNode(10);
root.right.right = new TreeNode(5);
console.log(`Level order traversal: ${traverse(root)}`);
```

- The time complexity of the above algorithm is `O(N)`, where `N` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` as we need to return a list containing the level order traversal. We will also need `O(N)` space for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue.

Easier to understand solution w/o `Deque()`

```
function TreeNode(val, left, right) {
  this.val = (val===undefined ? 0 : val)
  this.left = (left===undefined ? null : left)
  this.right = (right===undefined ? null : right)
 }

 const levelOrder = function  (root) {
  //If root is null return an empty array
  if(!root) return []

  const queue = [root] //initialize the queue with root
  const levels = [] //declare output array

  while(queue.length !== 0) {
    const queueLength = queue.length//get the length prior to deque
    const currLevel = []//declare this level
    //loop through to exahuast all options and only to include nodes at currLevel
    for(let i = 0; i < queueLength; i++) {
      //get next node
      const current = queue.shift()
      if(current.left) {
        queue.push(current.left)
      }
      if(current.right) {
```

```
          queue.push(current.right)
        }
        //after we add left and right for current, we add to currLevel
        currLevel.push(current.val)
      }
      //Level has been finished. Push into output array
      levels.push(currLevel)
    }
    return levels
};

levelOrder([3,9,20,null,null,15,7])//[[3],[9,20],[15,7]]
levelOrder([1])//[[1]]
levelOrder([])//[]
```

Reverse Level Order Traversal (easy)

https://leetcode.com/problems/binary-tree-level-order-traversal-ii/

> Given a binary tree, populate an array to represent its level-by-level traversal in reverse order, i.e., **the lowest level comes first**. You should populate the values of all nodes in each level from left to right in separate sub-arrays.

This problem follows the **Binary Tree Level Order Traversal** pattern. We can follow the same **BFS** approach. The only difference will be that instead of appending the current level at the end, we will append the current level at the beginning of the result list.

```
function TreeNode(val, left, right) {
  this.val = (val===undefined ? 0 : val)
  this.left = (left===undefined ? null : left)
  this.right = (right===undefined ? null : right)
 }

 const traverse = function(root) {
  if(!root) return []
   const queue = [root]

   const levels = []

   while(queue.length !== 0) {
     const queueLength = queue.length
     const currLevel = []
     for (let i = 0; i < queueLength; i++) {
       const current = queue.push()
       if(current.left) {
         queue.push(current.left)
       }
       if(current.right) {
         queue.push(current.right)
       }
```

```
        currLevel.push(current.val)
      }
      levels.unshift(currLevel)
    }

    return levels
}

 traverse([[12], [7,1], [9, 10, null, 5]])
```

- The time complexity of the above algorithm is `O(N)`, where `N` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` as we need to return a list containing the level order traversal. We will also need `O(N)` space for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue.

Zigzag Traversal (medium) 🌴

https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/

> Given a binary tree, populate an array to represent its zigzag level order traversal. You should populate the values of all **nodes of the first level from left to right**, then **right to left for the next level** and keep alternating in the same manner for the following levels.

This problem follows the **Binary Tree Level Order Traversal** pattern. We can follow the same **BFS** approach. The only additional step we have to keep in mind is to alternate the level order traversal, which means that for every other level, we will traverse similar to **Reverse Level Order Traversal**.

```
function TreeNode(val, left, right) {
  this.val = (val === undefined ? 0 : val)
  this.left = (left === undefined ? null : left)
  this.right = (right === undefined ? nill : right)
}

function zigzagLevelOrder(root) {
  //if root is null return an empty array
  if(!root) return []

  //initialize the queue with root
  const queue = [root]
  //declare the output array
  const levels = []
  let leftToRight = true

  while(queue.length !== 0) {
    //get the length prior to deque?
    const queueLength = queue.length
    //declare the current level
```

```
    const currentLevel = []

    //loop through to exhaust all aoption and only to include nodes at current Level
    for (let i = 0; i < queueLength; i++) {
      //get the next node
      const currentNode = queue.shift()

      //add the node to the current level based on the traverse direction
      if(leftToRight) {
        currentLevel.push(currentNode.val)
      } else {
        currentLevel.unshift(currentNode.val)
      }

      //insert the children of current node in the queue
      if(currentNode.left !== null) {
        queue.push(currentNode.left)
      }
      if(currentNode.right !== null) {
        queue.push(currentNode.right)
      }
    }
    //Level has been finished. push to the out put arra
    levels.push(currentLevel)

    //reverse the traversal direction
    leftToRight = !leftToRight
  }
  return levels
}

zigzagLevelOrder([1, 2, 3, 4, 5, 6, 7])
zigzagLevelOrder([3,9,20,null,null,15,7])
zigzagLevelOrder([1])
zigzagLevelOrder([])
```

- The time complexity of the above algorithm is `O(N)`, where `N` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` as we need to return a list containing the level order traversal. We will also need `O(N)` space for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue.

Level Averages in a Binary Tree (easy)

https://leetcode.com/problems/average-of-levels-in-binary-tree/

> Given a binary tree, populate an array to represent the **averages of all of its levels**

This problem follows the **Binary Tree Level Order Traversal** pattern. We can follow the same **BFS** approach. The only difference will be that instead of keeping track of all nodes of a level, we will only track the running sum of the values of all nodes in each level. In the end, we will append the average of the current level to the result array.

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

function findLevelAverages(root) {
  let result = []

  //edge case => no root
  if(!root) {
    return result
  }

  const queue = [root]

  while(queue.length > 0) {
    let levelSize = queue.length
    let levelSum = 0.0

    for(let i = 0; i < levelSize; i++){
      let currentNode = queue.shift()

      //add the node's value to the running sum
      levelSum += currentNode.value

      //insert the children of the current node to the queue
      if(currentNode.left !== null) {
        queue.push(currentNode.left)
      }

      if(currentNode.right !== null) {
        queue.push(currentNode.right)
      }
    }

    //append the current level's average to the result array
    result.push(levelSum/levelSize)
  }
  return result
}

var root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.left.left = new TreeNode(9)
```

```
root.left.right = new TreeNode(2)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)

console.log(`Level averages are: ${findLevelAverages(root)}`)
```

- The time complexity of the above algorithm is `O(N)`, where `N` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` which is required for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue

Level Maximum in a Binary Tree

https://leetcode.com/problems/maximum-level-sum-of-a-binary-tree/

> 🌟 Find the largest value on each level of a binary tree.

We will follow a similar approach, but instead of having a running sum we will track the maximum value of each level.

```
maxValue = Math.max(maxValue, currentNode.val)
```

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

function largestValue(root) {
  let result = []

  //edge case => no root
  if(!root) {
    return result
  }

  const queue = [root]

  while(queue.length > 0) {
    let levelSize = queue.length
    let maxValue = 0

    for(let i = 0; i < levelSize; i++){

      let currentNode = queue.shift()

      maxValue = Math.max(maxValue, currentNode.value)
```

```
      //insert the children of the current node to the queue
      if(currentNode.left !== null) {
        queue.push(currentNode.left)
      }

      if(currentNode.right !== null) {
        queue.push(currentNode.right)
      }
    }

    //append the current level's max value to the result array
    result.push(maxValue)
    maxValue = 0
  }
  return result
}

const root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.left.left = new TreeNode(9)
root.left.right = new TreeNode(2)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)

console.log(`Max value's for each level are: ${largestValue(root)}`)
```

Minimum Depth of a Binary Tree (easy)

https://leetcode.com/problems/minimum-depth-of-binary-tree/

> Find the minimum depth of a binary tree. The minimum depth is the number of nodes along the **shortest path from the root node to the nearest leaf node**.

This problem follows the **Binary Tree Level Order Traversal** pattern. We can follow the same **BFS** approach. The only difference will be, instead of keeping track of all the nodes in a level, we will only track the depth of the tree. As soon as we find our first leaf node, that level will represent the minimum depth of the tree.

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

function findMinimumDepth(root) {
  //edge case => no root
  if(!root) {
```

```
    return 0
  }

  const queue = [root]

  let minimumTreeDepth = 0
  while(queue.length > 0) {
    minimumTreeDepth++
    let levelSize = queue.length

    for(let i = 0; i < levelSize; i++) {
      let currentNode = queue.shift()

      //check if this is a leaf node
      if(currentNode.left === null && currentNode.right === null) {
        return minimumTreeDepth
      }

      //insert the children of current node in the queue
      if(currentNode.left !== null) {
        queue.push(currentNode.left)
      }
      if(currentNode.right !== null) {
        queue.push(currentNode.right)
      }
    }
  }
}

const root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
console.log(`Tree Minimum Depth: ${findMinimumDepth(root)}`)
root.left.left = new TreeNode(9)
root.right.left.left = new TreeNode(11)
console.log(`Tree Minimum Depth: ${findMinimumDepth(root)}`)
```

- The time complexity of the above algorithm is `O(N)`, where `N` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` which is required for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue.

Maximum Depth of a Binary Tree

https://leetcode.com/problems/maximum-depth-of-binary-tree/

> Given a binary tree, find its maximum depth (or height).

We will follow a similar approach. Instead of returning as soon as we find a leaf node, we will keep traversing for all the levels, incrementing `maximumDepth` each time we complete a level.

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

function findMaximumDepth(root) {
  //edge case => no root
  if(!root) {
    return 0
  }

  const queue = [root]

  let maximumTreeDepth = 0

  while(queue.length > 0) {
    maximumTreeDepth++
    const levelSize = queue.length

    for(let i = 0; i < levelSize; i++) {
      let currentNode = queue.shift()

      //insert the children of current node in the queue
      if(currentNode.left !== null) {
        queue.push(currentNode.left)
      }
      if(currentNode.right !== null) {
        queue.push(currentNode.right)
      }
    }
  }
  return maximumTreeDepth
}

const root = new TreeNode(12);
root.left = new TreeNode(7);
root.right = new TreeNode(1);
root.right.left = new TreeNode(10);
root.right.right = new TreeNode(5);
console.log(`Tree Maximum Depth: ${findMaximumDepth(root)}`);
root.left.left = new TreeNode(9);
root.right.left.left = new TreeNode(11);
console.log(`Tree Maximum Depth: ${findMaximumDepth(root)}`);
```

Level Order Successor (easy) 😕

> Given a binary tree and a node, find the level order successor of the given node in the tree. The level order successor is the node that appears right after the given node in the level order traversal.

This problem follows the **Binary Tree Level Order Traversal** pattern. We can follow the same **BFS** approach. The only difference will be that we will not keep track of all the levels. Instead we will keep inserting child nodes to the queue. As soon as we find the given node, we will return the next node from the queue as the level order successor.

```
class Deque {
    constructor() {
        this.front = this.back = undefined;
    }
    addFront(value) {
        if (!this.front) this.front = this.back = { value };
        else this.front = this.front.next = { value, prev: this.front };
    }
    removeFront() {
        let value = this.peekFront();
        if (this.front === this.back) this.front = this.back = undefined;
        else (this.front = this.front.prev).next = undefined;
        return value;
    }
    peekFront() {
        return this.front && this.front.value;
    }
    addBack(value) {
        if (!this.front) this.front = this.back = { value };
        else this.back = this.back.prev = { value, next: this.back };
    }
    removeBack() {
        let value = this.peekBack();
        if (this.front === this.back) this.front = this.back = undefined;
        else (this.back = this.back.next).back = undefined;
        return value;
    }
    peekBack() {
        return this.back && this.back.value;
    }
}

class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

function findSuccessor(root, key) {
  //edge case => no root
  // if(!root) {
```

```
    //   return null
    // }
   if (root === null) {
     return null;
    }

    const queue = new Deque();
    queue.addFront(root);
    while (queue.length > 0) {
      currentNode = queue.shift();
      // insert the children of current node in the queue
      if (currentNode.left !== null) {
        queue.push(currentNode.left);
      }
      if (currentNode.right !== null) {
        queue.push(currentNode.right);
      }
      // break if we have found the key
      if (currentNode.val === key) {
        break;
      }
    }

    if (queue.length > 0) {
      return queue.peek();
    }
    return null;
  }

  var root = new TreeNode(12)
  root.left = new TreeNode(7)
  root.right = new TreeNode(1)
  root.left.left = new TreeNode(9)
  root.right.left = new TreeNode(10)
  root.right.right = new TreeNode(5)
  result = findSuccessor(root, 12)
  if (result != null)
    console.log(result.val)
  result = findSuccessor(root, 9)
  if (result != null)
    console.log(result.val)
```

- The time complexity of the above algorithm is `O(N)`, where `N` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` which is required for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue.

😕 Connect Level Order Siblings (medium)

https://leetcode.com/problems/populating-next-right-pointers-in-each-node/

> Given a binary tree, connect each node with its level order successor. The last node of each level should point to a `null` node.

This problem follows the **Binary Tree Level Order Traversal** pattern. We can follow the same **BFS** approach. The only difference is that while traversing a level we will remember the previous node to connect it with the current node.

```
class TreeNode {
  constructor(val) {
    this.val = val
    this.left = null
    this.right = null
    this.next = null
  }
}

  // level order traversal using 'next' pointer
 function printLevelOrder() {
    console.log("Level order traversal using 'next' pointer: ");
    let nextLevelRoot = this;
    while (nextLevelRoot !== null) {
      let currentNode = nextLevelRoot;
      nextLevelRoot = null;
      while (currentNode != null) {
        process.stdout.write(`${currentNode.val} `);
        if (nextLevelRoot === null) {
          if (currentNode.left !== null) {
            nextLevelRoot = currentNode.left;
          } else if (currentNode.right !== null) {
            nextLevelRoot = currentNode.right;
          }
        }
        currentNode = currentNode.next;
      }
      console.log();
    }
  }


function connectLevelOrderSiblings(root) {
  //if root is null return an empty array
  if(!root) return []

  //initilize the queue with root
  const queue = [root]

  // //declare output array
  // const levels = []

  while(queue.length > 0) {
    let previousNode = null

    //get length prior to dequeue
```

```
      const levelSize = queue.length

      // //declare this level
      // const currLevel = []

      //connect all nodes of this level
      for(let i = 0; i < levelSize; i++) {
        //get the next node
        const currentNode = queue.shift()
        if(previousNode !== null) {
          previousNode.next = currentNode
        }
        previousNode = currentNode

        //insert the children of currentNode in the queue
        if(currentNode.left !== null) {
          queue.push(currentNode.left)
        }
        if(currentNode.right !== null) {
          queue.push(currentNode.right)
        }

      //   //after we add left and right for current, we add to currLevel
      //   currLevel.push(current.val)
      }

      // //level has been finished. Push into output array
      // levels.push(currLevel)
    }
    // return levels
}

const root = new TreeNode(12);
root.left = new TreeNode(7);
root.right = new TreeNode(1);
root.left.left = new TreeNode(9);
root.right.left = new TreeNode(10);
root.right.right = new TreeNode(5);
connectLevelOrderSiblings(root);

printLevelOrder(root)
```

- The time complexity of the above algorithm is `O(N)`, where `'N'` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)`, which is required for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue.

🌟 Connect All Level Order Siblings (medium)

> Given a binary tree, connect each node with its level order successor. The last node of each level should point to the first node of the next level.

This problem follows the **Binary Tree Level Order Traversal** pattern. We can follow the same **BFS** approach. The only difference will be that while traversing we will remember (irrespective of the level) the previous node to connect it with the current node.

```
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }

  // tree traversal using 'next' pointer
  printTree() {
    let result = "Traversal using 'next' pointer: ";
    let current = this;
    while (current != null) {
      result += current.value + " ";
      current = current.next;
    }
    console.log(result);
  }
};

function connectAllSiblings(root) {
  if(root === null) {
    return
  }

  const queue = [root]
  let currentNode = null
  let previousNode = null

  while(queue.length > 0) {
    currentNode = queue.shift()

    if(previousNode !== null) {
      previousNode.next = currentNode
    }

    previousNode = currentNode

    //insert the children of the currentNode into the queue
    if(currentNode.left !== null) {
      queue.push(currentNode.left)
    }
    if(currentNode.right !== null) {
      queue.push(currentNode.right)
    }
  }
};
```

```
const root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.left.left = new TreeNode(9)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
connectAllSiblings(root)
root.printTree()
```

- The time complexity of the above algorithm is `O(N)`, where `'N'` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` which is required for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue.

🌟 Right View of a Binary Tree (easy)

https://leetcode.com/problems/binary-tree-right-side-view/

> Given a binary tree, return an array containing nodes in its right view. The right view of a binary tree is the set of **nodes visible when the tree is seen from the right side**.

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

function treeRightView(root) {
  let result = [];

  if(root === null) {
    return result
  }

  const queue = [root]

  while(queue.length > 0) {
    let levelSize = queue.length

    for(let i = 0; i < levelSize; i++) {
      let currentNode = queue.shift()

      //if it is the last node of this level,
      //add it to the result
      if(i === levelSize - 1){
```

```
          result.push(currentNode.value)
        }
        //insert the children of current node in the queue
        if(currentNode.left !== null) {
          queue.push(currentNode.left)
        }
        if(currentNode.right !== null) {
          queue.push(currentNode.right)
        }
      }
    }

    return result;
  };

  const root = new TreeNode(12);
  root.left = new TreeNode(7);
  root.right = new TreeNode(1);
  root.left.left = new TreeNode(9);
  root.right.left = new TreeNode(10);
  root.right.right = new TreeNode(5);
  root.left.left.left = new TreeNode(3);
  console.log("Tree right view: " + treeRightView(root))
```

- The time complexity of the above algorithm is `O(N)`, where `'N'` is the total number of nodes in the tree. This is due to the fact that we traverse each node once
- The space complexity of the above algorithm will be `O(N)` as we need to return a list containing the level order traversal. We will also need `O(N)` space for the queue. Since we can have a maximum of `N/2` nodes at any level (this could happen only at the lowest level), therefore we will need `O(N)` space to store them in the queue.

Similar Questions

> Given a binary tree, return an array containing nodes in its left view. The left view of a binary tree is the set of nodes visible when the tree is seen from the left side.

We will be following a similar approach, but instead of appending the last element of each level, we will be appending the first element of each level to the output array.

```
  class TreeNode {
    constructor(value) {
      this.value = value
      this.left = null
      this.right = null
    }
  }

  function treeRightView(root) {
    let result = [];
```

```
    if(root === null) {
      return result
    }

    const queue = [root]

    while(queue.length > 0) {
      let levelSize = queue.length

      for(let i = 0; i < levelSize; i++) {
        let currentNode = queue.shift()

        //if it is the first node of this level,
        //add it to the result
        if(i === 0){
          result.push(currentNode.value)
        }
        //insert the children of current node in the queue
        if(currentNode.left !== null) {
          queue.push(currentNode.left)
        }
        if(currentNode.right !== null) {
          queue.push(currentNode.right)
        }
      }
    }

    return result;
  };

  const root = new TreeNode(12);
  root.left = new TreeNode(7);
  root.right = new TreeNode(1);
  root.left.left = new TreeNode(9);
  root.right.left = new TreeNode(10);
  root.right.right = new TreeNode(5);
  root.left.left.left = new TreeNode(3);
  console.log("Tree right view: " + treeRightView(root))
```

Pattern 8: Tree Depth First Search (DFS)

This pattern is based on the **Depth First Search (DFS)** technique to traverse a tree.

We will be using recursion (or we can also use a stack for the iterative approach) to keep track of all the previous (parent) nodes while traversing. This also means that the space complexity of the algorithm will be `O(H)`, where `'H'` is the maximum height of the tree.

Binary Tree Path Sum (easy)

https://leetcode.com/problems/path-sum/

> Given a binary tree and a number `'S'`, find if the tree has a path from root-to-leaf such that the sum of all the node values of that path equals `'S'`.

As we are trying to search for a root-to-leaf path, we can use the **Depth First Search (DFS)** technique to solve this problem.

To recursively traverse a binary tree in a DFS fashion, we can start from the root and at every step, make two recursive calls one for the left and one for the right child.

Here are the steps for our Binary Tree Path Sum problem:

1. Start DFS with the root of the tree.
2. If the current node is not a leaf node, do two things:
    - Subtract the value of the current node from the given number to get a new sum => `S = S - node.value`
    - Make two recursive calls for both the children of the current node with the new number calculated in the previous step.
3. At every step, see if the current node being visited is a leaf node and if its value is equal to the given number `'S'`. If both these conditions are true, we have found the required root-to-leaf path, therefore return `true`.
4. If the current node is a leaf but its value is not equal to the given number `'S'`, return `false`.

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

function hasPath(root, sum) {
  if(!root) {
```

```
    return false
  }

  //start DFS with the root of the tree
  //if the current node is a leaf and it's value is
  //equal to the sum then we've found a path
  if(root.value === sum && root.left === null && root.right === null) {
    return true
  }

  //recursively call to traverse the left and right sub-tree
  //return true if any of the two recursive calls return true
  return hasPath(root.left, sum - root.value) || hasPath(root.right, sum - root.value)
}

var root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.left.left = new TreeNode(9)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
console.log(`Tree has path: ${has_path(root, 23)}`)
console.log(`Tree has path: ${has_path(root, 16)}`)
```

- The time complexity of the above algorithm is `O(N)`, where `'N'` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

All Paths for a Sum (medium)

https://leetcode.com/problems/path-sum-ii/

> Given a binary tree and a number `'S'`, find all paths from root-to-leaf such that the sum of all the node values of each path equals `'S'`.

This problem follows the **Binary Tree Path Sum** pattern. We can follow the same **DFS** approach. There will be two differences:

1. Every time we find a root-to-leaf path, we will store it in a list.
2. We will traverse all paths and will not stop processing after finding the first path.

```
class TreeNode {
  constructor(value, left = null, right = null) {
    this.value = value;
    this.left = left;
    this.right = right;
  }
```

```
};


function findPaths(root, sum) {
  let allPaths = []

  findAPath(root, sum, [], allPaths)

  return allPaths
};

function findAPath(currentNode, sum, currentPath, allPaths) {
  if(currentNode === null) {
    return
  }

  //add the current node to the path
  currentPath.push(currentNode.value)

  //if the current node is a leaf and it's value is
  //equal to sum, save the current path
  if(currentNode.value === sum && currentNode.left === null && currentNode.right === n
    allPaths.push([...currentPath])
  } else {
    //traverse the left sub-tree
    findAPath(currentNode.left, sum - currentNode.value, currentPath, allPaths)
    //traverse the right sub-tree
    findAPath(currentNode.right, sum - currentNode.value, currentPath, allPaths)
  }

  //remove the current node for the path to backtrack,
  //we need to remove the currentNode while we are going
  //up the recursive call stack
  currentPath.pop()
}

const root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.left.left = new TreeNode(4)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
findPaths(root, 23);
```

- The time complexity of the above algorithm is `O(N^2)`, where `'N'` is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take `O(N)`), and for every leaf node, we might have to store its path (by making a copy of the current path) which will take `O(N)`.
  - We can calculate a tighter time complexity of `O(NlogN)` from the space complexity discussion below.
- If we ignore the space required for the `allPaths` list, the space complexity of the above algorithm will be `O(N)` in the worst case. This space will be used to store the recursion stack.

The worst-case will happen when the given tree is a linked list (i.e., every node has only one child).

> 🌟 Given a binary tree, return all root-to-leaf paths.

https://leetcode.com/problems/binary-tree-paths/

We can follow a similar approach. We just need to remove the "check for the path sum."

```
class TreeNode {
  constructor(value, left = null, right = null) {
    this.value = value;
    this.left = left;
    this.right = right;
  }
};


function findPaths(root) {
  let allPaths = []

  findAPath(root, [], allPaths)

  return allPaths
};

function findAPath(currentNode, currentPath, allPaths) {
  if(currentNode === null) {
    return
  }

  //add the current node to the path
  currentPath.push(currentNode.value)

  //if the current node is not a leaf, save the current path
  if(currentNode.left === null && currentNode.right === null) {
    allPaths.push([...currentPath])
  } else {
    //traverse the left sub-tree
    findAPath(currentNode.left, currentPath, allPaths)
    //traverse the right sub-tree
    findAPath(currentNode.right, currentPath, allPaths)
  }

  //remove the current node from the path to backtrack,
  //we need to remove the currentNode while we are going
  //up the recursive call stack
  currentPath.pop()
}

const root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
```

```
root.left.left = new TreeNode(4)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
findPaths(root);
```

> 🌟 Given a binary tree, find the root-to-leaf path with the maximum sum.

We need to find the path with the maximum sum. As we traverse all paths, we can keep track of the path with the maximum sum.

```
class TreeNode {
  constructor(value, left = null, right = null) {
    this.value = value;
    this.left = left;
    this.right = right;
  }
};


function maxPathSum(root) {
  let maxSum = -Infinity

  function findAPath(currentNode, currentPath) {
    if(currentNode === null) {
      return
    }
    //add the current node to the path
    currentPath.push(currentNode.value)

    let pathMax = 0
    //if the current node is not a leaf, save the current path
    if(currentNode.left === null && currentNode.right === null) {
      for(let i = 0; i < currentPath.length; i++) {
        pathMax += currentPath[i]
      }
      maxSum = Math.max(maxSum, pathMax)
    } else {
      //traverse the left sub-tree
      findAPath(currentNode.left, currentPath)
      //traverse the right sub-tree
      findAPath(currentNode.right, currentPath)
    }

    //remove the current node from the path to backtrack,
    //we need to remove the currentNode while we are going
    //up the recursive call stack
    currentPath.pop()
  }

  findAPath(root, [], maxSum)

  return maxSum
};
```

```
const root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.left.left = new TreeNode(4)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
maxPathSum(root);
```

Sum of Path Numbers (medium)

https://leetcode.com/problems/sum-root-to-leaf-numbers/

> Given a binary tree where each node can only have a digit (0-9) value, each root-to-leaf path will represent a number. Find the total sum of all the numbers represented by all paths.

This problem follows the **Binary Tree Path Sum** pattern. We can follow the same **DFS** approach. The additional thing we need to do is to keep track of the number representing the current path.

How do we calculate the path number for a node? Taking the first example mentioned above, say we are at node '7'. As we know, the path number for this node is '17', which was calculated by: `1 * 10 + 7 => 17` . We will follow the same approach to calculate the path number of each node.

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

function findSumOfPathNumbers(root) {
  return findRootToLeafPathNumbers(root, 0)
}

function findRootToLeafPathNumbers(currentNode, pathSum) {
    if(currentNode === null) {
      return 0
    }

    //calculate the path number of the current node
    pathSum = 10 * pathSum + currentNode.value

    //if the currentNode is a leaf, retuen the current pathSum
    if(currentNode.left === null && currentNode.right === null) {
      return pathSum
    }

    //traverse the left and the right sub-tree
```

```
    return findRootToLeafPathNumbers(currentNode.left, pathSum) + findRootToLeafPathNu
}

const root = new TreeNode(1)
root.left = new TreeNode(0)
root.right = new TreeNode(1)
root.left.left = new TreeNode(1)
root.right.left = new TreeNode(6)
root.right.right = new TreeNode(5)
console.log(`Total Sum of Path Numbers: ${findSumOfPathNumbers(root)}`)
```

- The time complexity of the above algorithm is `O(N)`, where `'N'` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

Path With Given Sequence (medium)

> Given a binary tree and a number sequence, find if the sequence is present as a root-to-leaf path in the given tree.

This problem follows the **Binary Tree Path Sum** pattern. We can follow the same **DFS** approach and additionally, track the element of the given sequence that we should match with the current node. Also, we can return false as soon as we find a mismatch between the sequence and the node value.

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.right = null
    this.left = null
  }
}

function findPath(root, sequence) {
  if(!root) {
    return sequence.length === 0
  }

  return findPathRecursive(root, sequence, 0)
}

function findPathRecursive(currentNode, sequence, sequenceIndex) {
  if(currentNode === null) return false

  const sequenceLength = sequence.length
  if(sequenceIndex >= sequenceLength || currentNode.value !== sequence[sequenceIndex])
```

```
  //if the current node is a leaf, and it is the end of the sequence, then we have fou
  if(currentNode.left === null && currentNode.right === null && sequenceIndex === sequ

  //recursively call to traverse the left and right sub-tree
  //return true if any of the two recursive calls return true

  return findPathRecursive(currentNode.left, sequence, sequenceIndex + 1) || findPathR
}

const root = new TreeNode(1)
root.left = new TreeNode(0)
root.right = new TreeNode(1)
root.left.left = new TreeNode(1)
root.right.left = new TreeNode(6)
root.right.right = new TreeNode(5)

console.log(`Tree has path sequence: ${findPath(root, [1, 0, 7])}`)
console.log(`Tree has path sequence: ${findPath(root, [1, 1, 6])}`)
```

- The time complexity of the above algorithm is `O(N)`, where `'N'` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

Count Paths for a Sum (medium)

https://leetcode.com/problems/path-sum-iii/

> Given a binary tree and a number `'S'`, find all paths in the tree such that the sum of all the node values of each path equals `'S'`. Please note that the paths can start or end at any node but all paths must follow direction from parent to child (top to bottom).

This problem follows the **Binary Tree Path Sum** pattern. We can follow the same **DFS** approach. But there will be four differences:

1. We will keep track of the current path in a list which will be passed to every recursive call.
2. Whenever we traverse a node we will do two things:
    - Add the current node to the current path.
    - As we added a new node to the current path, we should find the sums of all sub-paths ending at the current node. If the sum of any sub-path is equal to 'S' we will increment our path count.
3. We will traverse all paths and will not stop processing after finding the first path.

4. Remove the current node from the current path before returning from the function. This is needed to Backtrack while we are going up the recursive call stack to process other paths.

```
class TreeNode {
  constructor(value, right = null, left = null) {
    this.value = value
    this.right = right
    this.left = left
  }
}

function countPaths(root, S) {
  let currentPath = []
  return countPathsRecursive(root, S, currentPath)
}

function countPathsRecursive(currentNode, S, currentPath) {
  if(currentNode === null) return 0

  //add the currentNode to the path
  currentPath.push(currentNode.value)

  let pathCount = 0
  let pathSum = 0

  //find the sums of all aub-paths in the current path list
  for(let i = currentPath.length - 1; i >= 0; i--) {
    pathSum += currentPath[i]

    //if the sum of any sub-path is equal S we increment our path count
    if(pathSum === S) {
      pathCount++
    }
  }

  //traverse the left sub-tree
  pathCount += countPathsRecursive(currentNode.left, S, currentPath)
  //traverse the right sub-tree
  pathCount += countPathsRecursive(currentNode.right, S, currentPath)

  //remove the current node from the path to backtrack
  //we need to remove the current node while we are going upp the recursive call stack
  currentPath.pop()
  return pathCount
}

const root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.left.left = new TreeNode(4)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
console.log(`Tree has ${countPaths(root, 11)} paths`)
```

- The time complexity of the above algorithm is `O(N²)` in the worst case, where `'N'` is the total number of nodes in the tree. This is due to the fact that we traverse each node once, but for every node, we iterate the current path. The current path, in the worst case, can be `O(N)` (in the case of a skewed tree). But, if the tree is balanced, then the current path will be equal to the height of the tree, i.e., `O(logN)`. So the best case of our algorithm will be `O(NlogN)`.
- The space complexity of the above algorithm will be `O(N)`. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child). We also need `O(N)` space for storing the currentPath in the worst case. Overall space complexity of our algorithm is `O(N)`.

🌟 Tree Diameter (medium)

https://leetcode.com/problems/diameter-of-binary-tree/

> Given a binary tree, find the length of its diameter. The diameter of a tree is the number of nodes on the **longest path between any two leaf nodes**. The diameter of a tree may or may not pass through the root.
>
> Note: You can always assume that there are at least two leaf nodes in the given tree.

This problem follows the **Binary Tree Path Sum** pattern. We can follow the same **DFS** approach. There will be a few differences:

1. At every step, we need to find the height of both children of the current node. For this, we will make two recursive calls similar to **DFS**.
2. The height of the current node will be equal to the maximum of the heights of its left or right children, plus `1` for the `currentNode`.
3. The tree diameter at the `currentNode` will be equal to the height of the left child plus the height of the right child plus `1` for the current node: `diameter = leftTreeHeight + rightTreeHeight + 1`. To find the overall tree diameter, we will use a class level variable. This variable will store the maximum `diameter` of all the nodes visited so far, hence, eventually, it will have the final tree diameter.

```
class TreeNode {
  constructor(value, left = null, right = null) {
    this.value = value;
    this.left = left;
    this.right = right;
  }
};

class TreeDiameter {
  constructor() {
    this.treeDiameter = 0;
  }
```

```
  findDiameter(root) {
    this.calculateHeight(root)
    return this.treeDiameter
  }

   calculateHeight(currentNode) {
    if(currentNode === null) return 0

     const leftTreeHeight = this.calculateHeight(currentNode.left)
     const rightTreeHeight = this.calculateHeight(currentNode.right)

     //if the current node doesn't have a left or right sub-tree,
     //we can't have a path passing through it,
     //since we need a leaf node on each side
     if(leftTreeHeight !== 0 && rightTreeHeight !== 0) {
       //diameter at the currentNode will be equal to the height of the left
       //sub-tree the height of sub-trees + 1 for the currentNode
       const diameter = leftTreeHeight + rightTreeHeight + 1

       //update the global tree diameter
       this.treeDiameter = Math.max(this.treeDiameter, diameter)
     }

     //height of the currentNode will be equal to the maximum of the heights of
     //left or right sub-trees plus 1
     return Math.max(leftTreeHeight, rightTreeHeight) + 1
  }
};


const treeDiameter = new TreeDiameter()
const root = new TreeNode(1)
root.left = new TreeNode(2)
root.right = new TreeNode(3)
root.left.left = new TreeNode(4)
root.right.left = new TreeNode(5)
root.right.right = new TreeNode(6)
console.log(`Tree Diameter: ${treeDiameter.findDiameter(root)}`)
root.left.left = null
root.right.left.left = new TreeNode(7)
root.right.left.right = new TreeNode(8)
root.right.right.left = new TreeNode(9)
root.right.left.right.left = new TreeNode(10)
root.right.right.left.left = new TreeNode(11)
console.log(`Tree Diameter: ${treeDiameter.findDiameter(root)}`)
```

- The time complexity of the above algorithm is `O(N)`, where `N` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

🌟 Path with Maximum Sum (hard)

https://leetcode.com/problems/binary-tree-maximum-path-sum/

> Find the path with the maximum sum in a given binary tree. Write a function that returns the maximum sum.
>
> A path can be defined as a **sequence of nodes between any two nodes** and doesn't necessarily pass through the root. The path must contain at least one node.

This problem follows the **Binary Tree Path Sum** pattern and shares the algorithmic logic with **Tree Diameter**. We can follow the same **DFS** approach. The only difference will be to ignore the paths with negative sums. Since we need to find the overall maximum sum, we should ignore any path which has an overall negative sum.

```
class TreeNode {
  constructor(value, left = null, right = null) {
    this.value = value;
    this.left = left;
    this.right = right;
  }
};



function findMaximumPathSum(root) {
  let globalMaximumSum = -Infinity
  findMaximumPathSumRecursive(root)


  function findMaximumPathSumRecursive(currentNode) {
    if(currentNode === null) return 0

    let maxPathSumLeft = findMaximumPathSumRecursive(currentNode.left)
    let maxPathSumRight = findMaximumPathSumRecursive(currentNode.right)

    //ignore paths with negative sums, since we need to find the maximum sum
    //we should ignore any path which has an overall negative sum
    maxPathSumLeft = Math.max(maxPathSumLeft, 0)
    maxPathSumRight = Math.max(maxPathSumRight, 0)

    //maximum path sum at the currentNode will be equal to the sum from the
    //left subtree + the sum from the right subtree + value of the currentNode
    const localMaximumSum = maxPathSumLeft + maxPathSumRight + currentNode.value

    //update the globalMaximumSum
    globalMaximumSum = Math.max(globalMaximumSum, localMaximumSum)

    //maximum sum of any path from the currentNode will be equal to the maximum
    //of the sums from left to right sub-tree plus the value of the currentNode
    return Math.max(maxPathSumLeft, maxPathSumRight) + currentNode.value
```

```
    }
    return globalMaximumSum
};

let root = new TreeNode(1)
root.left = new TreeNode(2)
root.right = new TreeNode(3)
console.log(`Maximum Path Sum: ${findMaximumPathSum(root)}`)//6

root.left.left = new TreeNode(1)
root.left.right = new TreeNode(3)
root.right.left = new TreeNode(5)
root.right.right = new TreeNode(6)
root.right.left.left = new TreeNode(7)
root.right.left.right = new TreeNode(8)
root.right.right.left = new TreeNode(9)
console.log(`Maximum Path Sum: ${findMaximumPathSum(root)}`)//31

root = new TreeNode(-1)
root.left = new TreeNode(-3)
console.log(`Maximum Path Sum: ${findMaximumPathSum(root)}`)
```

- The time complexity of the above algorithm is `O(N)`, where `N` is the total number of nodes in the tree. This is due to the fact that we traverse each node once.
- The space complexity of the above algorithm will be `O(N)` in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

#DFS #DepthFirstSearch #JavaScript #GrokkingTheCodingInterviewPatterns #LeetCode #DataStructures #Algorithms

Pattern 9: Two Heaps

In many problems, where we are given a set of elements such that we can divide them into two parts. To solve the problem, we are interested in knowing the smallest element in one part and the biggest element in the other part. This pattern is an efficient approach to solve such problems.

This pattern uses two **Heaps** to solve these problems; A **Min Heap** to find the smallest element and a **Max Heap** to find the biggest element.

🌟 😕 Find the Median of a Number Stream (medium)

https://leetcode.com/problems/find-median-from-data-stream/

> Design a class to calculate the median of a number stream. The class should have the following two methods:
>
> 1. `insertNum(int num)` : stores the number in the class
> 2. `findMedian()` : returns the median of all numbers inserted in the class If the count of numbers inserted in the class is even, the median will be the average of the middle two numbers.

As we know, the median is the middle value in an ordered integer list. So a brute force solution could be to maintain a sorted list of all numbers inserted in the class so that we can efficiently return the median whenever required. Inserting a number in a sorted list will take `O(N)` time if there are `'N'` numbers in the list. This insertion will be similar to the **Insertion sort**. Can we do better than this? Can we utilize the fact that we don't need the fully sorted list - we are only interested in finding the middle element?

Assume 'x' is the median of a list. This means that half of the numbers in the list will be smaller than (or equal to) 'x' and half will be greater than (or equal to) 'x'. This leads us to an approach where we can divide the list into two halves: one half to store all the smaller numbers (let's call it `smallNumList`) and one half to store the larger numbers (let's call it `largNumList`). The median of all the numbers will either be the largest number in the `smallNumList` or the smallest number in the `largNumList` . If the total number of elements is even, the median will be the average of these two numbers.

The best data structure that comes to mind to find the smallest or largest number among a list of numbers is a **Heap**. Let's see how we can use a heap to find a better algorithm.

1. We can store the first half of numbers (i.e., `smallNumList`) in a **Max Heap**. We should use a Max Heap as we are interested in knowing the largest number in the first half.
2. We can store the second half of numbers (i.e., `largeNumList`) in a **Min Heap**, as we are interested in knowing the smallest number in the second half.

3. Inserting a number in a heap will take `O(logN)`, which is better than the brute force approach.
4. At any time, the median of the current list of numbers can be calculated from the top element of the two heaps.

Let's take the Example-1 mentioned above to go through each step of our algorithm:

1. `insertNum(3)`: We can insert a number in the **Max Heap** (i.e. first half) if the number is smaller than the top (largest) number of the heap. After every insertion, we will balance the number of elements in both heaps, so that they have an equal number of elements. If the count of numbers is odd, let's decide to have more numbers in max-heap than the Min Heap.
2. `insertNum(1)`: As '1' is smaller than '3', let's insert it into the **Max Heap**.

Now, we have two elements in the **Max Heap** and no elements in **Min Heap**. Let's take the largest element from the Max Heap and insert it into the **Min Heap**, to balance the number of elements in both heaps.

3. `findMedian()`: As we have an even number of elements, the median will be the average of the top element of both the heaps ➡️ `(1+3)/2 = 2.0` `(1+3)/2=2.0`
4. `insertNum(5)`: As '5' is greater than the top element of the **Max Heap**, we can insert it into the **Min Heap**. After the insertion, the total count of elements will be odd. As we had decided to have more numbers in the **Max Heap** than the **Min Heap**, we can take the top (smallest) number from the **Min Heap** and insert it into the **Max Heap**.
5. `findMedian()`: Since we have an odd number of elements, the median will be the top element of **Max Heap** ➡️ `3`. An odd number of elements also means that the **Max Heap** will have one extra element than the **Min Heap**.
6. `insertNum(4)`: Insert '4' into **Min Heap**.
7. `findMedian()`: As we have an even number of elements, the median will be the average of the top element of both the heaps ➡️ `(3+4)/2 = 3.5` `(3+4)/2=3.5`

Sliding Window Median (hard)

https://leetcode.com/problems/sliding-window-median/

> Given an array of numbers and a number 'k', find the median of all the 'k' sized sub-arrays (or windows) of the array.

Example 1:

Input: `nums=[1, 2, -1, 3, 5], k = 2`

Output: `[1.5, 0.5, 1.0, 4.0]`

Explanation:

Lets consider all windows of size '2':

```
[1, 2, -1, 3, 5] -> median is 1.5
[1, 2, -1, 3, 5] -> median is 0.5
[1, 2, -1, 3, 5] -> median is 1.0
[1, 2, -1, 3, 5] -> median is 4.0
```

Example 2:

Input: `nums=[1, 2, -1, 3, 5], k = 3`

Output: `[1.0, 2.0, 3.0]`

Explanation:

Lets consider all windows of size '3':

```
[1, 2, -1, 3, 5] -> median is 1.0
[1, 2, -1, 3, 5] -> median is 2.0
[1, 2, -1, 3, 5] -> median is 3.0
```

This problem follows the **Two Heaps** pattern and share similarities with **Find the Median of a Number Stream**. We can follow a similar approach of maintaining a **max-heap** and a **min-heap** for the list of numbers to find their median.

The only difference is that we need to keep track of a sliding window of 'k' numbers. This means, in each iteration, when we insert a new number in the heaps, we need to remove one number from the heaps which is going out of the sliding window. After the removal, we need to rebalance the heaps in the same way that we did while inserting.

Maximize Capital (hard)

https://leetcode.com/problems/ipo/

🌟 Next Interval (hard)

https://leetcode.com/problems/find-right-interval/

Pattern 10: Subsets

A huge number of coding interview problems involve dealing with **Permutations** and **Combinations** of a given set of elements. This pattern describes an efficient **Breadth First Search (BFS)** approach to handle all these problems.

Subsets (easy)

https://leetcode.com/problems/subsets/

> Given a set with distinct elements, find all of its distinct subsets.

To generate all subsets of the given set, we can use the **Breadth First Search (BFS)** approach. We can start with an empty set, iterate through all numbers one-by-one, and add them to existing sets to create new subsets.

Let's take the example-2 mentioned above to go through each step of our algorithm:

Given set: `[1, 5, 3]`

1. Start with an empty set: `[[]]`
2. Add the first number `1` to all the existing subsets to create new subsets: `[[], [1] ];`
3. Add the second number `5` to all the existing subsets: `[[], [1], [5], [1,5] ];`
4. Add the third number `3` to all the existing subsets: `[[], [1], [5], [1,5], [3], [1,3], [5,3], [1,5,3] ]`.

Since the input set has distinct elements, the above steps will ensure that we will not have any duplicate subsets.

```
function findSubsets(nums) {
  let subsets = []

  //start by adding the empty subset
  subsets.push([])

  for(let i = 0; i < nums.length; i++) {
    let currentNumber = nums[i]
    //we will take all existing subsets and insert the current number in them to creat
    const n = subsets.length

    for(let j = 0; j < n; j++) {
      //create a new subset from the existing subset and insert the current element to
      //clone the permutation?
      subsets.push([...subsets[j], nums[i]])
    }
  }
```

```
  return subsets
}

findSubsets([1, 5, 3])
findSubsets([1, 3])
```

- Since, in each step, the number of subsets doubles as we add each element to all the existing subsets, therefore, we will have a total of $O(2^N)$ subsets, where 'N' is the total number of elements in the input set. And since we construct a new subset from an existing set, therefore, the time complexity of the above algorithm will be $O(N*2^N)$.
- All the additional space used by our algorithm is for the output list. Since we will have a total of $O(2^N)$ subsets, and each subset can take up to $O(N)$ space, therefore, the space complexity of our algorithm will be $O(N*2^N)$.

Subsets With Duplicates (medium)

https://leetcode.com/problems/subsets-ii/

> Given a set of numbers that might contain duplicates, find all of its distinct subsets.

This problem follows the **Subsets** pattern and we can follow a similar **Breadth First Search (BFS)** approach. The only additional thing we need to do is handle duplicates. Since the given set can have duplicate numbers, if we follow the same approach discussed in **Subsets**, we will end up with duplicate subsets, which is not acceptable. To handle this, we will do two extra things:

1. Sort all numbers of the given set. This will ensure that all duplicate numbers are next to each other.
2. Follow the same **BFS** approach but whenever we are about to process a duplicate (i.e., when the current and the previous numbers are same), instead of adding the current number (which is a duplicate) to all the existing subsets, only add it to the subsets which were created in the previous step.

Let's take first Example mentioned below to go through each step of our algorithm:

```
Given set: [1, 5, 3, 3]
Sorted set: [1, 3, 3, 5]
```

1. Start with an empty set: `[[]]`
2. Add the first number `1` to all the existing subsets to create new subsets: `[[], [1]]`;
3. Add the second number `3` to all the existing subsets: `[[], [1], [3], [1,3]]`.
4. The next number `3` is a duplicate. If we add it to all existing subsets we will get:

```
[[], [1], [3], [1,3], [3], [1,3], [3,3], [1,3,3]]
```

```
We got two duplicate subsets: [3], [1,3]
Whereas we only needed the new subsets: [3,3], [1,3,3]
```

To handle this instead of adding `3` to all the existing subsets, we only add it to the new subsets which were created in the previous **3rd** step:

```
[[], [1], [3], [1,3], [3,3], [1,3,3]]
```

5. Finally, add the forth number `5` to all the existing subsets: `[[], [1], [3], [1,3], [3,3], [1,3,3], [5], [1,5], [3,5], [1,3,5], [3,3,5], [1,3,3,5]]`

```
function subsetsWithDupe(nums) {
  let subsets = []
  nums.sort((a, b) => a-b)

  //start by adding the empty subset
  subsets.push([])

  let start = 0
  let end = 0

  for(let i = 0; i < nums.length; i++) {
    //if the current and previous elements are the same,
    //create new subsets only from the subsets added in the previous step

    end = subsets.length

    for(let j = start; j < end; j++) {
      //create a new subset from the existing subset and add the current element to it
      subsets.push([...subsets[j], nums[i]])

      if(nums[i + 1] === nums[i]) {
        start = end
      } else {
        start = 0
      }
    }
  }
  return subsets
}

subsetsWithDupe([1, 5, 3, 3])
subsetsWithDupe([1, 3, 3])
```

- Since, in each step, the number of subsets doubles (if not duplicate) as we add each element to all the existing subsets, therefore, we will have a total of $O(2^N)$ subsets, where 'N' is the total number of elements in the input set. And since we construct a new subset from an existing set, therefore, the time complexity of the above algorithm will be $O(N*2^N)$.

- All the additional space used by our algorithm is for the output list. Since, at most, we will have a total of `O(2^N)` subsets, and each subset can take up to `O(N)` space, therefore, the space complexity of our algorithm will be `O(N*2^N)`.

Permutations (medium)

https://leetcode.com/problems/permutations/

> Given a set of distinct numbers, find all of its permutations.

Permutation is defined as the re-arranging of the elements of the set. For example, `{1, 2, 3}` has the following six permutations:

1. `{1, 2, 3}`
2. `{1, 3, 2}`
3. `{2, 1, 3}`
4. `{2, 3, 1}`
5. `{3, 1, 2}`
6. `{3, 2, 1}`

If a set has `'n'` distinct elements it will have `n!` permutations.

This problem follows the **Subsets** pattern and we can follow a similar **Breadth First Search (BFS)** approach. However, unlike **Subsets**, every permutation must contain all the numbers.

Let's take the example mentioned below to generate all the permutations. Following a **BFS** approach, we will consider one number at a time:

1. If the given set is empty then we have only an empty permutation set: `[]`
2. Let's add the first element `1`, the permutations will be: `[1]`
3. Let's add the second element `3`, the permutations will be: `[3,1], [1,3]`
4. Let's add the third element `5`, the permutations will be: `[5,3,1], [3,5,1], [3,1,5], [5,1,3], [1,5,3], [1,3,5]`

Let's analyze the permutations in the 3rd and 4th step. How can we generate permutations in the 4th step from the permutations of the 3rd step?

If we look closely, we will realize that when we add a new number `5`, we take each permutation of the previous step and insert the new number in every position to generate the new permutations. For example, inserting `5` in different positions of `[3,1]` will give us the following permutations:

1. Inserting `5` before `3`: `[5,3,1]`
2. Inserting `5` between `3` and `1`: `[3,5,1]`
3. Inserting `5` after `1`: `[3,1,5]`

```
function findPermutations(nums) {
  let result = []
  let permutations = [[]]
  let numsLength = nums.length

  for(let i = 0; i < nums.length; i++) {
    const currentNumber = nums[i]

    //we will take all existing permutations and add the
    //current number to create a new permutation
    const n = permutations.length

    for(let p = 0; p < n; p++) {
      const oldPermutation = permutations.shift()


      //create a new permustion by adding the current number at every position
      for(let j = 0; j < oldPermutation.length + 1; j++) {

        //clone the permutation
        const newPermutation = oldPermutation.slice(0)

        //insert the current number at index j
        newPermutation.splice(j, 0, currentNumber)

        if(newPermutation.length === numsLength) {
          result.push(newPermutation)
        } else {
          permutations.push(newPermutation)
        }
      }
    }
  }

  return result
}


findPermutations([1, 3, 5])
```

- We know that there are a total of `N!` permutations of a set with `'N'` numbers. In the algorithm above, we are iterating through all of these permutations with the help of the two 'for' loops. In each iteration, we go through all the current permutations to insert a new number in them. To insert a number into a permutation of size `'N'` will take `O(N)`, which makes the overall time complexity of our algorithm `O(N*N!)`.
- All the additional space used by our algorithm is for the `result` list and the `queue` to store the intermediate permutations. If you see closely, at any time, we don't have more than `N!` permutations between the result list and the queue. Therefore the overall space complexity to store `N!` permutations each containing `N` elements will be `O(N*N!)`.

Recursive Solution

```
function permute(nums) {
  //recursion
  let subsets = []

  generatePermuationsRecursive(nums, 0, [], subsets)

  return subsets
};

function generatePermuationsRecursive(nums, index, currentPermuation, subsets) {
    if(index === nums.length) {
      subsets.push(currentPermuation)
    } else {
      //create a new permuation by adding the current number at every position
      for(let i = 0; i < currentPermuation.length +1; i++) {
        let newPermutation = currentPermuation.slice(0)

        //insert nums[index] at index i
        newPermutation.splice(i, 0, nums[index])
        generatePermuationsRecursive(nums, index+1, newPermutation, subsets)
      }
    }
  }
```

String Permutations by changing case (medium)

Balanced Parentheses (hard)

https://leetcode.com/problems/generate-parentheses/

Unique Generalized Abbreviations (hard)

https://leetcode.com/problems/generalized-abbreviation/

🌟 Evaluate Expression (hard)

https://leetcode.com/problems/different-ways-to-add-parentheses/

🌟 Structurally Unique Binary Search Trees (hard)

https://leetcode.com/problems/unique-binary-search-trees-ii/

🌟 Count of Structurally Unique Binary Search Trees (hard)

https://leetcode.com/problems/unique-binary-search-trees/

Pattern 11: Modified Binary Search

As we know, whenever we are given a sorted **Array** or **LinkedList** or **Matrix**, and we are asked to find a certain element, the best algorithm we can use is the **Binary Search**.

Order-agnostic Binary Search (easy)

https://leetcode.com/problems/binary-search/

> Given a sorted array of numbers, find if a given number `key` is present in the array. Though we know that the array is sorted, we don't know if it's sorted in ascending or descending order. You should assume that the array can have duplicates.
>
> Write a function to return the index of the `key` if it is present in the array, otherwise return `-1`.

To make things simple, let's first solve this problem assuming that the input array is sorted in ascending order. Here are the set of steps for **Binary Search**:

1. Let's assume `start` is pointing to the first index and `end` is pointing to the last index of the input array (let's call it `arr`). This means:

```
int start = 0;
int end = arr.length - 1;
```

2. First, we will find the `middle` of `start` and `end`. An easy way to find the middle would be: *middle=(start+end)/2*. The safest way to find the middle of two numbers without getting an overflow is as follows:

```
middle  = start + (end-start)/2
```

3. Next, we will see if the `key` is equal to the number at index `middle`. If it is equal we return `middle` as the required index.
4. If `key` is not equal to number at index middle, we have to check two things:

- If `key < arr[middle]`, then we can conclude that the `key` will be smaller than all the numbers after index `middle` as the array is sorted in the ascending order. Hence, we can reduce our search to `end = mid - 1`.
- If `key > arr[middle]`, then we can conclude that the `key` will be greater than all numbers before index `middle` as the array is sorted in the ascending order. Hence, we can reduce our search to `start = mid + 1`.

- We will repeat steps 2-4 with new ranges of `start` to `end`. If at any time `start` becomes greater than `end`, this means that we can't find the `key` in the input array and we must return `-1`.

If the array is sorted in the descending order, we have to update the step 4 above as:

- If `key > arr[middle]`, then we can conclude that the `key` will be greater than all numbers after index `middle` as the array is sorted in the descending order. Hence, we can reduce our search to `end = mid - 1`.
- If `key < arr[middle]`, then we can conclude that the `key` will be smaller than all the numbers before index `middle` as the array is sorted in the descending order. Hence, we can reduce our search to `start = mid + 1`. Finally, how can we figure out the sort order of the input array? We can compare the numbers pointed out by `start` and `end` index to find the sort order. If `arr[start] < arr[end]`, it means that the numbers are sorted in ascending order otherwise they are sorted in the descending order.

```
function binarySearch (arr, key) {
  let start = 0
  let end = arr.length -1

  //check to see if arr is sorted ascending or descending
  const isAscending = arr[start] < arr[end]

  while(start <= end) {
    //calculate the middle of the current range
    let middle = Math.floor(start + (end-start)/2)

    if(key === arr[middle]) {
      return middle
    }

    if(isAscending) {
      //ascending order
      if(key < arr[middle]) {
        //the key can be in the first half
        end = middle - 1
      } else {
        //key > arr[middle], so the key can be in the
        //second half
        start = middle + 1
      }
    }
    else {
      //descending order
      if(key > arr[middle]) {
        //the key can be in the first half
        end = middle -1
      } else {
        //key < arr[middle], the key can be in the
        //second half
        start = middle + 1
```

```
        }
      }
    }

    // key not found
    return -1;
  };

  binarySearch([4, 6, 10], 10)//2
  binarySearch([1, 2, 3, 4, 5, 6, 7], 5)//4
  binarySearch([10, 6, 4], 10)//0
  binarySearch([10, 6, 4], 4)//2
```

- Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be `O(logN)` where `N` is the total elements in the given array.
- The algorithm runs in constant space `O(1)`.

Pattern 12: Bitwise XOR

XOR is a logical bitwise operator that returns `0` (false) if both bits are the same and returns `1` (true) otherwise. In other words, it only returns `1` if exactly one bit is set to `1` out of the two bits in comparison.

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

It is surprising to know the approaches that the XOR operator enables us to solve certain problems. For example, let's take a look at the following problem:

> Given an array of `n-1` integers in the range from `1` to `n`, find the one number that is missing from the array.

A straight forward approach to solve this problem can be:

1. Find the sum of all integers from `1` to `n`; let's call it `s1`.
2. Subtract all the numbers in the input array from `s1`; this will give us the missing number.

```
function findMissingNumber(arr) {
  const n = arr.length + 1

  //find sum of all numbers from 1 to n
  let s1 = 0
  for(let i = 1; i <= n; i++) {
    s1 += i
  }

  //subtract all numbers in input from sum
  arr.forEach((num) => {
    s1 -= num
  })

  //s1, is now the missing number
  return s1
}

findMissingNumber([1,5,2,6,4])//3
```

- The time complexity of the above algorithm is `O(n)` and the space complexity is `O(1)`.

What could go wrong with the above algorithm?

> While finding the sum of numbers from `1` to `n`, we can get integer overflow when `n` is large.

How can we avoid this? Can XOR help us here?

Remember the important property of XOR that it returns 0 if both the bits in comparison are the same. In other words, XOR of a number with itself will always result in 0. This means that if we XOR all the numbers in the input array with all numbers from the range `1` to `n` then each number in the input is going to get zeroed out except the missing number. Following are the set of steps to find the missing number using XOR:

1. XOR all the numbers from `1` to `n`, let's call it `x1`.
2. XOR all the numbers in the input array, let's call it `x2`.
3. The missing number can be found by `x1` XOR `x2`.

```
function findMissingNumber(arr) {
  const n = arr.length + 1

  //x1 represents XOR of all values from 1 to n
  //find sum of all numbers from 1 to n
  let x1 = 1
  for(let i = 2; i <= n; i++) {
    x1 = x1 ^ i
  }

  //x2 represents XOR of all values in arr
  let x2 = arr[0]
  for(let i = 1; i < n-1; i++) {
    x2 = x2 ^ arr[i]
  }

  //missing number is the xor of x1 and x2
  return x1 ^ x2
}

findMissingNumber([1,5,2,6,4])//3
```

- The time complexity of the above algorithm is `O(n)` and the space complexity is `O(1)`. The time and space complexities are the same as that of the previous solution but, in this algorithm, we will not have any integer overflow problem.

Important properties of XOR to remember

Following are some important properties of XOR to remember:

- Taking XOR of a number with itself returns 0, e.g.,
  - 1 ^ 1 = 0
  - 29 ^ 29 = 0
- Taking XOR of a number with 0 returns the same number, e.g.,
  - 1 ^ 0 = 1
  - 31 ^ 0 = 31
- XOR is Associative & Commutative, which means:
  - (a ^ b) ^ c = a ^ (b ^ c)
  - a ^ b = b ^ a

😕 Single Number (easy)

https://leetcode.com/problems/single-number/

> In a non-empty array of integers, every number appears twice except for one, find that single number.

One straight forward solution can be to use a **HashMap** kind of data structure and iterate through the input:

- If number is already present in **HashMap**, remove it.
- If number is not present in **HashMap**, add it.
- In the end, only number left in the **HashMap** is our required single number.

```
function singleNumber(arr) {
  //HashMap
  let numberMap = {}
  //if number is not in HashMap, add it
  for(let i of arr) {
    numberMap[i] = (numberMap[i] || 0) + 1
  }
  //if number is already in HashMap, remove it
  //number left at the end is out rquired single number
  return numberMap
}

findMissingNumber([1, 4, 2, 1, 3, 2, 3])//4
findMissingNumber([7, 9, 7])//9
```

Time and space complexity Time Complexity of the above solution will be `O(n)` and space complexity will also be `O(n)` .

Can we do better than this using the **XOR** Pattern?

Recall the following two properties of XOR:

- It returns zero if we take XOR of two same numbers.
- It returns the same number if we XOR with zero. So we can XOR all the numbers in the input; duplicate numbers will zero out each other and we will be left with the single number.

```
function singleNumber(arr) {
  //So we can XOR all the numbers in the input; duplicate numbers will zero out each o
  let num = 0

  for(let i = 0; i < arr.length; i++) {
    //console.log(num, arr[i], num^arr[i], "CL")
    num ^= arr[i]
  }
  return num
}

singleNumber([1, 4, 2, 1, 3, 2, 3])//4
singleNumber([7, 9, 7])//9
```

- Time complexity of this solution is `O(n)` as we iterate through all numbers of the input once.
- The algorithm runs in constant space `O(1)`.

😕 Two Single Numbers (medium)

https://leetcode.com/problems/single-number-iii/

> In a non-empty array of numbers, every number appears exactly twice except two numbers that appear only once. Find the two numbers that appear only once.

This problem is quite similar to **Single Number**, the only difference is that, in this problem, we have two single numbers instead of one. Can we still use XOR to solve this problem?

Let's assume `num1` and `num2` are the two single numbers. If we do XOR of all elements of the given array, we will be left with XOR of `num1` and `num2` as all other numbers will cancel each other because all of them appeared twice. Let's call this XOR `n1xn2`. Now that we have XOR of `num1` and `num2`, how can we find these two single numbers?

As we know that `num1` and `num2` are two different numbers, therefore, they should have at least one bit different between them. If a bit in `n1xn2` is '1', this means that `num1` and `num2` have different bits in that place, as we know that we can get '1' only when we do XOR of two different bits, i.e.,

```
1 XOR 0 = 0 XOR 1 = 1
```

We can take any bit which is '1' in `n1xn2` and partition all numbers in the given array into two groups based on that bit. One group will have all those numbers with that bit set to '0' and the other with the bit set to '1'. This will ensure that `num1` will be in one group and `num2` will be in the

other. We can take XOR of all numbers in each group separately to get `num1` and `num2`, as all other numbers in each group will cancel each other. Here are the steps of our algorithm:

1. Taking XOR of all numbers in the given array will give us XOR of `num1` and `num2`, calling this XOR as `n1xn2`.
2. Find any bit which is set in `n1xn2`. We can take the rightmost bit which is '1'. Let's call this `rightmostSetBit`.
3. Iterate through all numbers of the input array to partition them into two groups based on `rightmostSetBit`. Take XOR of all numbers in both the groups separately. Both these XORs are our required numbers.

```
function findSingleNumbers(nums) {
  //get the XOR of all the numbers
  let n1xn2 = 0
  nums.forEach((num) => {
    n1xn2 ^= num
  })

  //get the rightmost bit that is 1
  let rightmostSetBit = 1
  while((rightmostSetBit & n1xn2) === 0){
    rightmostSetBit = rightmostSetBit << 1
    //The left shift operator ( << ) shifts the first operand the specified number of
    //Excess bits shifted off to the left are discarded.
    //Zero bits are shifted in from the right.
  }
  let num1 = 0
  let num2 = 0

  nums.forEach((num)=> {
    //if the bit is set
    if((num & rightmostSetBit) !== 0) {
      num1 ^= num
    } else {
      //the bit is not set
      num2 ^= num
    }
  })
  return [num1, num2]
}

findSingleNumbers([1, 4, 2, 1, 3, 5, 6, 2, 3, 5])//[4, 6]
findSingleNumbers([2, 1, 3, 2])//[1, 3]
```

- The time complexity of this solution is `O(n)` where `n` is the number of elements in the input array.
- The algorithm runs in constant space `O(1)`.

😕 Complement of Base 10 Number (medium)

https://leetcode.com/problems/complement-of-base-10-integer/

Every non-negative integer N has a binary representation, for example, 8 can be represented as “1000” in binary and 7 as “0111” in binary.

The complement of a binary representation is the number in binary that we get when we change every 1 to a 0 and every 0 to a 1. For example, the binary complement of “1010” is “0101”.

For a given positive number N in base-10, return the complement of its binary representation as a base-10 integer.

Recall the following properties of XOR:

1. It will return 1 if we take XOR of two different bits i.e. `1^0 = 0^1 = 1`.
2. It will return 0 if we take XOR of two same bits i.e. `0^0 = 1^1 = 0`. In other words, XOR of two same numbers is 0.
3. It returns the same number if we XOR with 0.

From the above-mentioned first property, we can conclude that XOR of a number with its complement will result in a number that has all of its bits set to 1. For example, the binary complement of “101” is “010”; and if we take XOR of these two numbers, we will get a number with all bits set to 1, i.e., `101 ^ 010 = 111`

We can write this fact in the following equation:

```
number ^ complement = all_bits_set
```

Let’s add ‘number’ on both sides:

```
number ^ number ^ complement = number ^ all_bits_set
```

From the above-mentioned second property:

```
0 ^ complement = number ^ all_bits_set
```

From the above-mentioned third property:

```
complement = number ^ all_bits_set
```

We can use the above fact to find the complement of any number.

How do we calculate `all_bits_set` ? One way to calculate `all_bits_set` will be to first count the bits required to store the given number. We can then use the fact that for a number which is a complete power of ‘2’ i.e., it can be written as pow(2, n), if we subtract ‘1’ from such a number, we get a number which has ‘n’ least significant bits set to ‘1’. For example, ‘4’ which is a complete power of ‘2’, and ‘3’ (which is one less than 4) has a binary representation of ‘11’ i.e., it has ‘2’ least significant bits set to ‘1’.

```
function calculateBitwiseComplement(num) {
  //count number of total bits in num
  let bitCount = 0
  let n = num
  while(n > 0) {
    bitCount ++
    n = n >> 1
    //The right shift operator ( >> ) shifts the first operand the specified number of
    //Excess bits shifted off to the right are discarded.
  }

  //for a number which is a complete power of two,
  //i.e., it can be wrtiten as pow(1, n),
  //if we subtract 1 from such a number, we get a number
  //which has n least significant bits set to 1
  //For example, 4 which is a complete power of 2,
  //and 3 (which is one less than 4) ahs a binary
  //representation of 11 i.e., it has 2 least significant bits set to 1
  let allBitsSet = Math.pow(2, bitCount) -1

  //from the solution description: complement = number ^ allBitsSet
  return num ^ allBitsSet
}

calculateBitwiseComplement(8)//7, 8 is 1000 in binary, its complement is 0111 in binar
calculateBitwiseComplement(10)//5, 10 is 1010 in binary, its complement is 0101 in bin
```

- Time complexity of this solution is `O(b)` where `b` is the number of bits required to store the given number.
- Space complexity of this solution is `O(1)` .

🌟 Flip Binary Matrix(hard)