1. Introduction

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:

> 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

Here, we are asked to find the average of all contiguous subarrays of size '5' in the given array. Let's solve this:

1. For the first 5 numbers (subarray from index 0-4), the average is: $(1+3+2+6-1)/5 => 2.2$
2. The average of next 5 numbers (subarray from index 1-5) is: $(3+2+6-1+4)/5 => 2.8$
3. For the next 5 numbers (subarray from index 2-6), the average is: $(2+6-1+4+1)/5 => 2.4$
   ...

Here is the final output containing the averages of all contiguous subarrays of size 5:

```
Output: [2.2, 2.8, 2.4, 3.6, 2.8]
```

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. This is what the algorithm will look like:ddd

```
import java.util.Arrays;

class AverageOfSubarrayOfSizeK {

  public static double[] findAverages(int K, int[] arr) {

    double[] result = new double[arr.length - K + 1];

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

      // find sum of next 'K' elements
```

```
    double sum = 0;
    for (int j = i; j < i + K; j++)
      sum += arr[j];
    result[i] = sum / K; // calculate average
  }
  return result;
 }
 public static void main(String[] args) {
  double[] result = AverageOfSubarrayOfSizeK.findAverages(5, new int[] { 1, 3, 2, 6, -1, 4, 1, 8, 2 });
  System.out.println("Averages of subarrays of size K: " + Arrays.toString(result));
 }
}
```

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. For example, take the above-mentioned input:

As you can see, there are four overlapping elements between the subarray (indexed from 0-4) and the subarray (indexed from 1-5). Can we somehow reuse the `sum` we have calculated for the overlapping elements?

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:

```
import java.util.Arrays;

class AverageOfSubarrayOfSizeK {
  public static double[] findAverages(int K, int[] arr) {
    double[] result = new double[arr.length - K + 1];
    double windowSum = 0;
    int windowStart = 0;
    for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {
      windowSum += arr[windowEnd]; // add the next element
      // slide the window, we don't need to slide if we've not hit the required window size of 'k'
      if (windowEnd >= K - 1) {
        result[windowStart] = windowSum / K; // calculate the average
        windowSum -= arr[windowStart]; // subtract the element going out
        windowStart++; // slide the window ahead
      }
    }
    return result;
  }
  public static void main(String[] args) {
    double[] result = AverageOfSubarrayOfSizeK.findAverages(5, new int[] { 1, 3, 2, 6, -
1, 4, 1, 8, 2 });
    System.out.println("Averages of subarrays of size K: " + Arrays.toString(result));
  }
}
```

In the following chapters, we will apply the **Sliding Window** approach to solve a few problems.

In some problems, the size of the sliding window is not fixed. We have to expand or shrink the window based on the problem constraints. We will see a few examples of such problems in the next chapters.

Let's jump onto our first problem and apply the **Sliding Window** pattern.

Problem Statement

Given an array of positive numbers and a positive number ‘k,’ find the **maximum sum of any contiguous subarray of size ‘k’**.

Example 1:

Input: [2, 1, 5, 1, 3, 2], k=3
Output: 9
Explanation: Subarray with maximum sum is [5, 1, 3].

Example 2:

Input: [2, 3, 4, 1, 5], k=2
Output: 7
Explanation: Subarray with maximum sum is [3, 4].

Try it yourself

Try solving this question here:

```
class MaxSumSubArrayOfSizeK {

  public static int findMaxSumSubArray(int k, int[] arr) {

    // TODO: Write your code here

    return -1;

  }

}
```

Solution

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.

Following is the visual representation of this algorithm for Example-1:

Window Sum = 0 K = 3 2 1 5 1 3 2 Max Sum = 0 2 1 5 1 3 2 Window Sum = 8 Max Sum = 8 2 1 5 1 3 2 Window Sum = 7 Max Sum = 8 2 1 5 1 3 2 Window Sum = 9 Max Sum = 9 2 1 5 1 3 2 Max Sum = 9 Window Sum = 6

Code – Brute Force approach

Here is what our algorithm will look like:

```
class MaxSumSubArrayOfSizeK {

  public static int findMaxSumSubArray(int k, int[] arr) {

    int maxSum = 0, windowSum;

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

      windowSum = 0;

      for (int j = i; j < i + k; j++) {

        windowSum += arr[j];

      }

      maxSum = Math.max(maxSum, windowSum);

    }

    return maxSum;

  }


  public static void main(String[] args) {

    System.out.println("Maximum sum of a subarray of size K: "

        + MaxSumSubArrayOfSizeK.findMaxSumSubArray(3, new int[] { 2, 1, 5, 1, 3, 2 }));

    System.out.println("Maximum sum of a subarray of size K: "

        + MaxSumSubArrayOfSizeK.findMaxSumSubArray(2, new int[] { 2, 3, 4, 1, 5 }));

  }

}
```

The above algorithm's time complexity will be $O(N*K)$ where 'N' is the total number of elements in the given array. Is it possible to find a better algorithm than this?

A better 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. Here is what our algorithm will look like:

```
class MaxSumSubArrayOfSizeK {

  public static int findMaxSumSubArray(int k, int[] arr) {

    int windowSum = 0, maxSum = 0;

    int windowStart = 0;

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

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

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

      if (windowEnd >= k - 1) {

        maxSum = Math.max(maxSum, windowSum);

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

        windowStart++; // slide the window ahead

      }

    }

    return maxSum;
```

```
  }
  public static void main(String[] args) {
    System.out.println("Maximum sum of a subarray of size K: "
        + MaxSumSubArrayOfSizeK.findMaxSumSubArray(3, new int[] { 2, 1, 5, 1, 3, 2 }));
    System.out.println("Maximum sum of a subarray of size K: "
        + MaxSumSubArrayOfSizeK.findMaxSumSubArray(2, new int[] { 2, 3, 4, 1, 5 }));
  }
}
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$.

Space Complexity

The algorithm runs in constant space $O(1)$.

2. Smallest Subarray with a given sum (easy)

We'll cover the following

Problem Statement

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.

Example 1:

Input: [2, 1, 5, 2, 3, 2], S=7
Output: 2
Explanation: The smallest subarray with a sum greater than or equal to '7' is [5, 2].

Input: [2, 1, 5, 2, 8], S=7
Output: 1
Explanation: The smallest subarray with a sum greater than or equal to '7' is [8].

Example 3:

Input: [3, 4, 1, 1, 6], S=8
Output: 3
Explanation: Smallest subarrays with a sum greater than or equal to '8' are [3, 4, 1]
or [1, 1, 6].

Try it yourself

Try solving this question here:

Example 2:

```
class MinSizeSubArraySum {

  public static int findMinSubArray(int S, int[] arr) {

    // TODO: Write your code here

    return -1;

  }

}
```

Solution

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.

Here is the visual representation of this algorithm for the Example-1:

2 1 5 2 3 2 Window Sum = 0 Min Length = ∞ Required Sum = 7 2 1 5 2 3 2 Min Length = ∞ Window Sum = 2 2 1 5 2 3 2 Min Length = ∞ Window Sum = 3 2 1 5 2 3 2 Window Sum = 8 Min Length = 3 2 1 5 2 3 2 Window Sum = 7 2 1 5 2 3 2 2 1 5 2 3 2 Window Sum = 8 Min Length = 2 2 1 5 2 3 2 Min Length = 2 Window Sum = 7 window end window start window start window end window start window end window start window end window start window end window start window end Window Sum = 6 Min Length = 3 window start window end Min Length = 2 Min Length = 3 Window Sum >= 7, let's shrink the sliding window Window Sum >= 7, let's shrink the sliding window Window Sum still >= 7, let's shrink the sliding window 2 1 5 2 3 2 window start window end 2 1 5 2 3 2 Min Length = 2 Window Sum = 5 Window Sum = 2

Code

Here is what our algorithm will look like:

```
class MinSizeSubArraySum {
```

```
  public static int findMinSubArray(int S, int[] arr) {
    int windowSum = 0, minLength = Integer.MAX_VALUE;
    int windowStart = 0;
    for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {
      windowSum += arr[windowEnd]; // add the next element
      // shrink the window as small as possible until the 'windowSum' is smaller than 'S'
      while (windowSum >= S) {
        minLength = Math.min(minLength, windowEnd - windowStart + 1);
        windowSum -= arr[windowStart]; // subtract the element going out
        windowStart++; // slide the window ahead
      }
    }
    return minLength == Integer.MAX_VALUE ? 0 : minLength;
  }
  public static void main(String[] args) {
    int result = MinSizeSubArraySum.findMinSubArray(7, new int[] { 2, 1, 5, 2, 3, 2 });
    System.out.println("Smallest subarray length: " + result);
    result = MinSizeSubArraySum.findMinSubArray(7, new int[] { 2, 1, 5, 2, 8 });
    System.out.println("Smallest subarray length: " + result);
    result = MinSizeSubArraySum.findMinSubArray(8, new int[] { 3, 4, 1, 1, 6 });
    System.out.println("Smallest subarray length: " + result);
  }
}
```

Time Complexity

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 $O(N)$.

Space Complexity

The algorithm runs in constant space $O(1)$.

3. Longest Substring with K Distinct Characters (medium)

We'll cover the following

Problem Statement

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.

Example 1:

Input: String="araaci", K=2
Output: 4
Explanation: The longest substring with no more than '2' distinct characters is "araa".

Example 2:

Input: String="araaci", K=1
Output: 2
Explanation: The longest substring with no more than '1' distinct characters is "aa".

Example 3:

Input: String="cbbebi", K=3
Output: 5
Explanation: The longest substrings with no more than '3' distinct characters are "cbbeb" & "bb ebi".

Try it yourself

Try solving this question here:

```
using namespace std;
```

```
#include <iostream>
#include <string>
#include <unordered_map>
class LongestSubstringKDistinct {
 public:
  static int findLength(const string& str, int k) {
    int maxLength = 0;
    // TODO: Write your code here
    return maxLength;
  }
};
```

Solution

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.

Here is the visual representation of this algorithm for the Example-1:

window end window start window start window end window start window end window start window end window start window end window start window end window start window end window start window end a r a a c i K = 2 Max Length = 0 a r a a c i a r a a c i Max Length = 1 Max Length = 2 a r a a c i Max Length = 3 a r a a c i Max Length = 4 a r a a c i a r a a c i Max Length = 4 Max Length = 4 a r a a c i a r a a c i Max Length = 4 Max Length = 4 window start window end a r a a c i Number of distinct characters > 2, let's shrink the sliding window Number of distinct characters are still > 2, let's shrink the sliding window Number of distinct character > 2, let's shrink the sliding window

Code

Here is how our algorithm will look like:

```
unordered_map<char, int> charFrequencyMap;
// in the following loop we'll try to extend the range [windowStart, windowEnd]
for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
 char rightChar = str[windowEnd];
 charFrequencyMap[rightChar]++;
 // shrink the sliding window, until we are left with 'k' distinct characters in the frequency
 // map
 while ((int)charFrequencyMap.size() > k) {
  char leftChar = str[windowStart];
  charFrequencyMap[leftChar]--;
  if (charFrequencyMap[leftChar] == 0) {
   charFrequencyMap.erase(leftChar);
  }
  windowStart++; // shrink the window
 }
```

```
    maxLength = max(maxLength, windowEnd - windowStart + 1); // remember the maximum le
ngth so far
   }
   return maxLength;
 }
};
public:
 static int findLength(const string &str, int k) {
   int windowStart = 0, maxLength = 0;
#include <string>
#include <unordered_map>
class LongestSubstringKDistinct {
using namespace std;
#include <iostream>
```

Time Complexity

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 $O(N)$.

Space Complexity

The algorithm's space complexity is $O(K), as we will be storing a maximum of $K+1$ characters in the HashMap.

4. Fruits into Baskets (medium)

We'll cover the following

Problem Statement

Given an array of characters where each character represents a fruit tree, you are given **two baskets**, and your goal is to put **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.

Example 1:

Input: Fruit=['A', 'B', 'C', 'A', 'C']
Output: 3
Explanation: We can put 2 'C' in one basket and one 'A' in the other from the subarray ['C', 'A', ' C']

Example 2:

Input: Fruit=['A', 'B', 'C', 'B', 'B', 'C']
Output: 5
Explanation: We can put 3 'B' in one basket and two 'C' in the other basket.
This can be done if we start with the second letter: ['B', 'C', 'B', 'B', 'C']

Try it yourself

Try solving this question here:

```
using namespace std;

#include <iostream>

#include <unordered_map>

#include <vector>

class MaxFruitCountOf2Types {

 public:

  static int findLength(const vector<char>& arr) {
```

```
    int maxLength = 0;
    // TODO: Write your code here
    return maxLength;
  }
};
```

Solution

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.

Code

Here is what our algorithm will look like, only the highlighted lines are different from Longest Substring with K Distinct Characters:

```
class MaxFruitCountOf2Types {
 public:
  static int findLength(const vector<char>& arr) {
    int windowStart = 0, maxLength = 0;
    unordered_map<char, int> fruitFrequencyMap;
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < arr.size(); windowEnd++) {
      fruitFrequencyMap[arr[windowEnd]]++;
      // shrink the sliding window, until we are left with '2' fruits in the frequency map
      while ((int)fruitFrequencyMap.size() > 2) {
        fruitFrequencyMap[arr[windowStart]]--;
        if (fruitFrequencyMap[arr[windowStart]] == 0) {
```

```
        fruitFrequencyMap.erase(arr[windowStart]);
      }
      windowStart++;  // shrink the window
    }
    maxLength = max(maxLength, windowEnd - windowStart + 1);
  }
  return maxLength;
 }
};
int main(int argc, char* argv[]) {
 cout << "Maximum number of fruits: "
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;
```

Time Complexity

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 $O(N)$.

Space Complexity

The algorithm runs in constant space $O(1)$ as there can be a maximum of three types of fruits stored in the frequency map.

Similar Problems

Problem 1: Longest Substring with at most 2 distinct characters

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

Solution: This problem is exactly similar to our parent problem.

5. No-repeat Substring (hard)

We'll cover the following

Problem Statement

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

Example 1:

Input: String="aabccbb"
Output: 3
Explanation: The longest substring without any repeating characters is "abc".

Example 2:

Input: String="abbbb"
Output: 2
Explanation: The longest substring without any repeating characters is "ab".

Example 3:

Input: String="abccde"
Output: 3
Explanation: Longest substrings without any repeating characters are "abc" & "cde".

Try it yourself

Try solving this question here:

```
namespace std;
```

```
#include <iostream>
#include <string>
#include <unordered_map>
class NoRepeatSubstring {
 public:
  static int findLength(const string& str) {
    int maxLength = 0;
    // TODO: Write your code here
    return maxLength;
  }
};
```

Solution

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.

Code

Here is what our algorithm will look like:

```
using namespace std;
#include <iostream>
#include <string>
#include <unordered_map>
class NoRepeatSubstring {
 public:
  static int findLength(const string& str) {
```

```
    int windowStart = 0, maxLength = 0;
    unordered_map<char, int> charIndexMap;
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      // if the map already contains the 'rightChar', shrink the window from the beginning so that
      // we have only one occurrence of 'rightChar'
      if (charIndexMap.find(rightChar) != charIndexMap.end()) {
        // this is tricky; in the current window, we will not have any 'rightChar' after its
        // previous index and if 'windowStart' is already ahead of the last index of 'rightChar',
        // we'll keep 'windowStart'
        windowStart = max(windowStart, charIndexMap[rightChar] + 1);
      }
      charIndexMap[rightChar] = windowEnd;  // insert the 'rightChar' into the map
      maxLength =
          max(maxLength, windowEnd - windowStart + 1);  // remember the maximum length so far
    }
    return maxLength;
  }
};
```

Time Complexity

The above algorithm's time complexity will be $O(N)$, where 'N' is the number of characters in the input string.

Space Complexity

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 $O(1)$ $O(1)$; in this case, we can use a fixed-size array instead of the **HashMap**.

6. Longest Substring with Same Letters after Replacement (hard)

We'll cover the following

Problem Statement

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.

Example 1:

Input: String="aabccbb", k=2
Output: 5
Explanation: Replace the two 'c' with 'b' to have a longest repeating substring "bbbbb".

Example 2:

Input: String="abbcb", k=1
Output: 4
Explanation: Replace the 'c' with 'b' to have a longest repeating substring "bbbb".

Example 3:

Input: String="abccde", k=1
Output: 3
Explanation: Replace the 'b' or 'd' with 'c' to have the longest repeating substring "ccc".

Try it yourself

Try solving this question here:

```
using namespace std;
```

```
#include <iostream>
#include <string>
#include <unordered_map>
class CharacterReplacement {
 public:
  static int findLength(const string& str, int k) {
    int maxLength = 0;
    // TODO: Write your code here
    return maxLength;
  }
};
```

Solution

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.

Code

Here is what our algorithm will look like:

```
  unordered_map<char, int> letterFrequencyMap;
  // try to extend the range [windowStart, windowEnd]
  for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
    char rightChar = str[windowEnd];
    letterFrequencyMap[rightChar]++;
    maxRepeatLetterCount = max(maxRepeatLetterCount, letterFrequencyMap[rightChar]);
    // current window size is from windowStart to windowEnd, overall we have a letter which is
    // repeating 'maxRepeatLetterCount' times, this means we can have a window which has on
e
    // letter repeating 'maxRepeatLetterCount' times and the remaining letters we should replac
e.
    // if the remaining letters are more than 'k', it is the time to shrink the window as we
    // are not allowed to replace more than 'k' letters
    if (windowEnd - windowStart + 1 - maxRepeatLetterCount > k) {
      char leftChar = str[windowStart];
      letterFrequencyMap[leftChar]--;
      windowStart++;
    }
    maxLength = max(maxLength, windowEnd - windowStart + 1);
  }
  return maxLength;
}
```

```
};

int main(int argc, char *argv[]) {
  cout << CharacterReplacement::findLength("aabccbb", 2) << endl;
  cout << CharacterReplacement::findLength("abbcb", 1) << endl;
  cout << CharacterReplacement::findLength("abccde", 1) << endl;
}
```

Time Complexity

The above algorithm's time complexity will be $O(N)$, where 'N' is the number of letters in the input string.

Space Complexity

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 $O(1)$.

7. Longest Subarray with Ones after Replacement (hard)

We'll cover the following

Problem Statement

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**.

Example 1:

Input: Array=[0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1], k=2
Output: 6
Explanation: Replace the '0' at index 5 and 8 to have the longest contiguous subarray of 1s having length 6.

Example 2:

Input: Array=[0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1], k=3
Output: 9
Explanation: Replace the '0' at index 6, 9, and 10 to have the longest contiguous subarray of 1s having length 9.

Try it yourself

Try solving this question here:

```
using namespace std;

#include <iostream>

#include <vector>

class ReplacingOnes {

 public:

  static int findLength(const vector<int>& arr, int k) {

    int  maxLength = 0;

    // TODO: Write your code here

    return maxLength;

  }

};
```

Solution

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.

Code

Here is how our algorithm will look like:

```
      int windowStart = 0, maxLength = 0, maxOnesCount = 0;
   // try to extend the range [windowStart, windowEnd]
   for (int windowEnd = 0; windowEnd < arr.size(); windowEnd++) {
     if (arr[windowEnd] == 1) {
       maxOnesCount++;
     }
     // current window size is from windowStart to windowEnd, overall we have a maximum of 1
s
     // repeating a maximum of 'maxOnesCount' times, this means that we can have a window w
ith
     // 'maxOnesCount' 1s and the remaining are 0s which should replace with 1s.
     // now, if the remaining 0s are more than 'k', it is the time to shrink the window as we
     // are not allowed to replace more than 'k' Os
     if (windowEnd - windowStart + 1 - maxOnesCount > k) {
       if (arr[windowStart] == 1) {
         maxOnesCount--;
       }
       windowStart++;
     }
     maxLength = max(maxLength, windowEnd - windowStart + 1);
   }
   return maxLength;
 }
};
int main(int argc, char* argv[]) {
```

```
  cout << ReplacingOnes::findLength(vector<int>{0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1}, 2) << endl;
  cout << ReplacingOnes::findLength(vector<int>{0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1}, 3) << endl;
}
```

Time Complexity

The above algorithm's time complexity will be $O(N)$, where 'N' is the count of numbers in the input array.

Space Complexity

The algorithm runs in constant space $O(1)$.

8. Problem Challenge 1

We'll cover the following

Permutation in a String (hard)

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:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

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

Example 1:

Input: String="oidbcaf", Pattern="abc"
Output: true
Explanation: The string contains "bca" which is a permutation of the given pattern.

Example 2:

Input: String="odicf", Pattern="dc"
Output: false
Explanation: No permutation of the pattern is present in the given string as a substring.

Example 3:

Input: String="bcdxabcdy", Pattern="bcdyabcdx"
Output: true
Explanation: Both the string and the pattern are a permutation of each other.

Example 4:

Input: String="aaacb", Pattern="abc"
Output: true
Explanation: The string contains "acb" which is a permutation of the given pattern.

Try it yourself

Try solving this question here:

```
using namespace std;

#include <iostream>

#include <string>

class StringPermutation {

 public:

  static bool findPermutation(const string &str, const string &pattern) {

    // TODO: Write your code here

    return false;

  }

};
```

9. Solution Review: Problem Challenge 1

We'll cover the following

- Permutation in a String (hard)

Permutation in a String (hard)

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:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

If a string has ‘n’ distinct characters, it will have n! permutations.

Example 1:

Input: String="oidbcaf", Pattern="abc"
Output: true
Explanation: The string contains "bca" which is a permutation of the given pattern.

Example 2:

Input: String="odicf", Pattern="dc"
Output: false
Explanation: No permutation of the pattern is present in the given string as a substring.

Example 3:

Input: String="bcdxabcdy", Pattern="bcdyabcdx"
Output: true
Explanation: Both the string and the pattern are a permutation of each other.

Example 4:

Input: String="aaacb", Pattern="abc"
Output: true
Explanation: The string contains "acb" which is a permutation of the given pattern.

Solution

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:

1. Create a **HashMap** to calculate the frequencies of all characters in the pattern.
2. Iterate through the string, adding one character at a time in the sliding window.
3. 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.
4. 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.
5. 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**.

Code

Here is what our algorithm will look like:

```
for (auto chr : pattern) {
  charFrequencyMap[chr]++;
}
// our goal is to match all the characters from the 'charFrequencyMap' with the current window
// try to extend the range [windowStart, windowEnd]
for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
  char rightChar = str[windowEnd];
  if (charFrequencyMap.find(rightChar) != charFrequencyMap.end()) {
    // decrement the frequency of the matched character
```

```
      charFrequencyMap[rightChar]--;
      if (charFrequencyMap[rightChar] == 0) {  // character is completely matched
        matched++;
      }
    }
    if (matched == (int)charFrequencyMap.size()) {
      return true;
    }
    if (windowEnd >= pattern.length() - 1) {  // shrink the window
      char leftChar = str[windowStart++];
      if (charFrequencyMap.find(leftChar) != charFrequencyMap.end()) {
        if (charFrequencyMap[leftChar] == 0) {
          matched--;  // before putting the character back, decrement the matched count
        }
        // put the character back for matching
        charFrequencyMap[leftChar]++;
      }
    }
  }
```

Time Complexity

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.

Space Complexity

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)

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*N* characters. For example, here are the six anagrams of the string “abc”:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

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

Example 1:

Input: String="ppqp", Pattern="pq"
Output: [1, 2]
Explanation: The two anagrams of the pattern in the given string are "pq" and "qp".

Example 2:

Input: String="abbcabc", Pattern="abc"
Output: [2, 3, 4]
Explanation: The three anagrams of the pattern in the given string are "bca", "cab", and "abc".

Solution

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.

```
using namespace std;
```

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

class StringAnagrams {
 public:
  static vector<int> findStringAnagrams(const string &str, const string &pattern) {
    int windowStart = 0, matched = 0;
    unordered_map<char, int> charFrequencyMap;
    for (auto chr : pattern) {
      charFrequencyMap[chr]++;
    }

    vector<int> resultIndices;
    // our goal is to match all the characters from the map with the current window
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      // decrement the frequency of the matched character
      if (charFrequencyMap.find(rightChar) != charFrequencyMap.end()) {
        charFrequencyMap[rightChar]--;
        if (charFrequencyMap[rightChar] == 0) {
          matched++;
        }
      }

      if (matched == (int)charFrequencyMap.size()) {  // have we found an anagram?
        resultIndices.push_back(windowStart);
```

```
      }

      if (windowEnd >= pattern.length() - 1) {  // shrink the window
        char leftChar = str[windowStart++];
        if (charFrequencyMap.find(leftChar) != charFrequencyMap.end()) {
          if (charFrequencyMap[leftChar] == 0) {
            matched--;  // before putting the character back, decrement the matched count
          }
          // put the character back
          charFrequencyMap[leftChar]++;
        }
      }
    }

    return resultIndices;
  }
};

int main(int argc, char *argv[]) {
  auto result = StringAnagrams::findStringAnagrams("ppqp", "pq");
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = StringAnagrams::findStringAnagrams("abbcabc", "abc");
  for (auto num : result) {
    cout << num << " ";
  }
```

```
  cout << endl;
}
```

Time Complexity

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.

Space Complexity

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)

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

Example 1:

Input: String="aabdec", Pattern="abc"
Output: "abdec"
Explanation: The smallest substring having all characters of the pattern is "abdec"

Example 2:

Input: String="abdbca", Pattern="abc"
Output: "bca"
Explanation: The smallest substring having all characters of the pattern is "bca".

Example 3:

Input: String="adcad", Pattern="abc"
Output: ""
Explanation: No substring in the given string has all characters of the pattern.

Solution

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.

```
using namespace std;


#include <iostream>

#include <string>

#include <unordered_map>


class MinimumWindowSubstring {

 public:

  static string findSubstring(const string &str, const string &pattern) {

    int windowStart = 0, matched = 0, minLength = str.length() + 1, subStrStart = 0;

    unordered_map<char, int> charFrequencyMap;

    for (auto chr : pattern) {

      charFrequencyMap[chr]++;
```

```
}

// try to extend the range [windowStart, windowEnd]
for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
  char rightChar = str[windowEnd];
  if (charFrequencyMap.find(rightChar) != charFrequencyMap.end()) {
    charFrequencyMap[rightChar]--;
    if (charFrequencyMap[rightChar] >= 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;
    }

    char leftChar = str[windowStart++];
    if (charFrequencyMap.find(leftChar) != charFrequencyMap.end()) {
      // note that we could have redundant matching characters, therefore we'll decrement the
      // matched count only when a useful occurrence of a matched character is going out of the
      // window
      if (charFrequencyMap[leftChar] == 0) {
        matched--;
      }
      charFrequencyMap[leftChar]++;
    }
```

```
    }
   }


   return minLength > str.length() ? "" : str.substr(subStrStart, minLength);
  }
};


int main(int argc, char *argv[]) {
  cout << MinimumWindowSubstring::findSubstring("aabdec", "abc") << endl;
  cout << MinimumWindowSubstring::findSubstring("abdbca", "abc") << endl;
  cout << MinimumWindowSubstring::findSubstring("adcad", "abc") << endl;
}
```

Time Complexity

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.

Space Complexity

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)

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.

Example 1:

Input: String="catfoxcat", Words=["cat", "fox"]
Output: [0, 3]
Explanation: The two substring containing both the words are "catfox" & "foxcat".

Example 2:

Input: String="catcatfoxfox", Words=["cat", "fox"]
Output: [3]
Explanation: The only substring containing both the words is "catfox".

Solution

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.

```
using namespace std;


#include <iostream>

#include <string>

#include <unordered_map>

#include <vector>


class WordConcatenation {

 public:
```

```
static vector<int> findWordConcatenation(const string &str, const vector<string> &words) {
  unordered_map<string, int> wordFrequencyMap;
  for (auto word : words) {
    wordFrequencyMap[word]++;
  }

  vector<int> resultIndices;
  int wordsCount = words.size(), wordLength = words[0].length();

  for (int i = 0; i <= int(str.length()) - wordsCount * wordLength; i++) {
    unordered_map<string, int> wordsSeen;
    for (int j = 0; j < wordsCount; j++) {
      int nextWordIndex = i + j * wordLength;
      // get the next word from the string
      string word = str.substr(nextWordIndex, wordLength);
      if (wordFrequencyMap.find(word) ==
          wordFrequencyMap.end()) {  // break if we don't need this word
        break;
      }

      wordsSeen[word]++;  // add the word to the 'wordsSeen' map

      // no need to process further if the word has higher frequency than required
      if (wordsSeen[word] > wordFrequencyMap[word]) {
        break;
      }

      if (j + 1 == wordsCount) {  // store index if we have found all the words
        resultIndices.push_back(i);
```

```
      }
     }
    }

    return resultIndices;
  }
};

int main(int argc, char *argv[]) {
  vector<int> result =
      WordConcatenation::findWordConcatenation("catfoxcat", vector<string>{"cat", "fox"});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = WordConcatenation::findWordConcatenation("catcatfoxfox", vector<string>{"cat", "fox"});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time Complexity

The time complexity of the above algorithm will be $O(N * M * Len)$ $O(N*M*Len)$ 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.

Space Complexity

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 $O(M+N)$.

10. Introduction

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.

Here is the visual representation of this algorithm:

1 2 3 4 6 target sum = 6 1 + 6 > target sum, therefore let's decrement Pointer2 1 2 3 4 6 1 + 4 < target sum, therefore let's increment Pointer1 1 2 3 4 6 2 + 4 == target sum, we have found our pair!

The time complexity of the above algorithm will be $O(N)$.

In the following chapters, we will apply the **Two Pointers** approach to solve a few problems.

Problem Statement

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.

Example 1:

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

Example 2:

Input: [2, 5, 9, 11], target=11
Output: [0, 2]
Explanation: The numbers at index 0 and 2 add up to 11: 2+9=11

Solution

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.

Here is the visual representation of this algorithm for Example-1:

1 2 3 4 6 target sum = 6 1 + 6 > target sum, therefore let's decrement Pointer2 1 2 3 4 6 1 + 4 < target sum, therefore let's increment Pointer1 1 2 3 4 6 2 + 4 == target sum, we have found our pair!

```
using namespace std;


#include <iostream>

#include <vector>


class PairWithTargetSum {

public:

  static pair<int, int> search(const vector<int> &arr, int targetSum) {

    int left = 0, right = arr.size() - 1;

    while (left < right) {

      int currentSum = arr[left] + arr[right];

      if (currentSum == targetSum) { // found the pair

        return make_pair(left, right);

      }


      if (targetSum > currentSum)

        left++; // we need a pair with a bigger sum

      else

        right--; // we need a pair with a smaller sum
```

```
    }
    return make_pair(-1, -1);
  }
};

int main(int argc, char *argv[]) {
  auto result = PairWithTargetSum::search(vector<int>{1, 2, 3, 4, 6}, 6);
  cout << "Pair with target sum: [" << result.first << ", " << result.second << "]" << endl;
  result = PairWithTargetSum::search(vector<int>{2, 5, 9, 11}, 11);
  cout << "Pair with target sum: [" << result.first << ", " << result.second << "]" << endl;
}
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$, where 'N' is the total number of elements in the given array.

Space Complexity

The algorithm runs in constant space $O(1)$.

An Alternate approach

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$X+Y==Target$". We will do two things here:

1. Search for 'Y' (which is equivalent to "Target - X$Target-X$") 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.

```
using namespace std;
```

```
#include <iostream>
#include <unordered_map>
#include <vector>

class PairWithTargetSum {
 public:
  static pair<int, int> search(const vector<int>& arr, int targetSum) {
    unordered_map<int, int> nums;  // to store number and its index
    for (int i = 0; i < arr.size(); i++) {
      if (nums.find(targetSum - arr[i]) != nums.end()) {
        return make_pair(nums[targetSum - arr[i]], i);
      } else {
        nums[arr[i]] = i;  // put the number and its index in the map
      }
    }
    return make_pair(-1, -1);  // pair not found
  }
};

int main(int argc, char* argv[]) {
  auto result = PairWithTargetSum::search(vector<int>{1, 2, 3, 4, 6}, 6);
  cout << "Pair with target sum: [" << result.first << ", " << result.second << "]" << endl;
  result = PairWithTargetSum::search(vector<int>{2, 5, 9, 11}, 11);
  cout << "Pair with target sum: [" << result.first << ", " << result.second << "]" << endl;
}
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$, where 'N' is the total number of elements in the given array.

Space Complexity

The space complexity will also be $O(N)$, as, in the worst case, we will be pushing 'N' numbers in the **HashTable**.

Problem Statement

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.

Example 1:

Input: [2, 3, 3, 3, 6, 9, 9]
Output: 4
Explanation: The first four elements after removing the duplicates will be [2, 3, 6, 9].

Example 2:

Input: [2, 2, 2, 11]
Output: 2
Explanation: The first two elements after removing the duplicates will be [2, 11].

Solution

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.

Here is the visual representation of this algorithm for Example-1:

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

```
using namespace std;


#include <iostream>
#include <vector>


class RemoveDuplicates {
 public:
  static int remove(vector<int>& arr) {
    int nextNonDuplicate = 1;  // index of the next non-duplicate element
    for (int i = 1; i < arr.size(); i++) {
      if (arr[nextNonDuplicate - 1] != arr[i]) {
        arr[nextNonDuplicate] = arr[i];
        nextNonDuplicate++;
      }
    }

    return nextNonDuplicate;
  }
};


int main(int argc, char* argv[]) {
  vector<int> arr = {2, 3, 3, 3, 6, 9, 9};
  cout << "Array new length: " << RemoveDuplicates::remove(arr) << endl;

  arr = vector<int>{2, 2, 2, 11};
  cout << "Array new length: " << RemoveDuplicates::remove(arr) << endl;
```

```
}
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$, where 'N' is the total number of elements in the given array.

Space Complexity

The algorithm runs in constant space $O(1)$.

Similar Questions

Problem 1: 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.

Example 1:

Input: [3, 2, 3, 6, 3, 10, 9, 3], Key=3
Output: 4
Explanation: The first four elements after removing every 'Key' will be [2, 6, 10, 9].

Example 2:

Input: [2, 11, 2, 2, 1], Key=2
Output: 2
Explanation: The first two elements after removing every 'Key' will be [11, 1].

Solution: This problem is quite similar to our parent problem. We can follow a two-pointer approach and shift numbers left upon encountering the 'key'. Here is what the code will look like:

```
using namespace std;


#include <iostream>

#include <vector>


class RemoveElement {
```

```
 public:
  static int remove(vector<int>& arr, int key) {
    int nextElement = 0;  // index of the next element which is not 'key'
    for (int i = 0; i < arr.size(); i++) {
      if (arr[i] != key) {
        arr[nextElement] = arr[i];
        nextElement++;
      }
    }

    return nextElement;
  }
};

int main(int argc, char* argv[]) {
  vector<int> arr = {3, 2, 3, 6, 3, 10, 9, 3};
  cout << "Array new length: " << RemoveElement::remove(arr, 3) << endl;

  arr = vector<int>{2, 11, 2, 2, 1};
  cout << "Array new length: " << RemoveElement::remove(arr, 2) << endl;
}
```

Time and Space Complexity: 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 $O(1)$.

Problem Statement

Given a sorted array, create a new array containing **squares of all the numbers of the input array** in the sorted order.

Example 1:

Input: [-2, -1, 0, 2, 3]
Output: [0, 1, 4, 4, 9]

Example 2:

Input: [-3, -1, 0, 1, 2]
Output: [0, 1, 1, 4, 9]

Solution

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. For the above-mentioned Example-1, we will do something like this:

-2 -1 0 2 3

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. For the above-mentioned Example-1, we will do something like this:

-2 -1 0 2 3

```
using namespace std;
```

```
#include <iostream>
#include <vector>

class SortedArraySquares {
 public:
  static vector<int> makeSquares(const vector<int>& arr) {
    int n = arr.size();
    vector<int> squares(n);
    int highestSquareIdx = n - 1;
    int left = 0, right = n - 1;
    while (left <= right) {
      int leftSquare = arr[left] * arr[left];
      int rightSquare = arr[right] * arr[right];
      if (leftSquare > rightSquare) {
        squares[highestSquareIdx--] = leftSquare;
        left++;
      } else {
        squares[highestSquareIdx--] = rightSquare;
        right--;
      }
    }
    return squares;
  }
};

int main(int argc, char* argv[]) {
  vector<int> result = SortedArraySquares::makeSquares(vector<int>{-2, -1, 0, 2, 3});
  for (auto num : result) {
    cout << num << " ";
```

```
  }
  cout << endl;

  result = SortedArraySquares::makeSquares(vector<int>{-3, -1, 0, 1, 2});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

The above algorithm's time complexity will be $O(N)$ as we are iterating the input array only once.

Space complexity

The above algorithm's space complexity will also be $O(N)$; this space will be used for the output array.

Problem Statement

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

Example 1:

```
Input: [-3, 0, 1, 2, -1, 1, -2]
Output: [-3, 1, 2], [-2, 0, 2], [-2, 1, 1], [-1, 0, 1]
Explanation: There are four unique triplets whose sum is equal to zero.
```

Example 2:

```
Input: [-5, 2, -1, -2, 3]
Output: [[-5, 2, 3], [-2, -1, 3]]
Explanation: There are two unique triplets whose sum is equal to zero.
```

Solution

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$X+Y+Z==0$. At this stage, our problem translates into finding a pair whose sum is equal to "-X−X" (as from the above equation Y + Z == -X$Y+Z==-X$).

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.

```
using namespace std;

#include <algorithm>

#include <iostream>

#include <vector>

class TripletSumToZero {

public:

 static vector<vector<int>> searchTriplets(vector<int> &arr) {

  sort(arr.begin(), arr.end());

  vector<vector<int>> triplets;

  for (int i = 0; i < arr.size() - 2; i++) {

   if (i > 0 && arr[i] == arr[i - 1]) { // skip same element to avoid duplicate triplets

    continue;

   }

   searchPair(arr, -arr[i], i + 1, triplets);

  }
```

```
    return triplets;
  }

private:
  static void searchPair(const vector<int> &arr, int targetSum, int left,
                         vector<vector<int>> &triplets) {
    int right = arr.size() - 1;
    while (left < right) {
      int currentSum = arr[left] + arr[right];
      if (currentSum == targetSum) { // found the pair
        triplets.push_back({-targetSum, arr[left], arr[right]});
        left++;
        right--;
        while (left < right && arr[left] == arr[left - 1]) {
          left++; // skip same element to avoid duplicate triplets
        }
        while (left < right && arr[right] == arr[right + 1]) {
          right--; // skip same element to avoid duplicate triplets
        }
      } else if (targetSum > currentSum) {
        left++; // we need a pair with a bigger sum
      } else {
        right--; // we need a pair with a smaller sum
      }
    }
  }
};

int main(int argc, char *argv[]) {
```

```
 vector<int> vec = {-3, 0, 1, 2, -1, 1, -2};
 auto result = TripletSumToZero::searchTriplets(vec);
 for (auto vec : result) {
  cout << "[";
  for (auto num : vec) {
   cout << num << " ";
  }
  cout << "]";
 }
 cout << endl;

 vec = {-5, 2, -1, -2, 3};
 result = TripletSumToZero::searchTriplets(vec);
 for (auto vec : result) {
  cout << "[";
  for (auto num : vec) {
   cout << num << " ";
  }
  cout << "]";
 }
}
```

Time complexity

Sorting the array will take $O(N * logN)$ $O(N*logN)$. The `searchPair()` function will take $O(N)$ $O(N)$. As we are calling `searchPair()` for every number in the input array, this means that overall `searchTriplets()` will take $O(N * logN + N^2)$ $O(N*logN+N2)$, which is asymptotically equivalent to $O(N^2)$ $O(N2)$.

Space complexity

Ignoring the space required for the output array, the space complexity of the above algorithm will be $O(N)$ which is required for sorting.

Problem Statement

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.

Example 1:

Input: [-2, 0, 1, 2], target=2
Output: 1
Explanation: The triplet [-2, 1, 2] has the closest sum to the target.

Example 2:

Input: [-3, -1, 1, 2], target=1
Output: 0
Explanation: The triplet [-3, 1, 2] has the closest sum to the target.

Example 3:

Input: [1, 0, 1, 1], target=100
Output: 3
Explanation: The triplet [1, 1, 1] has the closest sum to the target.

Solution

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.

```
using namespace std;
```

```
#include <algorithm>
#include <iostream>
#include <limits>
#include <vector>

class TripletSumCloseToTarget {
 public:
  static int searchTriplet(vector<int>& arr, int targetSum) {
    sort(arr.begin(), arr.end());
    int smallestDifference = numeric_limits<int>::max();
    for (int i = 0; i < arr.size() - 2; i++) {
      int left = i + 1, right = arr.size() - 1;
      while (left < right) {
        // comparing the sum of three numbers to the 'targetSum' can cause overflow
        // so, we will try to find a target difference
        int targetDiff = targetSum - arr[i] - arr[left] - arr[right];
        if (targetDiff == 0) {           //  we've found a triplet with an exact sum
          return targetSum - targetDiff;  // return sum of all the numbers
        }

        if (abs(targetDiff) < abs(smallestDifference) ||
            (abs(targetDiff) == abs(smallestDifference) && targetDiff > smallestDifference)) {
          smallestDifference = targetDiff;  // save the closest difference
        }

        if (targetDiff > 0) {
          left++;  // we need a triplet with a bigger sum
        } else {
          right--;  // we need a triplet with a smaller sum
```

```
      }
     }
    }
    return targetSum - smallestDifference;
  }
};

int main(int argc, char* argv[]) {
  vector<int> vec = {-2, 0, 1, 2};
  cout << TripletSumCloseToTarget::searchTriplet(vec, 2) << endl;
  vec = {-3, -1, 1, 2};
  cout << TripletSumCloseToTarget::searchTriplet(vec, 1) << endl;
  vec = {1, 0, 1, 1};
  cout << TripletSumCloseToTarget::searchTriplet(vec, 100) << endl;
}
```

Time complexity

Sorting the array will take $O(N*logN)$. Overall, the function will take $O(N*logN+N_2)$, which is asymptotically equivalent to $O(N_2)$.

Space complexity

The above algorithm's space complexity will be $O(N)$, which is required for sorting.

Problem Statement

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.

Example 1:

Input: [-1, 0, 2, 3], target=3
Output: 2
Explanation: There are two triplets whose sum is less than the target: [-1, 0, 3], [-1, 0, 2]

Example 2:

Input: [-1, 4, 2, 1, 3], target=5
Output: 4
Explanation: There are four triplets whose sum is less than the target:
[-1, 1, 4], [-1, 1, 3], [-1, 1, 2], [-1, 2, 3]

Solution

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$X+Y+Z<target$. At this stage, our problem translates into finding a pair whose sum is less than "target - X$target-X$" (as from the above equation Y + Z == target - X$Y+Z==target-X$). We can use a similar approach as discussed in Triplet Sum to Zero.

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <vector>


class TripletWithSmallerSum {

 public:
```

```
 static int searchTriplets(vector<int> &arr, int target) {
  sort(arr.begin(), arr.end());
  int count = 0;
  for (int i = 0; i < arr.size() - 2; i++) {
   count += searchPair(arr, target - arr[i], i);
  }
  return count;
 }

private:
 static int searchPair(const vector<int> &arr, int targetSum, int first) {
  int count = 0;
  int left = first + 1, right = arr.size() - 1;
  while (left < right) {
   if (arr[left] + arr[right] < targetSum) {  // found the triplet
    // since arr[right] >= arr[left], therefore, we can replace arr[right] by any number between
    // left and right to get a sum less than the target sum
    count += right - left;
    left++;
   } else {
    right--;  // we need a pair with a smaller sum
   }
  }
  return count;
 }
};

int main(int argc, char *argv[]) {
 vector<int> vec = {-1, 0, 2, 3};
```

```
  cout << TripletWithSmallerSum::searchTriplets(vec, 3) << endl;
  vec = {-1, 4, 2, 1, 3};
  cout << TripletWithSmallerSum::searchTriplets(vec, 5) << endl;
}
```

Time complexity

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 $O(N^2)$.

Space complexity

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.

Similar Problems

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

Solution: Following a similar approach we can create a list containing all the triplets. Here is the code - only the highlighted lines have changed:

```
using namespace std;


#include <algorithm>
#include <iostream>
#include <string>
#include <vector>


class TripletWithSmallerSum {
 public:
  static vector<vector<int>> searchTriplets(vector<int> &arr, int target) {
    sort(arr.begin(), arr.end());
```

```
    vector<vector<int>> triplets;
    for (int i = 0; i < arr.size() - 2; i++) {
      searchPair(arr, target - arr[i], i, triplets);
    }
    return triplets;
  }

 private:
  static void searchPair(vector<int> &arr, int targetSum, int first,
                         vector<vector<int>> &triplets) {
    int left = first + 1, right = arr.size() - 1;
    while (left < right) {
      if (arr[left] + arr[right] < targetSum) {  // found the triplet
        // since arr[right] >= arr[left], therefore, we can replace arr[right] by any number between
        // left and right to get a sum less than the target sum
        for (int i = right; i > left; i--) {
          triplets.push_back({arr[first], arr[left], arr[i]});
        }
        left++;
      } else {
        right--;  // we need a pair with a smaller sum
      }
    }
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {-1, 0, 2, 3};
  auto result = TripletWithSmallerSum::searchTriplets(vec, 3);
```

```
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
  cout << endl;

  vec = {-1, 4, 2, 1, 3};
  result = TripletWithSmallerSum::searchTriplets(vec, 5);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
}
```

Another simpler approach could be to check every triplet of the array with three nested loops and create a list of triplets that meet the required condition.

Time complexity #

Sorting the array will take O(N * logN)$O(N*logN)$. The `searchPair()`, in this case, will take O(N^2)$O(N2)$; the main `while` loop will run in O(N)$O(N)$ but the nested `for` loop can also take O(N)$O(N)$ - 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 $O(N^3)$.

Space complexity

Ignoring the space required for the output array, the space complexity of the above algorithm will be $O(N)$ which is required for sorting.

Problem Statement

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

Example 1:

Input: [2, 5, 3, 10], target=30
Output: [2], [5], [2, 5], [3], [5, 3], [10]
Explanation: There are six contiguous subarrays whose product is less than the target.

Example 2:

Input: [8, 2, 6, 5], target=50
Output: [8], [2], [8, 2], [6], [2, 6], [5], [6, 5]
Explanation: There are seven contiguous subarrays whose product is less than the target.

Solution

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.

```
using namespace std;
```

```
#include <deque>
#include <iostream>
#include <vector>

class SubarrayProductLessThanK {
 public:
  static vector<vector<int>> findSubarrays(const vector<int>& arr, int target) {
    vector<vector<int>> result;
    int product = 1, left = 0;
    for (int right = 0; right < arr.size(); right++) {
      product *= arr[right];
      while (product >= target && left < arr.size()) {
        product /= arr[left++];
      }
      // since the product of all numbers from left to right is less than the target therefore,
      // all subarrays from left to right will have a product less than the target too; to avoid
      // duplicates, we will start with a subarray containing only arr[right] and then extend it
      deque<int> tempList;
      for (int i = right; i >= left; i--) {
        tempList.push_front(arr[i]);
        vector<int> resultVec;
        std::move(std::begin(tempList), std::end(tempList), std::back_inserter(resultVec));
        result.push_back(resultVec);
      }
    }
    return result;
  }
};
```

```
int main(int argc, char* argv[]) {
  auto result = SubarrayProductLessThanK::findSubarrays(vector<int>{2, 5, 3, 10}, 30);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
  cout << endl;

  result = SubarrayProductLessThanK::findSubarrays(vector<int>{8, 2, 6, 5}, 50);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
}
```

Time complexity

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 $O(N^3)$.

Space complexity

Ignoring the space required for the output list, the algorithm runs in $O(N)$ space which is used for the temp list.

Can you try estimating how much space will be required for the output list?

Show Hint

It is not all the Combinations of all elements of the array!

For an array with distinct elements, finding all of its contiguous subarrays is like finding the number of ways to choose two indices, `i` and `j`, in the array such that `i <= j`.

If there are a total of `n` elements in the array, here is how we can count all the contiguous subarrays:

- When `i = 0`, `j` can have any value from `0` to `n-1`, giving a total of `n` choices.
- When `i = 1`, `j` can have any value from `1` to `n-1`, giving a total of `n-1` choices.
- Similarly, when `i = 2`, `j` can have `n-2` choices.
  ...
  ...
- When `i = n-1`, `j` can only have only `1` choice.

Let's combine all the choices:

```
n + (n-1) + (n-2) + ... 3 + 2 + 1
```

Which gives us a total of: $n*(n+1)/2$

So, at most, we need space for $O(n^2)$ output lists. At worst, each subarray can take $O(n)$ space, so overall, our algorithm's space complexity will be $O(n^3)$.

Problem Statement

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.

Example 1:

Input: [1, 0, 2, 1, 0]
Output: [0 0 1 1 2]

Example 2:

Input: [2, 2, 0, 1, 2, 0]
Output: [0 0 1 2 2 2]

Solution

The brute force solution will be to use an in-place sorting algorithm like Heapsort which will take O(N*logN)$O(N*logN)$. 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 0s before `low` and all 2s after `high` so that in the end, all 1s will be between `low` and `high`.

```
using namespace std;


#include <iostream>

#include <vector>


class DutchFlag {

 public:

  static void sort(vector<int> &arr) {

    // all elements < low are 0 and all elements > high are 2

    // all elements from >= low < i are 1

    int low = 0, high = arr.size() - 1;

    for (int i = 0; i <= high;) {

      if (arr[i] == 0) {

        swap(arr, i, low);
```

```
      i++;
      low++;
    } else if (arr[i] == 1) {
      i++;
    } else {  // the case for arr[i] == 2
      swap(arr, i, high);
      // decrement 'high' only, after the swap the number at index 'i' could be 0, 1 or 2
      high--;
    }
  }
 }

 private:
  static void swap(vector<int> &arr, int i, int j) {
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> arr = {1, 0, 2, 1, 0};
  DutchFlag::sort(arr);
  for (auto num : arr) {
   cout << num << " ";
  }
  cout << endl;

  arr = vector<int>{2, 2, 0, 1, 2, 0};
```

```
  DutchFlag::sort(arr);
  for (auto num : arr) {
    cout << num << " ";
  }
}
```

Time complexity

The time complexity of the above algorithm will be $O(N)$ as we are iterating the input array only once.

Space complexity

The algorithm runs in constant space $O(1)$.

Quadruple Sum to Target (medium)

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

Example 1:

Input: [4, 1, 2, -1, 1, -3], target=1
Output: [-3, -1, 1, 4], [-3, 1, 1, 2]
Explanation: Both the quadruplets add up to the target.

Example 2:

Input: [2, 0, -1, 1, -2, 2], target=2
Output: [-2, 0, 2, 2], [-1, 0, 1, 2]
Explanation: Both the quadruplets add up to the target.

Solution

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.

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <vector>


class QuadrupleSumToTarget {

 public:

  static vector<vector<int>> searchQuadruplets(vector<int> &arr, int target) {

    sort(arr.begin(), arr.end());

    vector<vector<int>> quadruplets;

    for (int i = 0; i < arr.size() - 3; i++) {

      if (i > 0 && arr[i] == arr[i - 1]) {  // skip same element to avoid duplicate quadruplets

        continue;

      }

      for (int j = i + 1; j < arr.size() - 2; j++) {

        if (j > i + 1 &&

            arr[j] == arr[j - 1]) {  // skip same element to avoid duplicate quadruplets

          continue;

        }

        searchPairs(arr, target, i, j, quadruplets);

      }

    }

    return quadruplets;

  }
```

```
 private:
  static void searchPairs(const vector<int> &arr, int targetSum, int first, int second,
                          vector<vector<int>> &quadruplets) {
    int left = second + 1;
    int right = arr.size() - 1;
    while (left < right) {
      int sum = arr[first] + arr[second] + arr[left] + arr[right];
      if (sum == targetSum) {  // found the quadruplet
        quadruplets.push_back({arr[first], arr[second], arr[left], arr[right]});
        left++;
        right--;
        while (left < right && arr[left] == arr[left - 1]) {
          left++;  // skip same element to avoid duplicate quadruplets
        }
        while (left < right && arr[right] == arr[right + 1]) {
          right--;  // skip same element to avoid duplicate quadruplets
        }
      } else if (sum < targetSum) {
        left++;  // we need a pair with a bigger sum
      } else {
        right--;  // we need a pair with a smaller sum
      }
    }
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {4, 1, 2, -1, 1, -3};
```

```
 auto result = QuadrupleSumToTarget::searchQuadruplets(vec, 1);
 for (auto vec : result) {
  cout << "[";
  for (auto num : vec) {
   cout << num << " ";
  }
  cout << "]";
 }
 cout << endl;

 vec = {2, 0, -1, 1, -2, 2};
 result = QuadrupleSumToTarget::searchQuadruplets(vec, 2);
 for (auto vec : result) {
  cout << "[";
  for (auto num : vec) {
   cout << num << " ";
  }
  cout << "]";
 }
}
```

Time complexity

Sorting the array will take $O(N*logN)$.
Overall `searchQuadruplets()` will take $O(N * logN + N^3)$, which is asymptotically equivalent to $O(N^3)$.

Space complexity

The space complexity of the above algorithm will be $O(N)$ which is required for sorting.

Comparing Strings containing Backspaces (medium)

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

Example 1:

Input: str1="xy#z", str2="xzz#"
Output: true
Explanation: After applying backspaces the strings become "xz" and "xz" respectively.

Example 2:

Input: str1="xy#z", str2="xyz#"
Output: false
Explanation: After applying backspaces the strings become "xz" and "xy" respectively.

Example 3:

Input: str1="xp#", str2="xyz##"
Output: true
Explanation: After applying backspaces the strings become "x" and "x" respectively.
In "xyz##", the first '#' removes the character 'z' and the second '#' removes the character 'y'.

Example 4:

Input: str1="xywrrmp", str2="xywrrmu#p"
Output: true
Explanation: After applying backspaces the strings become "xywrrmp" and "xywrrmp" respectively.

Solution

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.

```
using namespace std;
```

```
#include <iostream>
#include <string>

class BackspaceCompare {
 public:
  static bool compare(const string &str1, const string &str2) {
    // use two pointer approach to compare the strings
    int index1 = str1.length() - 1, index2 = str2.length() - 1;
    while (index1 >= 0 || index2 >= 0) {
      int i1 = getNextValidCharIndex(str1, index1);
      int i2 = getNextValidCharIndex(str2, index2);

      if (i1 < 0 && i2 < 0) {  // reached the end of both the strings
        return true;
      }

      if (i1 < 0 || i2 < 0) {  // reached the end of one of the strings
        return false;
      }

      if (str1[i1] != str2[i2]) {  // check if the characters are equal
        return false;
      }

      index1 = i1 - 1;
      index2 = i2 - 1;
    }

    return true;
```

```
  }

 private:
  static int getNextValidCharIndex(const string &str, int index) {
    int backspaceCount = 0;
    while (index >= 0) {
      if (str[index] == '#') {  // found a backspace character
        backspaceCount++;
      } else if (backspaceCount > 0) {  // a non-backspace character
        backspaceCount--;
      } else {
        break;
      }

      index--;  // skip a backspace or a valid character
    }
    return index;
  }
};

int main(int argc, char *argv[]) {
  cout << BackspaceCompare::compare("xy#z", "xzz#") << endl;
  cout << BackspaceCompare::compare("xy#z", "xyz#") << endl;
  cout << BackspaceCompare::compare("xp#", "xyz##") << endl;
  cout << BackspaceCompare::compare("xywrrmp", "xywrrmu#p") << endl;
}
```

Time complexity

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.

Space complexity

The algorithm runs in constant space $O(1)$.

Minimum Window Sort (medium)

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

Example 1:

Input: [1, 2, 5, 3, 7, 10, 9, 12]
Output: 5
Explanation: We need to sort only the subarray [5, 3, 7, 10, 9] to make the whole array sorted

Example 2:

Input: [1, 3, 2, 0, -1, 7, 10]
Output: 5
Explanation: We need to sort only the subarray [1, 3, 2, 0, -1] to make the whole array sorted

Example 3:

Input: [1, 2, 3]
Output: 0
Explanation: The array is already sorted

Example 4:

Input: [3, 2, 1]
Output: 3
Explanation: The whole array needs to be sorted.

Solution

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]
```

1. 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'.
2. 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.

```
using namespace std;


#include <iostream>
```

```
#include <limits>
#include <vector>

class ShortestWindowSort {
 public:
  static int sort(const vector<int>& arr) {
    int low = 0, high = arr.size() - 1;
    // find the first number out of sorting order from the beginning
    while (low < arr.size() - 1 && arr[low] <= arr[low + 1]) {
      low++;
    }

    if (low == arr.size() - 1) {  // if the array is 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 maximum and minimum of the subarray
    int subarrayMax = numeric_limits<int>::min(), subarrayMin = numeric_limits<int>::max();
    for (int k = low; k <= high; k++) {
      subarrayMax = max(subarrayMax, arr[k]);
      subarrayMin = min(subarrayMin, arr[k]);
    }

    // extend the subarray to include any number which is bigger than the minimum of the subarray
```

```
    while (low > 0 && arr[low - 1] > subarrayMin) {
      low--;
    }
    // extend the subarray to include any number which is smaller than the maximum of the subarray
    while (high < arr.size() - 1 && arr[high + 1] < subarrayMax) {
      high++;
    }

    return high - low + 1;
  }
};

int main(int argc, char* argv[]) {
  cout << ShortestWindowSort::sort(vector<int>{1, 2, 5, 3, 7, 10, 9, 12}) << endl;
  cout << ShortestWindowSort::sort(vector<int>{1, 3, 2, 0, -1, 7, 10}) << endl;
  cout << ShortestWindowSort::sort(vector<int>{1, 2, 3}) << endl;
  cout << ShortestWindowSort::sort(vector<int>{3, 2, 1}) << endl;
}
```

Time complexity

The time complexity of the above algorithm will be $O(N)$.

Space complexity

The algorithm runs in constant space $O(1)$.

1. Introduction

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.

Problem Statement

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

Example: head Following LinkedList has a cycle: Following LinkedList doesn't have a cycle: 2 4 6 8 10 null

Solution

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. A similar analysis can be done where the slow pointer moves first. Here is a visual representation of the above discussion:

```
using namespace std;


#include <iostream>


class ListNode {

 public:

  int value = 0;

  ListNode *next;


  ListNode(int value) {
```

```
    this->value = value;
    next = nullptr;
  }
};

class LinkedListCycle {
 public:
  static bool hasCycle(ListNode *head) {
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      fast = fast->next->next;
      slow = slow->next;
      if (slow == fast) {
        return true;  // found the cycle
      }
    }
    return false;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;
```

```
  head->next->next->next->next->next->next = head->next->next;
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;


  head->next->next->next->next->next->next = head->next->next->next;
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;
}
```

Time Complexity

As we have concluded above, 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.

Space Complexity

The algorithm runs in constant space $O(1)$.

Similar Problems

Problem 1: Given the head of a LinkedList with a cycle, find the length of the cycle.

Solution: We can use the above solution to find the cycle in the LinkedList. 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.

Here is what our algorithm will look like:

```
using namespace std;


#include <iostream>
```

```
class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class LinkedListCycleLength {
 public:
  static int findCycleLength(ListNode *head) {
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      fast = fast->next->next;
      slow = slow->next;
      if (slow == fast)  // found the cycle
      {
        return calculateLength(slow);
      }
    }
    return 0;
  }

 private:
```

```
  static int calculateLength(ListNode *slow) {
    ListNode *current = slow;
    int cycleLength = 0;
    do {
      current = current->next;
      cycleLength++;
    } while (current != slow);
    return cycleLength;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);
  head->next->next->next->next->next->next = head->next->next;
  cout << "LinkedList cycle length: " << LinkedListCycleLength::findCycleLength(head) << endl;

  head->next->next->next->next->next->next = head->next->next->next;
  cout << "LinkedList cycle length: " << LinkedListCycleLength::findCycleLength(head) << endl;
}
```

Time and Space Complexity: The above algorithm runs in O(N)$O(N)$ time complexity and O(1)$O(1)$ space complexity.

Problem Statement

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

head 1 2 3 4 5 6 Cycle start Examples: Cycle start 1 2 3 4 5 6 1 2 3 4 5 6

Solution

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.

Let's visually see this with the above-mentioned Example-1:

1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 pointer1 pointer2 pointer2 pointer1 pointer1, pointer2 1 2 3 4 5 6 pointer1, pointer2 -Keep incrementing both pointers until they meet -Move pointer2 '4' nodes ahead

We can use the algorithm discussed in LinkedList Cycle to find the length of the cycle and then follow the above-mentioned steps to find the start of the cycle.

```
using namespace std;


#include <iostream>


class ListNode {

 public:
```

```
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class LinkedListCycleStart {
 public:
  static ListNode *findCycleStart(ListNode *head) {
    int cycleLength = 0;
    // find the LinkedList cycle
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      fast = fast->next->next;
      slow = slow->next;
      if (slow == fast) {  // found the cycle
        cycleLength = calculateCycleLength(slow);
        break;
      }
    }

    return findStart(head, cycleLength);
  }

 private:
```

```
  static int calculateCycleLength(ListNode *slow) {
    ListNode *current = slow;
    int cycleLength = 0;
    do {
      current = current->next;
      cycleLength++;
    } while (current != slow);

    return cycleLength;
  }

  static ListNode *findStart(ListNode *head, int cycleLength) {
    ListNode *pointer1 = head, *pointer2 = head;
    // move pointer2 ahead '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;
  }
};
```

```
int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);

  head->next->next->next->next->next->next = head->next->next;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;

  head->next->next->next->next->next->next = head->next->next->next;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;

  head->next->next->next->next->next->next = head;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;
}
```

Time Complexity

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 $O(N)$.

Space Complexity

The algorithm runs in constant space $O(1)$.

Problem Statement

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'.

Example 1:

Input: 23
Output: true (23 is a happy number)
Explanations: Here are the steps to find out that 23 is a happy number:

1. 2^2 + 3 ^22$_2$+3$_2$ = 4 + 9 = 13
2. 1^2 + 3^21$_2$+3$_2$ = 1 + 9 = 10
3. 1^2 + 0^21$_2$+0$_2$ = 1 + 0 = 1

Example 2:

Input: 12
Output: false (12 is not a happy number)
Explanations: Here are the steps to find out that 12 is not a happy number:

1. 1^2 + 2 ^21$_2$+2$_2$ = 1 + 4 = 5
2. 5^25$_2$ = 25
3. 2^2 + 5^22$_2$+5$_2$ = 4 + 25 = 29
4. 2^2 + 9^22$_2$+9$_2$ = 4 + 81 = 85
5. 8^2 + 5^28$_2$+5$_2$ = 64 + 25 = 89
6. 8^2 + 9^28$_2$+9$_2$ = 64 + 81 = 145
7. 1^2 + 4^2 + 5^21$_2$+4$_2$+5$_2$ = 1 + 16 + 25 = 42
8. 4^2 + 2^24$_2$+2$_2$ = 16 + 4 = 20
9. 2^2 + 0^22$_2$+0$_2$ = 4 + 0 = 4
10. 4^24$_2$ = 16
11. 1^2 + 6^21$_2$+6$_2$ = 1 + 36 = 37
12. 3^2 + 7^23$_2$+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.

Solution

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.

```
using namespace std;


#include <iostream>


class HappyNumber {
 public:
  static bool find(int num) {
    int slow = num, fast = num;
    do {
      slow = findSquareSum(slow);               // move one step
      fast = findSquareSum(findSquareSum(fast));  // move two steps
    } while (slow != fast);                     // found the cycle
```

```
    return slow == 1;  // see if the cycle is stuck on the number '1'
  }

 private:
  static int findSquareSum(int num) {
    int sum = 0, digit;
    while (num > 0) {
      digit = num % 10;
      sum += digit * digit;
      num /= 10;
    }
    return sum;
  }
};

int main(int argc, char* argv[]) {
  cout << HappyNumber::find(23) << endl;
  cout << HappyNumber::find(12) << endl;
}
```

Time Complexity

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*N* is less than or equal to 1000, then we reach the cycle or '1' in at most 1001 steps.
2. For N > 1000*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 $9\text{^}2M9^2M$, or $81M81M$ (this will happen when all digits of 'N' are '9').

This means:

1. $N1 < 81MN1<81M$
2. As we know $M = \log(N+1)M=log(N+1)$
3. Therefore: $N1 < 81 * \log(N+1)N1<81*log(N+1)$ => $N1 = O(\log N)N1=O(logN)$

This concludes that the above algorithm will have a time complexity of $O(\log N)O(logN)$.

Space Complexity

The algorithm runs in constant space $O(1)O(1)$.

Problem Statement

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.

Example 1:

Input: 1 -> 2 -> 3 -> 4 -> 5 -> null
Output: 3

Example 2:

Input: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> null
Output: 4

Example 3:

Input: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> null
Output: 4

Solution

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.

```
using namespace std;


#include <iostream>


class ListNode {
 public:
  int value = 0;
  ListNode *next;


  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};


class MiddleOfLinkedList {
 public:
  static ListNode *findMiddle(ListNode *head) {
    ListNode *slow = head;
    ListNode *fast = head;
```

```
    while (fast != nullptr && fast->next != nullptr) {
      slow = slow->next;
      fast = fast->next->next;
    }

    return slow;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;

  head->next->next->next->next->next = new ListNode(6);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;

  head->next->next->next->next->next->next = new ListNode(7);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;
}
```

Time complexity

The above algorithm will have a time complexity of $O(N)$ where 'N' is the number of nodes in the LinkedList.

Space complexity

The algorithm runs in constant space $O(1)$.

Palindrome LinkedList (medium)

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

Solution

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.

```
using namespace std;


#include <iostream>


class ListNode {
 public:
  int value = 0;
  ListNode *next;


  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};


class PalindromicLinkedList {
 public:
  static bool isPalindrome(ListNode *head) {
    if (head == nullptr || head->next == nullptr) {
      return true;
    }


    // find middle of the LinkedList
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
```

```
    slow = slow->next;
    fast = fast->next->next;
  }

  ListNode *headSecondHalf = reverse(slow);  // reverse the second half
  ListNode *copyHeadSecondHalf =
      headSecondHalf;  // store the head of reversed part to revert back later

  // compare the first and the second half
  while (head != nullptr && headSecondHalf != nullptr) {
    if (head->value != headSecondHalf->value) {
      break;  // not a palindrome
    }
    head = head->next;
    headSecondHalf = headSecondHalf->next;
  }

  reverse(copyHeadSecondHalf);                    // revert the reverse of the second half
  if (head == nullptr || headSecondHalf == nullptr) {  // if both halves match
    return true;
  }
  return false;
}

private:
 static ListNode *reverse(ListNode *head) {
  ListNode *prev = nullptr;
  while (head != nullptr) {
    ListNode *next = head->next;
```

```
    head->next = prev;
    prev = head;
    head = next;
   }
   return prev;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(2);
  head->next = new ListNode(4);
  head->next->next = new ListNode(6);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(2);
  cout << "Is palindrome: " << PalindromicLinkedList::isPalindrome(head) << endl;

  head->next->next->next->next->next = new ListNode(2);
  cout << "Is palindrome: " << PalindromicLinkedList::isPalindrome(head) << endl;
}
```

Time complexity

The above algorithm will have a time complexity of $O(N)$ where 'N' is the number of nodes in the LinkedList.

Space complexity

The algorithm runs in constant space $O(1)$.

Rearrange a LinkedList (medium)

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

Solution

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.

```
using namespace std;


#include <iostream>


class ListNode {
```

```
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class RearrangeList {
 public:
  static void reorder(ListNode *head) {
    if (head == nullptr || head->next == nullptr) {
      return;
    }

    // find the middle of the LinkedList
    ListNode *slow = head, *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      slow = slow->next;
      fast = fast->next->next;
    }

    // slow is now pointing to the middle node
    ListNode *headSecondHalf = reverse(slow);  // reverse the second half
    ListNode *headFirstHalf = head;

    // rearrange to produce the LinkedList in the required order
```

```
    while (headFirstHalf != nullptr && headSecondHalf != nullptr) {
      ListNode *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 != nullptr) {
      headFirstHalf->next = nullptr;
    }
  }

 private:
  static ListNode *reverse(ListNode *head) {
    ListNode *prev = nullptr;
    while (head != nullptr) {
      ListNode *next = head->next;
      head->next = prev;
      prev = head;
      head = next;
    }
    return prev;
  }
};
```

```
int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(2);
  head->next = new ListNode(4);
  head->next->next = new ListNode(6);
  head->next->next->next = new ListNode(8);
  head->next->next->next->next = new ListNode(10);
  head->next->next->next->next->next = new ListNode(12);
  RearrangeList::reorder(head);
  while (head != nullptr) {
    cout << head->value << " ";
    head = head->next;
  }
}
```

Time Complexity

The above algorithm will have a time complexity of $O(N)$ where 'N' is the number of nodes in the LinkedList.

Space Complexity

The algorithm runs in constant space $O(1)$.

Cycle in a Circular Array (hard)

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.

Solution

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.

```
using namespace std;


#include <iostream>

#include <vector>


class CircularArrayLoop {

 public:

  static bool loopExists(const vector<int> &arr) {

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

      bool isForward = arr[i] >= 0;  // if we are moving forward or not

      int slow = i, fast = i;


      // if slow or fast becomes '-1' this means we can't find cycle for this number

      do {

        slow = findNextIndex(arr, isForward, slow);  // move one step for slow pointer

        fast = findNextIndex(arr, isForward, fast);  // move one step for fast pointer

        if (fast != -1) {

          fast = findNextIndex(arr, isForward, fast);  // move another step for fast pointer

        }

      } while (slow != -1 && fast != -1 && slow != fast);


      if (slow != -1 && slow == fast) {
```

```
      return true;
    }
   }

   return false;
  }

 private:
  static int findNextIndex(const vector<int> &arr, bool isForward, int currentIndex) {
   bool direction = arr[currentIndex] >= 0;
   if (isForward != direction) {
    return -1;  // change in direction, return -1
   }

   // wrap around for negative numbers
   int nextIndex = (currentIndex + arr[currentIndex] + arr.size()) % arr.size();

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

   return nextIndex;
  }
};

int main(int argc, char *argv[]) {
  cout << CircularArrayLoop::loopExists(vector<int>{1, 2, -1, 2, 2}) << endl;
  cout << CircularArrayLoop::loopExists(vector<int>{2, 2, -1, 2}) << endl;
```

```
  cout << CircularArrayLoop::loopExists(vector<int>{2, 1, -1, -2}) << endl;
}
```

Time Complexity

The above algorithm will have a time complexity of $O(N^2)$ 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.

Space Complexity

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 the space complexity of our algorithm will increase to $O(N)$.

1. Introduction

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:

Understanding the above six cases will help us in solving all intervals related problems. Let's jump onto our first problem to understand the **Merge Interval** pattern.

Problem Statement

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

Example 1:

Intervals: [[1,4], [2,5], [7,9]]
Output: [[1,5], [7,9]]
Explanation: Since the first two intervals [1,4] and [2,5] overlap, we merged them into one [1,5].

Example 2:

Intervals: [[6,7], [2,4], [5,9]]
Output: [[2,4], [5,9]]
Explanation: Since the intervals [6,7] and [5,9] overlap, we merged them into one [5,9].

Example 3:

Intervals: [[1,4], [2,6], [3,5]]
Output: [[1,6]]
Explanation: Since all the given intervals overlap, we merged them into one.

Solution

Let's take the example of two intervals ('a' and 'b') such that `a.start <= b.start`. There are four possible scenarios:

Our goal is to merge the intervals whenever they overlap. For the above-mentioned three overlapping scenarios (2, 3, and 4), this is how we will merge them:

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)
```

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

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <vector>


class Interval {

 public:

  int start = 0;

  int end = 0;


  Interval(int start, int end) {

    this->start = start;

    this->end = end;

  }

};


class MergeIntervals {

 public:

  static vector<Interval> merge(vector<Interval> &intervals) {
```

```
    if (intervals.size() < 2) {
      return intervals;
    }

    // sort the intervals by start time
    sort(intervals.begin(), intervals.end(),
        [](const Interval &x, const Interval &y) { return x.start < y.start; });

    vector<Interval> mergedIntervals;

    vector<Interval>::const_iterator intervalItr = intervals.begin();
    Interval interval = *intervalItr++;
    int start = interval.start;
    int end = interval.end;
    while (intervalItr != intervals.end()) {
      interval = *intervalItr++;
      if (interval.start <= end) {  // overlapping intervals, adjust the 'end'
        end = max(interval.end, end);
      } else {  // non-overlapping interval, add the previous interval and reset
        mergedIntervals.push_back({start, end});
        start = interval.start;
        end = interval.end;
      }
    }
    // add the last interval
    mergedIntervals.push_back({start, end});
    return mergedIntervals;
  }
};
```

```
int main(int argc, char *argv[]) {
  vector<Interval> input = {{1, 3}, {2, 5}, {7, 9}};
  cout << "Merged intervals: ";
  for (auto interval : MergeIntervals::merge(input)) {
    cout << "[" << interval.start << "," << interval.end << "] ";
  }
  cout << endl;

  input = {{6, 7}, {2, 4}, {5, 9}};
  cout << "Merged intervals: ";
  for (auto interval : MergeIntervals::merge(input)) {
    cout << "[" << interval.start << "," << interval.end << "] ";
  }
  cout << endl;

  input = {{1, 4}, {2, 6}, {3, 5}};
  cout << "Merged intervals: ";
  for (auto interval : MergeIntervals::merge(input)) {
    cout << "[" << interval.start << "," << interval.end << "] ";
  }
  cout << endl;
}
```

Time complexity

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 $O(N * logN)$.

Space complexity

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. For Java, depending on its version, `Collections.sort()` either uses Merge sort or Timsort, and both these algorithms need $O(N)$ space. Overall, our algorithm has a space complexity of $O(N)$.

Similar Problems

Problem 1: Given a set of intervals, find out if any two intervals overlap.

Example:

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

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

Problem Statement

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.

Example 1:

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 one [4,7].

Example 2:

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 them into [4,12].

Example 3:

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 one [1,4].

Solution

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)$O(N*logN)$

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.

```
using namespace std;


#include <iostream>

#include <vector>


class Interval {

 public:

  int start = 0;

  int end = 0;


  Interval(int start, int end) {

    this->start = start;

    this->end = end;

  }

};


class InsertInterval {

 public:

  static vector<Interval> insert(const vector<Interval> &intervals, Interval newInterval) {

    if (intervals.empty()) {

      return vector<Interval>{newInterval};

    }
```

```
    vector<Interval> mergedIntervals;

    int i = 0;
    // skip (and add to output) all intervals that come before the 'newInterval'
    while (i < intervals.size() && intervals[i].end < newInterval.start) {
      mergedIntervals.push_back(intervals[i++]);
    }

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

    // insert the newInterval
    mergedIntervals.push_back(newInterval);

    // add all the remaining intervals to the output
    while (i < intervals.size()) {
      mergedIntervals.push_back(intervals[i++]);
    }

    return mergedIntervals;
  }
};

int main(int argc, char *argv[]) {
```

```
  vector<Interval> input = {{1, 3}, {5, 7}, {8, 12}};
  cout << "Intervals after inserting the new interval: ";
  for (auto interval : InsertInterval::insert(input, {4, 6})) {
    cout << "[" << interval.start << "," << interval.end << "] ";
  }
  cout << endl;

  cout << "Intervals after inserting the new interval: ";
  for (auto interval : InsertInterval::insert(input, {4, 10})) {
    cout << "[" << interval.start << "," << interval.end << "] ";
  }
  cout << endl;

  input = {{2, 3}, {5, 7}};
  cout << "Intervals after inserting the new interval: ";
  for (auto interval : InsertInterval::insert(input, {1, 4})) {
    cout << "[" << interval.start << "," << interval.end << "] ";
  }
  cout << endl;
}
```

Time complexity

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.

Space complexity

The space complexity of the above algorithm will be $O(N)$ as we need to return a list containing all the merged intervals.

Problem Statement

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

Example 1:

Input: arr1=[[1, 3], [5, 6], [7, 9]], arr2=[[2, 3], [5, 7]]
Output: [2, 3], [5, 6], [7, 7]
Explanation: The output list contains the common intervals between the two lists.

Example 2:

Input: arr1=[[1, 3], [5, 7], [9, 12]], arr2=[[5, 10]]
Output: [5, 7], [9, 10]
Explanation: The output list contains the common intervals between the two lists.

Solution

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.

```
using namespace std;

#include <iostream>
#include <vector>

class Interval {
 public:
  int start = 0;
  int end = 0;

  Interval(int start, int end) {
    this->start = start;
    this->end = end;
  }
};

class IntervalsIntersection {
 public:
  static vector<Interval> merge(const vector<Interval> &arr1, const vector<Interval> &arr2) {
    vector<Interval> result;
    int i = 0, j = 0;
    while (i < arr1.size() && j < arr2.size()) {
      // check if the interval arr[i] intersects with arr2[j]
      // check if one of the interval's start time lies within the other interval
      if ((arr1[i].start >= arr2[j].start && arr1[i].start <= arr2[j].end) ||
          (arr2[j].start >= arr1[i].start && arr2[j].start <= arr1[i].end)) {
        // store the intersection part
```

```
      result.push_back({max(arr1[i].start, arr2[j].start), min(arr1[i].end, arr2[j].end)});
    }

    // move next from the interval which is finishing first
    if (arr1[i].end < arr2[j].end) {
      i++;
    } else {
      j++;
    }
  }

  return result;
 }
};

int main(int argc, char *argv[]) {
 vector<Interval> input1 = {{1, 3}, {5, 6}, {7, 9}};
 vector<Interval> input2 = {{2, 3}, {5, 7}};
 vector<Interval> result = IntervalsIntersection::merge(input1, input2);
 cout << "Intervals Intersection: ";
 for (auto interval : result) {
  cout << "[" << interval.start << "," << interval.end << "] ";
 }
 cout << endl;

 input1 = {{1, 3}, {5, 7}, {9, 12}};
 input2 = {{5, 10}};
 result = IntervalsIntersection::merge(input1, input2);
 cout << "Intervals Intersection: ";
```

```
  for (auto interval : result) {
    cout << "[" << interval.start << "," << interval.end << "] ";
  }
}
```

Time complexity

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.

Space complexity

Ignoring the space needed for the result list, the algorithm runs in constant space $O(1)$.

Problem Statement

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

Example 1:

Appointments: [[1,4], [2,5], [7,9]]
Output: false
Explanation: Since [1,4] and [2,5] overlap, a person cannot attend both of these appointments.

Example 2:

Appointments: [[6,7], [2,4], [8,12]]
Output: true
Explanation: None of the appointments overlap, therefore a person can attend all of them.

Example 3:

Appointments: [[4,5], [2,3], [3,6]]
Output: false
Explanation: Since [4,5] and [3,6] overlap, a person cannot attend both of these appointments.

Solution

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.

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <vector>


class Interval {

 public:

  int start;

  int end;


  Interval(int start, int end) {

    this->start = start;

    this->end = end;

  }

};


class ConflictingAppointments {

 public:

  static bool canAttendAllAppointments(vector<Interval>& intervals) {

    // sort the intervals by start time

    sort(intervals.begin(), intervals.end(),

        [](const Interval& x, const Interval& y) { return x.start < y.start; });
```

```
    // find any overlapping appointment
    for (int i = 1; i < intervals.size(); i++) {
      if (intervals[i].start < intervals[i - 1].end) {
        // please note the comparison above, it is "<" and not "<="
        // while merging we needed "<=" comparison, as we will be merging the two
        // intervals having condition "intervals[i].start == intervals[i - 1].end" but
        // such intervals don't represent conflicting appointments as one starts right
        // after the other
        return false;
      }
    }
    return true;
  }
};

int main(int argc, char* argv[]) {
  vector<Interval> intervals = {{1, 4}, {2, 5}, {7, 9}};
  bool result = ConflictingAppointments::canAttendAllAppointments(intervals);
  cout << "Can attend all appointments: " << result << endl;

  intervals = {{6, 7}, {2, 4}, {8, 12}};
  result = ConflictingAppointments::canAttendAllAppointments(intervals);
  cout << "Can attend all appointments: " << result << endl;

  intervals = {{4, 5}, {2, 3}, {3, 6}};
  result = ConflictingAppointments::canAttendAllAppointments(intervals);
  cout << "Can attend all appointments: " << result << endl;
}
```

Time complexity

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.

Space complexity

The space complexity of the above algorithm will be $O(N)$, which we need for sorting. For Java, `Arrays.sort()` uses Timsort, which needs $O(N)$ space.

Similar Problems

Problem 1: Given a list of appointments, find all the conflicting appointments.

Example:

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

Minimum Meeting Rooms (hard)

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 1:

Meetings: [[1,4], [2,5], [7,9]]
Output: 2
Explanation: Since [1,4] and [2,5] overlap, we need two rooms to hold these two meetings. [7,9] can occur in any of the two rooms later.

Example 2:

Meetings: [[6,7], [2,4], [8,12]]
Output: 1
Explanation: None of the meetings overlap, therefore we only need one room to hold all meetings.

Example 3:

Meetings: [[1,4], [2,3], [3,6]]
Output:2
Explanation: Since [1,4] overlaps with the other two meetings [2,3] and [3,6], we need two rooms to hold all the meetings.

Example 4:

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 [4,5].

Here is a visual representation of Example 4:

Solution

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

Meetings: [[4,5], [2,3], [2,4], [3,5]]

Step 1: Sorting these meetings on their start time will give us: [[2,3], [2,4], [3,5], [4,5]]

Step 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.

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>

class Meeting {
 public:
  int start = 0;
  int end = 0;

  Meeting(int start, int end) {
    this->start = start;
    this->end = end;
  }
};

class MinimumMeetingRooms {
 public:
  struct endCompare {
    bool operator()(const Meeting &x, const Meeting &y) { return x.end > y.end; }
  };

  static int findMinimumMeetingRooms(vector<Meeting> &meetings) {
    if (meetings.empty()) {
```

```
    return 0;
  }

  // sort the meetings by start time
  sort(meetings.begin(), meetings.end(),
      [](const Meeting &x, const Meeting &y) { return x.start < y.start; });

  int minRooms = 0;
  priority_queue<Meeting, vector<Meeting>, endCompare> minHeap;
  for (auto meeting : meetings) {
    // remove all meetings that have ended
    while (!minHeap.empty() && meeting.start >= minHeap.top().end) {
      minHeap.pop();
    }
    // add the current meeting into the minHeap
    minHeap.push(meeting);
    // all active meeting are in the minHeap, so we need rooms for all of them.
    minRooms = max(minRooms, (int)minHeap.size());
  }

  return minRooms;
 }
};

int main(int argc, char *argv[]) {
 vector<Meeting> input = {{4, 5}, {2, 3}, {2, 4}, {3, 5}};
 int result = MinimumMeetingRooms::findMinimumMeetingRooms(input);
 cout << "Minimum meeting rooms required: " << result << endl;
```

```
  input = {{1, 4}, {2, 5}, {7, 9}};
  result = MinimumMeetingRooms::findMinimumMeetingRooms(input);
  cout << "Minimum meeting rooms required: " << result << endl;

  input = {{6, 7}, {2, 4}, {8, 12}};
  result = MinimumMeetingRooms::findMinimumMeetingRooms(input);
  cout << "Minimum meeting rooms required: " << result << endl;

  input = {{1, 4}, {2, 3}, {3, 6}};
  result = MinimumMeetingRooms::findMinimumMeetingRooms(input);
  cout << "Minimum meeting rooms required: " << result << endl;

  input = {{4, 5}, {2, 3}, {2, 4}, {3, 5}};
  result = MinimumMeetingRooms::findMinimumMeetingRooms(input);
  cout << "Minimum meeting rooms required: " << result << endl;
}
[
```

Time complexity

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 $O(NlogN)$.

Space complexity

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

Problem 1: Given a list of intervals, find the point where the maximum number of intervals overlap.

Problem 2: 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)

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 when both the 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 job which is 15.

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 [3,4].

Solution

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.

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <queue>

#include <vector>


class Job {

 public:

  int start = 0;

  int end = 0;

  int cpuLoad = 0;


  Job(int start, int end, int cpuLoad) {

    this->start = start;

    this->end = end;

    this->cpuLoad = cpuLoad;

  }

};


class MaximumCPULoad {
```

```
public:
 struct endCompare {
  bool operator()(const Job &x, const Job &y) { return x.end > y.end; }
 };

 static int findMaxCPULoad(vector<Job> &jobs) {
  if (jobs.empty()) {
   return 0;
  }

  // sort the jobs by start time
  sort(jobs.begin(), jobs.end(), [](const Job &a, const Job &b) { return a.start < b.start; });

  int maxCPULoad = 0;
  int currentCPULoad = 0;
  priority_queue<Job, vector<Job>, endCompare> minHeap;
  for (auto job : jobs) {
   // remove all jobs that have ended
   while (!minHeap.empty() && job.start > minHeap.top().end) {
    currentCPULoad -= minHeap.top().cpuLoad;
    minHeap.pop();
   }

   // add the current job into the minHeap
   minHeap.push(job);
   currentCPULoad += job.cpuLoad;
   maxCPULoad = max(maxCPULoad, currentCPULoad);
  }
```

```
    return maxCPULoad;
  }
};

int main(int argc, char *argv[]) {
  vector<Job> input = {{1, 4, 3}, {7, 9, 6}, {2, 5, 4}};
  cout << "Maximum CPU load at any time: " << MaximumCPULoad::findMaxCPULoad(input) << endl;

  input = {{6, 7, 10}, {8, 12, 15}, {2, 4, 11}};
  cout << "Maximum CPU load at any time: " << MaximumCPULoad::findMaxCPULoad(input) << endl;

  input = {{1, 4, 2}, {3, 6, 5}, {2, 4, 1}};
  cout << "Maximum CPU load at any time: " << MaximumCPULoad::findMaxCPULoad(input) << endl;
}
```

Time complexity

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 $O(NlogN)$.

Space complexity

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)

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.

Example 1:

Input: Employee Working Hours=[[[1,3], [5,6]], [[2,3], [6,8]]]
Output: [3,5]
Explanation: Both the employees are free between [3,5].

Example 2:

Input: Employee Working Hours=[[[1,3], [9,12]], [[2,4]], [[6,8]]]
Output: [4,6], [8,9]
Explanation: All employees are free between [4,6] and [8,9].

Example 3:

Input: Employee Working Hours=[[[1,3]], [[2,4]], [[3,5], [7,9]]]
Output: [5,7]
Explanation: All employees are free between [5,7].

Solution

This problem follows the Merge Intervals pattern. Let's take the above-mentioned example (2) and visually draw it:

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]). 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?

Using a Heap to Sort the Intervals

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.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class Interval {

 public:

  int start = 0;

  int end = 0;


  Interval(int start, int end) {

    this->start = start;

    this->end = end;

  }
```

```
};

class EmployeeFreeTime {
 public:
  struct startCompare {
    bool operator()(const pair<Interval, pair<int, int>> &x,
                    const pair<Interval, pair<int, int>> &y) {
      return x.first.start > y.first.start;
    }
  };

  static vector<Interval> findEmployeeFreeTime(const vector<vector<Interval>> &schedule) {
    vector<Interval> result;
    if (schedule.empty()) {
      return result;
    }

    // PriorityQueue to store one interval from each employee
    priority_queue<pair<Interval, pair<int, int>>, vector<pair<Interval, pair<int, int>>>,
                   startCompare>
        minHeap;

    // insert the first interval of each employee to the queue
    for (int i = 0; i < schedule.size(); i++) {
      minHeap.push(make_pair(schedule[i][0], make_pair(i, 0)));
    }

    Interval previousInterval = minHeap.top().first;
    while (!minHeap.empty()) {
```

```
    auto queueTop = minHeap.top();
    minHeap.pop();
    // if previousInterval is not overlapping with the next interval, insert a free interval
    if (previousInterval.end < queueTop.first.start) {
      result.push_back({previousInterval.end, queueTop.first.start});
      previousInterval = queueTop.first;
    } else {  // overlapping intervals, update the previousInterval if needed
      if (previousInterval.end < queueTop.first.end) {
        previousInterval = queueTop.first;
      }
    }

    // if there are more intervals available for the same employee, add their next interval
    vector<Interval> employeeSchedule = schedule[queueTop.second.first];
    if (employeeSchedule.size() > queueTop.second.second + 1) {
      minHeap.push(make_pair(employeeSchedule[queueTop.second.second + 1],
                   make_pair(queueTop.second.first, queueTop.second.second + 1)));
    }
   }

   return result;
  }
};

int main(int argc, char *argv[]) {
  vector<vector<Interval>> input = {{{1, 3}, {5, 6}}, {{2, 3}, {6, 8}}};
  vector<Interval> result = EmployeeFreeTime::findEmployeeFreeTime(input);
  cout << "Free intervals: ";
  for (auto interval : result) {
```

```
  cout << "[" << interval.start << ", " << interval.end << "] ";
 }
 cout << endl;

 input = {{{1, 3}, {9, 12}}, {{2, 4}}, {{6, 8}}};
 result = EmployeeFreeTime::findEmployeeFreeTime(input);
 cout << "Free intervals: ";
 for (auto interval : result) {
  cout << "[" << interval.start << ", " << interval.end << "] ";
 }
 cout << endl;

 input = {{{1, 3}}, {{2, 4}}, {{3, 5}, {7, 9}}};
 result = EmployeeFreeTime::findEmployeeFreeTime(input);
 cout << "Free intervals: ";
 for (auto interval : result) {
  cout << "[" << interval.start << ", " << interval.end << "] ";
 }
}
```

Time complexity

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.

Space complexity

The space complexity of the above algorithm will be $O(K)$ as at any time, the heap will not have more than 'K' elements.

1. Introduction

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.

Let's jump on to our first problem to understand the **Cyclic Sort** pattern in detail.

Problem Statement

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.

Example 1:

Input: [3, 1, 5, 4, 2]
Output: [1, 2, 3, 4, 5]

Example 2:

Input: [2, 6, 4, 3, 1, 5]
Output: [1, 2, 3, 4, 5, 6]

Example 3:

Input: [1, 5, 6, 4, 3, 2]
Output: [1, 2, 3, 4, 5, 6]

Solution

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.

Let’s see this visually with the above-mentioned Example-2:

After the swap, number '2' is placed at its correct index. Number '2' is not at its correct place, let's swap it with the correct index. 2 6 4 3 1 5 2 6 4 3 1 5 6 2 4 3 1 5 1 2 3 4 5 6 Whole array is sorted.

```
using namespace std;


#include <iostream>

#include <vector>


class CyclicSort {

 public:

  static void sort(vector<int> &nums) {

    int i = 0;
```

```
    while (i < nums.size()) {
      int j = nums[i] - 1;
      if (nums[i] != nums[j]) {
        swap(nums, i, j);
      } else {
        i++;
      }
    }
  }

 private:
  static void swap(vector<int> &arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> arr = {3, 1, 5, 4, 2};
  CyclicSort::sort(arr);
  for (auto num : arr) {
    cout << num << " ";
  }
  cout << endl;

  arr = vector<int>{2, 6, 4, 3, 1, 5};
  CyclicSort::sort(arr);
  for (auto num : arr) {
```

```
  cout << num << " ";
 }
 cout << endl;


 arr = vector<int>{1, 5, 6, 4, 3, 2};
 CyclicSort::sort(arr);
 for (auto num : arr) {
  cout << num << " ";
 }
 cout << endl;
}
```

Time complexity

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 $O(n)$.

Space complexity

The algorithm runs in constant space $O(1)$.

Problem Statement

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.

Example 1:

Input: [4, 0, 3, 1]
Output: 2

Example 2:

Input: [8, 3, 5, 2, 4, 6, 0, 1]
Output: 7

Solution

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`.

```
using namespace std;


#include <iostream>

#include <vector>
```

```
class MissingNumber {
 public:
  static int findMissingNumber(vector<int> &nums) {
    int i = 0;
    while (i < nums.size()) {
      if (nums[i] < nums.size() && nums[i] != nums[nums[i]]) {
        swap(nums, i, nums[i]);
      } else {
        i++;
      }
    }

    // find the first number missing from its index, that will be our required number
    for (i = 0; i < nums.size(); i++) {
      if (nums[i] != i) {
        return i;
      }
    }

    return nums.size();
  }

 private:
  static void swap(vector<int> &arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
};
```

```
int main(int argc, char *argv[]) {
  vector<int> v1 = {4, 0, 3, 1};
  cout << MissingNumber::findMissingNumber(v1) << endl;
  v1 = {8, 3, 5, 2, 4, 6, 0, 1};
  cout << MissingNumber::findMissingNumber(v1) << endl;
}
```

Time complexity

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 $O(n)$.

Space complexity

The algorithm runs in constant space $O(1)$.

Problem Statement

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.

Example 1:

Input: [2, 3, 1, 8, 2, 3, 5, 1]
Output: 4, 6, 7
Explanation: The array should have all numbers from 1 to 8, due to duplicates 4, 6, and 7 are missing.

Example 2:

Input: [2, 4, 1, 2]
Output: 3

Example 3:

Input: [2, 3, 2, 1]
Output: 4

Solution

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.

```
using namespace std;


#include <iostream>

#include <vector>


class AllMissingNumbers {

 public:

  static vector<int> findNumbers(vector<int> &nums) {

    int i = 0;

    while (i < nums.size()) {

      if (nums[i] != nums[nums[i] - 1]) {

        swap(nums, i, nums[i] - 1);

      } else {
```

```
      i++;
    }
   }

   vector<int> missingNumbers;
   for (i = 0; i < nums.size(); i++) {
    if (nums[i] != i + 1) {
     missingNumbers.push_back(i + 1);
    }
   }

   return missingNumbers;
  }

 private:
  static void swap(vector<int> &arr, int i, int j) {
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> v1 = {2, 3, 1, 8, 2, 3, 5, 1};
  vector<int> missing = AllMissingNumbers::findNumbers(v1);
  cout << "Missing numbers: ";
  for (auto num : missing) {
   cout << num << " ";
  }
```

```
  cout << endl;

  v1 = {2, 4, 1, 2};
  missing = AllMissingNumbers::findNumbers(v1);
  cout << "Missing numbers: ";
  for (auto num : missing) {
    cout << num << " ";
  }
  cout << endl;

  v1 = {2, 3, 2, 1};
  missing = AllMissingNumbers::findNumbers(v1);
  cout << "Missing numbers: ";
  for (auto num : missing) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(n)$.

Space complexity

Ignoring the space required for the output array, the algorithm runs in constant space $O(1)$.

Problem Statement

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.

Example 1:

Input: [1, 4, 4, 3, 2]
Output: 4

Example 2:

Input: [2, 1, 3, 3, 5, 4]
Output: 3

Example 3:

Input: [2, 4, 1, 4, 4]
Output: 4

Solution

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!

```
using namespace std;


#include <iostream>

#include <vector>


class FindDuplicate {

 public:

  static int findNumber(vector<int> &nums) {
```

```
    int i = 0;
    while (i < nums.size()) {
      if (nums[i] != i + 1) {
        if (nums[i] != nums[nums[i] - 1]) {
          swap(nums, i, nums[i] - 1);
        } else  // we have found the duplicate
        {
          return nums[i];
        }
      } else {
        i++;
      }
    }

    return -1;
  }

 private:
  static void swap(vector<int> &arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> v1 = {1, 4, 4, 3, 2};
  cout << FindDuplicate::findNumber(v1) << endl;
```

```
  v1 = {2, 1, 3, 3, 5, 4};
  cout << FindDuplicate::findNumber(v1) << endl;

  v1 = {2, 4, 1, 4, 4};
  cout << FindDuplicate::findNumber(v1) << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(n)$.

Space complexity

The algorithm runs in constant space $O(1)$ but modifies the input array.

Similar Problems

Problem 1: Can we solve the above problem in $O(1)$ space and without modifying the input array?

Solution: 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.

```
using namespace std;

#include <iostream>
#include <vector>

class DuplicateNumber {
```

```
public:
 static int findDuplicate(const vector<int> &arr) {
  int slow = 0, fast = 0;
  do {
   slow = arr[slow];
   fast = arr[arr[fast]];
  } while (slow != fast);

  // find cycle length
  int current = arr[slow];
  int cycleLength = 0;
  do {
   current = arr[current];
   cycleLength++;
  } while (current != arr[slow]);

  return findStart(arr, cycleLength);
 }

private:
 static int findStart(const vector<int> &arr, int cycleLength) {
  int pointer1 = arr[0], pointer2 = arr[0];
  // move pointer2 ahead 'cycleLength' steps
  while (cycleLength > 0) {
   pointer2 = arr[pointer2];
   cycleLength--;
  }

  // increment both pointers until they meet at the start of the cycle
```

```
    while (pointer1 != pointer2) {
      pointer1 = arr[pointer1];
      pointer2 = arr[pointer2];
    }

    return pointer1;
  }
};

int main(int argc, char *argv[]) {
  cout << DuplicateNumber::findDuplicate(vector<int>{1, 4, 4, 3, 2}) << endl;
  cout << DuplicateNumber::findDuplicate(vector<int>{2, 1, 3, 3, 5, 4}) << endl;
  cout << DuplicateNumber::findDuplicate(vector<int>{2, 4, 1, 4, 4}) << endl;
}
```

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

Problem Statement

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**.

Example 1:

Input: [3, 4, 4, 5, 5]
Output: [4, 5]

Example 2:

Input: [5, 4, 7, 2, 3, 5, 3]
Output: [3, 5]

Solution

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.

```
using namespace std;


#include <iostream>

#include <vector>


class FindAllDuplicate {

 public:

  static vector<int> findNumbers(vector<int> &nums) {

    int i = 0;

    while (i < nums.size()) {

      if (nums[i] != nums[nums[i] - 1]) {

        swap(nums, i, nums[i] - 1);

      } else {

        i++;

      }

    }


    vector<int> duplicateNumbers;

    for (i = 0; i < nums.size(); i++) {

      if (nums[i] != i + 1) {

        duplicateNumbers.push_back(nums[i]);

      }

    }
```

```
    return duplicateNumbers;
  }

 private:
  static void swap(vector<int> &arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> v1 = {3, 4, 4, 5, 5};
  vector<int> duplicates = FindAllDuplicate::findNumbers(v1);
  cout << "Duplicates are: ";
  for (auto num : duplicates) {
    cout << num << " ";
  }
  cout << endl;

  v1 = {5, 4, 7, 2, 3, 5, 3};
  duplicates = FindAllDuplicate::findNumbers(v1);
  cout << "Duplicates are: ";
  for (auto num : duplicates) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(n)$.

Space complexity

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.

Example 1:

Input: [3, 1, 2, 5, 2]
Output: [2, 4]
Explanation: '2' is duplicated and '4' is missing.

Example 2:

Input: [3, 1, 2, 3, 6, 4]
Output: [3, 5]
Explanation: '3' is duplicated and '5' is missing.

Solution

This problem follows the **Cyclic Sort** pattern and shares similarities with Find all Duplicate Numbers. Following a similar approach, we will place each number at its correct index. Once we are done with the cyclic sort, we will iterate through the array to find the number that is not at the correct index. Since only one number got corrupted, the number at the wrong index is the duplicated number and the index itself represents the missing number.

```
using namespace std;
```

```
#include <iostream>
#include <string>
#include <vector>

class FindCorruptNums {
 public:
  static vector<int> findNumbers(vector<int> &nums) {
    int i = 0;
    while (i < nums.size()) {
      if (nums[i] != nums[nums[i] - 1]) {
        swap(nums, i, nums[i] - 1);
      } else {
        i++;
      }
    }

    for (i = 0; i < nums.size(); i++) {
      if (nums[i] != i + 1) {
        return vector<int>{nums[i], i + 1};
      }
    }

    return vector<int>{-1, -1};
  }

 private:
  static void swap(vector<int> &arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
```

```
    arr[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> v1 = {3, 1, 2, 5, 2};
  vector<int> nums = FindCorruptNums::findNumbers(v1);
  cout << nums[0] << ", " << nums[1] << endl;

  v1 = {3, 1, 2, 3, 6, 4};
  nums = FindCorruptNums::findNumbers(v1);
  cout << nums[0] << ", " << nums[1] << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(n)$.

Space complexity

The algorithm runs in constant space $O(1)$.

Find the Smallest Missing Positive Number (medium)

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

Example 1:

Input: [-3, 1, 5, 4, 2]
Output: 3
Explanation: The smallest missing positive number is '3'

Example 2:

Input: [3, -2, 0, 1, 2]
Output: 4

Example 3:

Input: [3, 2, 5, 1]
Output: 4

Solution

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!

```
using namespace std;


#include <iostream>

#include <vector>


class FirstSmallestMissingPositive {

 public:

  static int findNumber(vector<int> &nums) {

    int i = 0;

    while (i < nums.size()) {

      if (nums[i] > 0 && nums[i] <= nums.size() && nums[i] != nums[nums[i] - 1]) {

        swap(nums, i, nums[i] - 1);

      } else {
```

```
      i++;
    }
  }

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

  return nums.size() + 1;
 }

private:
 static void swap(vector<int> &arr, int i, int j) {
  int temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
 }
};

int main(int argc, char *argv[]) {
 vector<int> v1 = {-3, 1, 5, 4, 2};
 cout << FirstSmallestMissingPositive::findNumber(v1) << endl;

 v1 = {3, -2, 0, 1, 2};
 cout << FirstSmallestMissingPositive::findNumber(v1) << endl;

 v1 = {3, 2, 5, 1};
```

```
  cout << FirstSmallestMissingPositive::findNumber(v1) << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(n)$.

Space complexity

The algorithm runs in constant space $O(1)$.

Find the First K Missing Positive Numbers (hard)

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

Example 1:

Input: [3, -1, 4, 5, 5], k=3
Output: [1, 2, 6]
Explanation: The smallest missing positive numbers are 1, 2 and 6.

Example 2:

Input: [2, 3, 4], k=3
Output: [1, 5, 6]
Explanation: The smallest missing positive numbers are 1, 5 and 6.

Example 3:

Input: [-2, -3, 4], k=2
Output: [1, 2]
Explanation: The smallest missing positive numbers are 1 and 2.

Solution

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'.

```
using namespace std;


#include <iostream>

#include <unordered_set>

#include <vector>
```

```
class FirstKMissingPositive {
 public:
  static vector<int> findNumbers(vector<int> &nums, int k) {
    int i = 0;
    while (i < nums.size()) {
      if (nums[i] > 0 && nums[i] <= nums.size() && nums[i] != nums[nums[i] - 1]) {
        swap(nums, i, nums[i] - 1);
      } else {
        i++;
      }
    }

    vector<int> missingNumbers;
    unordered_set<int> extraNumbers;
    for (i = 0; i < nums.size() && missingNumbers.size() < k; i++) {
      if (nums[i] != i + 1) {
        missingNumbers.push_back(i + 1);
        extraNumbers.insert(nums[i]);
      }
    }

    // add the remaining missing numbers
    for (i = 1; missingNumbers.size() < k; i++) {
      int candidateNumber = i + nums.size();
      // ignore if the array contains the candidate number
      if (extraNumbers.find(candidateNumber) == extraNumbers.end()) {
        missingNumbers.push_back(candidateNumber);
      }
    }
```

```
    return missingNumbers;
  }

 private:
  static void swap(vector<int> &arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> v1 = {3, -1, 4, 5, 5};
  vector<int> missingNumbers = FirstKMissingPositive::findNumbers(v1, 3);
  cout << "Missing numbers: ";
  for (auto num : missingNumbers) {
    cout << num << " ";
  }
  cout << endl;

  v1 = {2, 3, 4};
  missingNumbers = FirstKMissingPositive::findNumbers(v1, 3);
  cout << "Missing numbers: ";
  for (auto num : missingNumbers) {
    cout << num << " ";
  }
  cout << endl;
```

```
  v1 = {-2, -3, 4};
  missingNumbers = FirstKMissingPositive::findNumbers(v1, 2);
  cout << "Missing numbers: ";
  for (auto num : missingNumbers) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

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.

Space complexity

The algorithm needs $O(k)$ space to store the `extraNumbers`.

1. Introduction

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. In the following chapters, we will solve a bunch of problems using this pattern.

Let's jump on to our first problem to understand this pattern.

Problem Statement

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

2 4 6 8 10 null Original List: Reversed List: Example: 2 4 6 8 10

Solution

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. Here is the visual representation of our algorithm:

```
using namespace std;


#include <iostream>


class ListNode {
 public:
  int value = 0;
  ListNode *next;


  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};


class ReverseLinkedList {
 public:
  static ListNode *reverse(ListNode *head) {
    ListNode *current = head;      // current node that we will be processing
    ListNode *previous = nullptr;  // previous node that we have processed
    ListNode *next = nullptr;      // will be used to temporarily store the next node
```

```
    while (current != nullptr) {
      next = current->next;      // temporarily store the next node
      current->next = previous;  // reverse the current node
      previous = current;  // before we move to the next node, point previous to the current node
      current = next;      // move on the next node
    }
    // after the loop current will be pointing to 'null' and 'previous' will be the new head
    return previous;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(2);
  head->next = new ListNode(4);
  head->next->next = new ListNode(6);
  head->next->next->next = new ListNode(8);
  head->next->next->next->next = new ListNode(10);

  ListNode *result = ReverseLinkedList::reverse(head);
  cout << "Nodes of the reversed LinkedList are: ";
  while (result != nullptr) {
    cout << result->value << " ";
    result = result->next;
  }
}
```

Time complexity

The time complexity of our algorithm will be $O(N)$ where 'N' is the total number of nodes in the LinkedList.

Space complexity

We only used constant space, therefore, the space complexity of our algorithm is $O(1)$.

Problem Statement

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

Original List: Example: head 1 2 3 4 5 null 1 4 3 2 5 null

Solution

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.

```
using namespace std;


#include <iostream>


class ListNode {

 public:

  int value = 0;
```

```
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class ReverseSubList {
 public:
  static ListNode *reverse(ListNode *head, int p, int q) {
    if (p == q) {
      return head;
    }

    // after skipping 'p-1' nodes, current will point to 'p'th node
    ListNode *current = head, *previous = nullptr;
    for (int i = 0; current != nullptr && i < p - 1; ++i) {
      previous = current;
      current = current->next;
    }

    // we are interested in three parts of the LinkedList, part before index 'p', part between 'p'
    // and 'q', and the part after index 'q'
    ListNode *lastNodeOfFirstPart = previous;  // points to the node at index 'p-1'

    // after reversing the LinkedList 'current' will become the last node of the sub-list
    ListNode *lastNodeOfSubList = current;
    ListNode *next = nullptr;  // will be used to temporarily store the next node
```

```
    // reverse nodes between 'p' and 'q'
    for (int i = 0; current != nullptr && i < q - p + 1; i++) {
      next = current->next;
      current->next = previous;
      previous = current;
      current = next;
    }

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

    // connect with the last part
    lastNodeOfSubList->next = current;

    return head;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
```

```
  ListNode *result = ReverseSubList::reverse(head, 2, 4);
  cout << "Nodes of the reversed LinkedList are: ";
  while (result != nullptr) {
    cout << result->value << " ";
    result = result->next;
  }
}
```

Time complexity

The time complexity of our algorithm will be $O(N)$ where 'N' is the total number of nodes in the LinkedList.

Space complexity

We only used constant space, therefore, the space complexity of our algorithm is $O(1)$.

Similar Questions

Problem 1: Reverse the first 'k' elements of a given LinkedList.

Solution: 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`.

Problem 2: 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.

Solution: 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.

Problem Statement

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.

Original List: Example: head head 1 2 3 4 5 6 7 8 null 3 2 1 6 5 4 8 7 null

Solution

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’.

```
using namespace std;


#include <iostream>


class ListNode {

 public:

  int value = 0;
```

```
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class ReverseEveryKElements {
 public:
  static ListNode *reverse(ListNode *head, int k) {
    if (k <= 1 || head == nullptr) {
      return head;
    }

    ListNode *current = head, *previous = nullptr;
    while (true) {
      ListNode *lastNodeOfPreviousPart = previous;
      // after reversing the LinkedList 'current' will become the last node of the sub-list
      ListNode *lastNodeOfSubList = current;
      ListNode *next = nullptr;  // will be used to temporarily store the next node
      // reverse 'k' nodes
      for (int i = 0; current != nullptr && i < k; i++) {
        next = current->next;
        current->next = previous;
        previous = current;
        current = next;
      }
```

```
    // connect with the previous part
    if (lastNodeOfPreviousPart != nullptr) {
      lastNodeOfPreviousPart->next =
          previous;  // 'previous' is now the first node of the sub-list
    } else {        // this means we are changing the first node (head) of the LinkedList
      head = previous;
    }

    // connect with the next part
    lastNodeOfSubList->next = current;

    if (current == nullptr) {  // break, if we've reached the end of the LinkedList
      break;
    }
    // prepare for the next sub-list
    previous = lastNodeOfSubList;
   }

   return head;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);
```

```
  head->next->next->next->next->next->next = new ListNode(7);
  head->next->next->next->next->next->next->next = new ListNode(8);

  ListNode *result = ReverseEveryKElements::reverse(head, 3);
  cout << "Nodes of the reversed LinkedList are: ";
  while (result != nullptr) {
    cout << result->value << " ";
    result = result->next;
  }
}
```

Time complexity

The time complexity of our algorithm will be $O(N)$ where 'N' is the total number of nodes in the LinkedList.

Space complexity

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.

Original List: Example: head head 1 2 3 4 5 6 7 8 null Reversed Sub-list: k=2 2 1 3 4 6 5 7 8 null

Solution

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.

```
using namespace std;


#include <iostream>


class ListNode {
 public:
  int value = 0;
  ListNode *next;


  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};


class ReverseAlternatingKElements {
 public:
  static ListNode *reverse(ListNode *head, int k) {
    if (k <= 1 || head == nullptr) {
      return head;
    }


    ListNode *current = head, *previous = nullptr;
    while (current != nullptr) { // break if we've reached the end of the list
      ListNode *lastNodeOfPreviousPart = previous;
      // after reversing the LinkedList 'current' will become the last node of the sub-list
      ListNode *lastNodeOfSubList = current;
```

```
  ListNode *next = nullptr;  // will be used to temporarily store the next node
  // reverse 'k' nodes
  for (int i = 0; current != nullptr && i < k; i++) {
    next = current->next;
    current->next = previous;
    previous = current;
    current = next;
  }

  // connect with the previous part
  if (lastNodeOfPreviousPart != nullptr) {
    lastNodeOfPreviousPart->next =
        previous;  // 'previous' is now the first node of the sub-list
  } else {        // this means we are changing the first node (head) of the LinkedList
    head = previous;
  }

  // connect with the next part
  lastNodeOfSubList->next = current;

  // skip 'k' nodes
  for (int i = 0; current != nullptr && i < k; ++i) {
    previous = current;
    current = current->next;
  }
 }

 return head;
}
```

```
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);
  head->next->next->next->next->next->next = new ListNode(7);
  head->next->next->next->next->next->next->next = new ListNode(8);

  ListNode *result = ReverseAlternatingKElements::reverse(head, 2);
  cout << "Nodes of the reversed LinkedList are: ";
  while (result != nullptr) {
    cout << result->value << " ";
    result = result->next;
  }
}
```

Time complexity

The time complexity of our algorithm will be $O(N)$ where 'N' is the total number of nodes in the LinkedList.

Space complexity

We only used constant space, therefore, the space complexity of our algorithm is $O(1)$.

Rotate a LinkedList (medium)

Given the head of a Singly LinkedList and a number ‘k’, rotate the LinkedList to the right by ‘k’ nodes.

head head k=3 1 2 3 4 5 6 null 4 5 6 1 2 3 null Rotated LinkedList: Example 1:

Original List: head head Rotated LinkedList: 1 2 3 4 5 null Example 2: k=8 3 4 5 1 2 null

Solution

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.

```
using namespace std;


#include <iostream>


class ListNode {
 public:
  int value = 0;
  ListNode *next;


  ListNode(int value) {
    this->value = value;
    next = nullptr;
```

```
  }
};

class RotateList {
 public:
  static ListNode *rotate(ListNode *head, int rotations) {
    if (head == nullptr || head->next == nullptr || rotations <= 0) {
      return head;
    }

    // find the length and the last node of the list
    ListNode *lastNode = head;
    int listLength = 1;
    while (lastNode->next != nullptr) {
      lastNode = lastNode->next;
      listLength++;
    }

    lastNode->next = head;    // connect the last node with the head to make it a circular list
    rotations %= listLength;  // no need to do rotations more than the length of the list
    int skipLength = listLength - rotations;
    ListNode *lastNodeOfRotatedList = head;
    for (int 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 = nullptr;
```

```
    return head;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);

  ListNode *result = RotateList::rotate(head, 3);
  cout << "Nodes of the reversed LinkedList are: ";
  while (result != nullptr) {
    cout << result->value << " ";
    result = result->next;
  }
}
```

Time complexity

The time complexity of our algorithm will be $O(N)$ where 'N' is the total number of nodes in the LinkedList.

Space complexity

We only used constant space, therefore, the space complexity of our algorithm is $O(1)$.

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.

Let's jump onto our first problem to understand this pattern.

Problem Statement

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.

Example 1:

1 2 3 4 5 6 7 Level Order Traversal: [[1],[2,3],[4,5,6,7]]

Example 2:

12 7 1 9 10 5

Solution

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.

Let's take the example-2 mentioned above to visually represent our algorithm:

```
using namespace std;


#include <iostream>
#include <queue>


class TreeNode {
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;


  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};


class LevelOrderTraversal {
 public:
  static vector<vector<int>> traverse(TreeNode *root) {
    vector<vector<int>> result;
    if (root == nullptr) {
      return result;
    }
```

```
    queue<TreeNode *> queue;
    queue.push(root);
    while (!queue.empty()) {
      int levelSize = queue.size();
      vector<int> currentLevel;
      for (int i = 0; i < levelSize; i++) {
        TreeNode *currentNode = queue.front();
        queue.pop();
        // add the node to the current level
        currentLevel.push_back(currentNode->val);
        // insert the children of current node in the queue
        if (currentNode->left != nullptr) {
          queue.push(currentNode->left);
        }
        if (currentNode->right != nullptr) {
          queue.push(currentNode->right);
        }
      }
      result.push_back(currentLevel);
    }

    return result;
  }
};

int main(int argc, char *argv[]) {
  TreeNode *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);
  vector<vector<int>> result = LevelOrderTraversal::traverse(root);
  cout << "Level order traversal: ";
  for (auto vec : result) {
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << endl;
  }
}
```

Time complexity

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.

Space complexity

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.

Problem Statement

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.

Solution

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.

Code

Here is what our algorithm will look like; only the highlighted lines have changed. Please note that, for **Java**, we will use a `LinkedList` instead of an `ArrayList` for our result list. As in the case of `ArrayList`, appending an element at the beginning means shifting all the existing elements. Since we need to append the level array at the beginning of the result list, a `LinkedList` will be better, as this shifting of elements is not required in a `LinkedList`. Similarly, we will use a double-ended queue (deque) for **Python**, **C++**, and **JavaScript**.

```
using namespace std;


#include <deque>

#include <iostream>

#include <queue>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;


  TreeNode(int x) {

    val = x;

    left = right = nullptr;

  }

};
```

```
class ReverseLevelOrderTraversal {
 public:
  static deque<vector<int>> traverse(TreeNode *root) {
    deque<vector<int>> result = deque<vector<int>>();
    if (root == nullptr) {
      return result;
    }

    queue<TreeNode *> queue;
    queue.push(root);
    while (!queue.empty()) {
      int levelSize = queue.size();
      vector<int> currentLevel;
      for (int i = 0; i < levelSize; i++) {
        TreeNode *currentNode = queue.front();
        queue.pop();
        // add the node to the current level
        currentLevel.push_back(currentNode->val);
        // insert the children of current node to the queue
        if (currentNode->left != nullptr) {
          queue.push(currentNode->left);
        }
        if (currentNode->right != nullptr) {
          queue.push(currentNode->right);
        }
      }
      // append the current level at the beginning
      result.push_front(currentLevel);
```

```
    }


    return result;
  }
};
int main(int argc, char *argv[]) {
  TreeNode *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);
  auto result = ReverseLevelOrderTraversal::traverse(root);

  cout << "Reverse level order traversal: ";
  for (auto que : result) {
    for (auto num : que) {
      cout << num << " ";
    }
    cout << endl;
  }
}
```

Time complexity

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.

Space complexity

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.

Problem Statement

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.

Solution

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.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;


  TreeNode(int x) {

    val = x;
```

```
    left = right = nullptr;
  }
};

class ZigzagTraversal {
 public:
  static vector<vector<int>> traverse(TreeNode *root) {
    vector<vector<int>> result;
    if (root == nullptr) {
      return result;
    }

    queue<TreeNode *> queue;
    queue.push(root);
    bool leftToRight = true;
    while (!queue.empty()) {
      int levelSize = queue.size();
      vector<int> currentLevel(levelSize);
      for (int i = 0; i < levelSize; i++) {
        TreeNode *currentNode = queue.front();
        queue.pop();

        // add the node to the current level based on the traverse direction
        if (leftToRight) {
          currentLevel[i] = currentNode->val;
        } else {
          currentLevel[levelSize - 1 - i] = currentNode->val;
        }
```

```
      // insert the children of current node in the queue
      if (currentNode->left != nullptr) {
        queue.push(currentNode->left);
      }
      if (currentNode->right != nullptr) {
        queue.push(currentNode->right);
      }
    }
    result.push_back(currentLevel);
    // reverse the traversal direction
    leftToRight = !leftToRight;
  }

  return result;
 }
};

int main(int argc, char *argv[]) {
 TreeNode *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->right->left->left = new TreeNode(20);
 root->right->left->right = new TreeNode(17);
 vector<vector<int>> result = ZigzagTraversal::traverse(root);
 cout << "Zigzag traversal: ";
 for (auto vec : result) {
```

```
    for (auto num : vec) {
      cout << num << " ";
    }
  }
}
```

Time complexity

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.

Space complexity

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.

Problem Statement

Given a binary tree, populate an array to represent the **averages of all of its levels**.

Solution

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.

```
using namespace std;


#include <iostream>
```

```
#include <queue>

class TreeNode {
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;

  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};

class LevelAverage {
 public:
  static vector<double> findLevelAverages(TreeNode *root) {
    vector<double> result;
    if (root == nullptr) {
      return result;
    }

    queue<TreeNode *> queue;
    queue.push(root);
    while (!queue.empty()) {
      int levelSize = queue.size();
      double levelSum = 0;
      for (int i = 0; i < levelSize; i++) {
        TreeNode *currentNode = queue.front();
```

```
      queue.pop();
      // add the node's value to the running sum
      levelSum += currentNode->val;
      // insert the children of current node to the queue
      if (currentNode->left != nullptr) {
        queue.push(currentNode->left);
      }
      if (currentNode->right != nullptr) {
        queue.push(currentNode->right);
      }
    }
    // append the current level's average to the result array
    result.push_back(levelSum / levelSize);
  }

  return result;
 }
};

int main(int argc, char *argv[]) {
 TreeNode *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);
 vector<double> result = LevelAverage::findLevelAverages(root);
 cout << "Level averages are: ";
```

```
  for (auto num : result) {
    cout << num << " ";
  }
}
```

Time complexity

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.

Space complexity

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.

Similar Problems

Problem 1: Find the largest value on each level of a binary tree.

Solution: We will follow a similar approach, but instead of having a running sum we will track the maximum value of each level.

```
maxValue = max(maxValue, currentNode.val)
```

Problem Statement

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**.

Solution

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.

```
using namespace std;


#include <iostream>

#include <queue>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;


  TreeNode(int x) {

    val = x;

    left = right = nullptr;

  }

};


class MinimumBinaryTreeDepth {

 public:

  static int findDepth(TreeNode *root) {

    if (root == nullptr) {

      return 0;

    }


    queue<TreeNode *> queue;
```

```
    queue.push(root);
    int minimumTreeDepth = 0;
    while (!queue.empty()) {
      minimumTreeDepth++;
      int levelSize = queue.size();
      for (int i = 0; i < levelSize; i++) {
        TreeNode *currentNode = queue.front();
        queue.pop();

        // check if this is a leaf node
        if (currentNode->left == nullptr && currentNode->right == nullptr) {
          return minimumTreeDepth;
        }

        // insert the children of current node in the queue
        if (currentNode->left != nullptr) {
          queue.push(currentNode->left);
        }
        if (currentNode->right != nullptr) {
          queue.push(currentNode->right);
        }
      }
    }
    return minimumTreeDepth;
  }
};

int main(int argc, char *argv[]) {
  TreeNode *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);
  cout << "Tree Minimum Depth: " << MinimumBinaryTreeDepth::findDepth(root) << endl;
  root->left->left = new TreeNode(9);
  root->right->left->left = new TreeNode(11);
  cout << "Tree Minimum Depth: " << MinimumBinaryTreeDepth::findDepth(root) << endl;
}
```

Time complexity

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.

Space complexity

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.

Similar Problems

Problem 1: Given a binary tree, find its maximum depth (or height).

Solution: 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. Here is what the code will look like:

```
using namespace std;


#include <iostream>
```

```
#include <queue>

class TreeNode {
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;

  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};

class MaximumBinaryTreeDepth {
 public:
  static int findDepth(TreeNode *root) {
    if (root == nullptr) {
      return 0;
    }

    queue<TreeNode *> queue;
    queue.push(root);
    int maximumTreeDepth = 0;
    while (!queue.empty()) {
      maximumTreeDepth++;
      int levelSize = queue.size();
      for (int i = 0; i < levelSize; i++) {
        TreeNode *currentNode = queue.front();
```

```
      queue.pop();
      // insert the children of current node in the queue
      if (currentNode->left != nullptr) {
        queue.push(currentNode->left);
      }
      if (currentNode->right != nullptr) {
        queue.push(currentNode->right);
      }
    }
  }

  return maximumTreeDepth;
 }
};

int main(int argc, char *argv[]) {
 TreeNode *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);
 cout << "Tree Maximum Depth: " << MaximumBinaryTreeDepth::findDepth(root) << endl;
 root->left->left = new TreeNode(9);
 root->right->left->left = new TreeNode(11);
 cout << "Tree Maximum Depth: " << MaximumBinaryTreeDepth::findDepth(root) << endl;
}
```

Problem Statement

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.

Solution

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.

```
using namespace std;


#include <iostream>

#include <queue>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;


  TreeNode(int x) {

    val = x;

    left = right = nullptr;

  }

};


class LevelOrderSuccessor {

 public:
```

```
  static TreeNode *findSuccessor(TreeNode *root, int key) {
    if (root == nullptr) {
      return nullptr;
    }

    queue<TreeNode *> queue;
    queue.push(root);
    while (!queue.empty()) {
      TreeNode *currentNode = queue.front();
      queue.pop();
      // insert the children of current node in the queue
      if (currentNode->left != nullptr) {
        queue.push(currentNode->left);
      }
      if (currentNode->right != nullptr) {
        queue.push(currentNode->right);
      }

      // break if we have found the key
      if (currentNode->val == key) {
        break;
      }
    }

    return queue.front();
  }
};

int main(int argc, char *argv[]) {
```

```
  TreeNode *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);
  TreeNode *result = LevelOrderSuccessor::findSuccessor(root, 12);
  if (result != nullptr) {
    cout << result->val << " " << endl;
  }
  result = LevelOrderSuccessor::findSuccessor(root, 9);
  if (result != nullptr) {
    cout << result->val << " " << endl;
  }
}
```

Time complexity

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.

Space complexity

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.

Problem Statement

Given a binary tree, connect each node with its level order successor. The last node of each level should point to a `null` node.

Solution

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.

```
using namespace std;


#include <iostream>

#include <queue>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;

  TreeNode *next;


  TreeNode(int x) {

    val = x;

    left = right = next = nullptr;

  }


  // level order traversal using 'next' pointer

  virtual void printLevelOrder() {

    TreeNode *nextLevelRoot = this;

    while (nextLevelRoot != nullptr) {
```

```
      TreeNode *current = nextLevelRoot;
      nextLevelRoot = nullptr;
      while (current != nullptr) {
        cout << current->val << " ";
        if (nextLevelRoot == nullptr) {
          if (current->left != nullptr) {
            nextLevelRoot = current->left;
          } else if (current->right != nullptr) {
            nextLevelRoot = current->right;
          }
        }
        current = current->next;
      }
      cout << endl;
    }
  }
};

class ConnectLevelOrderSiblings {
 public:
  static void connect(TreeNode *root) {
    if (root == nullptr) {
      return;
    }

    queue<TreeNode *> queue;
    queue.push(root);
    while (!queue.empty()) {
      TreeNode *previousNode = nullptr;
```

```
    int levelSize = queue.size();
    // connect all nodes of this level
    for (int i = 0; i < levelSize; i++) {
      TreeNode *currentNode = queue.front();
      queue.pop();
      if (previousNode != nullptr) {
        previousNode->next = currentNode;
      }
      previousNode = currentNode;

      // insert the children of current node in the queue
      if (currentNode->left != nullptr) {
        queue.push(currentNode->left);
      }
      if (currentNode->right != nullptr) {
        queue.push(currentNode->right);
      }
    }
   }
  }
};

int main(int argc, char *argv[]) {
  TreeNode *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::connect(root);
  cout << "Level order traversal using 'next' pointer: " << endl;
  root->printLevelOrder();
}
```

Time complexity

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.

Space complexity

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.

Solution

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.

```
using namespace std;


#include <iostream>
#include <queue>


class TreeNode {
```

```
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;
  TreeNode *next;

  TreeNode(int x) {
    val = x;
    left = right = next = nullptr;
  }

  // tree traversal using 'next' pointer
  virtual void printTree() {
    TreeNode *current = this;
    cout << "Traversal using 'next' pointer: ";
    while (current != nullptr) {
      cout << current->val << " ";
      current = current->next;
    }
  }
};

class ConnectAllSiblings {
 public:
  static void connect(TreeNode *root) {
    if (root == nullptr) {
      return;
    }
```

```
    queue<TreeNode *> queue;
    queue.push(root);
    TreeNode *currentNode = nullptr, *previousNode = nullptr;
    while (!queue.empty()) {
      currentNode = queue.front();
      queue.pop();
      if (previousNode != nullptr) {
        previousNode->next = currentNode;
      }
      previousNode = currentNode;

      // insert the children of current node in the queue
      if (currentNode->left != nullptr) {
        queue.push(currentNode->left);
      }
      if (currentNode->right != nullptr) {
        queue.push(currentNode->right);
      }
    }
  }
};

int main(int argc, char *argv[]) {
  TreeNode *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::connect(root);
  root->printTree();
}
```

Time complexity

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.

Space complexity

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)

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**.

Solution

This problem follows the Binary Tree Level Order Traversal pattern. We can follow the same **BFS** approach. The only additional thing we will be doing is to append the last node of each level to the result array.

```
using namespace std;


#include <iostream>
#include <queue>
#include <vector>
```

```
class TreeNode {
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;

  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};

class RightViewTree {
 public:
  static vector<TreeNode *> traverse(TreeNode *root) {
    vector<TreeNode *> result;
    if (root == nullptr) {
      return result;
    }

    queue<TreeNode *> queue;
    queue.push(root);
    while (!queue.empty()) {
      int levelSize = queue.size();
      for (int i = 0; i < levelSize; i++) {
        TreeNode *currentNode = queue.front();
        queue.pop();
        // if it is the last node of this level, add it to the result
```

```
      if (i == levelSize - 1) {
        result.push_back(currentNode);
      }
      // insert the children of current node in the queue
      if (currentNode->left != nullptr) {
        queue.push(currentNode->left);
      }
      if (currentNode->right != nullptr) {
        queue.push(currentNode->right);
      }
    }
  }

  return result;
 }
};

int main(int argc, char *argv[]) {
 TreeNode *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);
 vector<TreeNode *> result = RightViewTree::traverse(root);
 for (auto node : result) {
  cout << node->val << " ";
 }
```

}

Time complexity

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.

Space complexity

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

Problem 1: 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.

Solution: 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.

1. Introduction

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.

Let's jump onto our first problem to understand this pattern.

Problem Statement

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’.

Solution

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.

Let’s take the example-2 mentioned above to visually see our algorithm:

```
using namespace std;


#include <iostream>
```

```
class TreeNode {
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;

  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};

class TreePathSum {
 public:
  static bool hasPath(TreeNode *root, int sum) {
    if (root == nullptr) {
      return false;
    }

    // if the current node is a leaf and its value is equal to the sum, we've found a path
    if (root->val == sum && root->left == nullptr && root->right == nullptr) {
      return true;
    }

    // recursively call to traverse the left and right sub-tree
    // return true if any of the two recursive call return true
    return hasPath(root->left, sum - root->val) || hasPath(root->right, sum - root->val);
  }
};
```

```
int main(int argc, char *argv[]) {
  TreeNode *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);
  cout << "Tree has path: " << TreePathSum::hasPath(root, 23) << endl;
  cout << "Tree has path: " << TreePathSum::hasPath(root, 16) << endl;
}
```

Time complexity

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.

Space complexity

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).

Problem Statement

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'.

Solution

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.

```
using namespace std;


#include <iostream>
#include <vector>


class TreeNode {
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;


  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};


class FindAllTreePaths {
 public:
  static vector<vector<int>> findPaths(TreeNode *root, int sum) {
    vector<vector<int>> allPaths;
    vector<int> currentPath;
    findPathsRecursive(root, sum, currentPath, allPaths);
    return allPaths;
  }
```

```
 private:
  static void findPathsRecursive(TreeNode *currentNode, int sum, vector<int> &currentPath,
                                 vector<vector<int>> &allPaths) {
    if (currentNode == nullptr) {
      return;
    }

    // add the current node to the path
    currentPath.push_back(currentNode->val);

    // if the current node is a leaf and its value is equal to sum, save the current path
    if (currentNode->val == sum && currentNode->left == nullptr && currentNode->right == nullptr) {
      allPaths.push_back(vector<int>(currentPath));
    } else {
      // traverse the left sub-tree
      findPathsRecursive(currentNode->left, sum - currentNode->val, currentPath, allPaths);
      // traverse the right sub-tree
      findPathsRecursive(currentNode->right, sum - currentNode->val, currentPath, allPaths);
    }

    // remove the current node from the path to backtrack,
    // we need to remove the current node while we are going up the recursive call stack.
    currentPath.pop_back();
  }
};

int main(int argc, char *argv[]) {
  TreeNode *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);
  int sum = 23;
  vector<vector<int>> result = FindAllTreePaths::findPaths(root, sum);
  cout << "Tree paths with sum " << sum << ": " << endl;
  for (auto vec : result) {
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << endl;
  }
}
```

Time complexity

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.

Space complexity

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).

How can we estimate the space used for the `allPaths` array? Take the example of the following balanced tree:

Here we have seven nodes (i.e., `N = 7`). Since, for binary trees, there exists only one path to reach any leaf node, we can easily say that total root-to-leaf paths in a binary tree can't be more than $(N+1)/2$ leaves in a binary tree, therefore the maximum number of elements in `allPaths` will be $O((N+1)/2) = O(N)$. Now, each of these paths can have many nodes in them. For a balanced binary tree (like above), each leaf node will be at maximum depth. As we know that the depth (or height) of a balanced binary tree is $O(logN)$ we can say that, at the most, each path can have $logN$ nodes in it. This means that the total size of the allPaths list will be $O(N*logN)$. If the tree is not balanced, we will still have the same worst-case space complexity.

From the above discussion, we can conclude that our algorithm's overall space complexity is $O(N*logN)$.

Also, from the above discussion, since for each leaf node, in the worst case, we have to copy $log(N)$ nodes to store its path; therefore, the time complexity of our algorithm will also be $O(N*logN)$.

Similar Problems

Problem 1: Given a binary tree, return all root-to-leaf paths.

Solution: We can follow a similar approach. We just need to remove the "check for the path sum."

Problem 2: Given a binary tree, find the root-to-leaf path with the maximum sum.

Solution: 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.

Problem Statement

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.

Solution

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.

```
using namespace std;


#include <iostream>


class TreeNode {
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;


  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};


class SumOfPathNumbers {
```

```
 public:
  static int findSumOfPathNumbers(TreeNode *root) {
    return findRootToLeafPathNumbers(root, 0);
  }

 private:
  static int findRootToLeafPathNumbers(TreeNode *currentNode, int pathSum) {
    if (currentNode == nullptr) {
      return 0;
    }

    // calculate the path number of the current node
    pathSum = 10 * pathSum + currentNode->val;

    // if the current node is a leaf, return the current path sum.
    if (currentNode->left == nullptr && currentNode->right == nullptr) {
      return pathSum;
    }
    // traverse the left and the right sub-tree
    return findRootToLeafPathNumbers(currentNode->left, pathSum) +
           findRootToLeafPathNumbers(currentNode->right, pathSum);
  }
};

int main(int argc, char *argv[]) {
  TreeNode *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);
  cout << "Total Sum of Path Numbers: " << SumOfPathNumbers::findSumOfPathNumbers(root) << endl;
}
```

Time complexity

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.

Space complexity

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).

Problem Statement

Given a binary tree and a number sequence, find if the sequence is present as a root-to-leaf path in the given tree.

Solution

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.

```
#include <iostream>
#include <vector>

using namespace std;
```

```
class TreeNode {
 public:
  int val;
  TreeNode *left;
  TreeNode *right;

  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};

class PathWithGivenSequence {
 public:
  static bool findPath(TreeNode *root, const vector<int> &sequence) {
    if (root == nullptr) {
      return sequence.empty();
    }

    return findPathRecursive(root, sequence, 0);
  }

 private:
  static bool findPathRecursive(TreeNode *currentNode, const vector<int> &sequence,
                                int sequenceIndex) {
    if (currentNode == nullptr) {
      return false;
    }
```

```
    if (sequenceIndex >= sequence.size() || currentNode->val != sequence[sequenceIndex]) {
      return false;
    }

    // if the current node is a leaf, add it is the end of the sequence, we have found a path!
    if (currentNode->left == nullptr && currentNode->right == nullptr &&
        sequenceIndex == sequence.size() - 1) {
      return true;
    }

    // recursively call to traverse the left and right sub-tree
    // return true if any of the two recursive call return true
    return findPathRecursive(currentNode->left, sequence, sequenceIndex + 1) ||
           findPathRecursive(currentNode->right, sequence, sequenceIndex + 1);
  }
};

int main(int argc, char *argv[]) {
  TreeNode *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);

  cout << "Tree has path sequence: " << PathWithGivenSequence::findPath(root, vector<int>{1, 0, 7})
       << endl;
  cout << "Tree has path sequence: " << PathWithGivenSequence::findPath(root, vector<int>{1, 1, 6})
       << endl;
```

}

Time complexity

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.

Space complexity

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).

Problem Statement

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).

Solution

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.

```
using namespace std;


#include <iostream>
#include <vector>


class TreeNode {
 public:
  int val = 0;
  TreeNode *left;
  TreeNode *right;


  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};


class CountAllPathSum {
 public:
  static int countPaths(TreeNode *root, int S) {
    vector<int> currentPath;
    return countPathsRecursive(root, S, currentPath);
  }


 private:
```

```
static int countPathsRecursive(TreeNode *currentNode, int S, vector<int> &currentPath) {
  if (currentNode == nullptr) {
    return 0;
  }

  // add the current node to the path
  currentPath.push_back(currentNode->val);
  int pathCount = 0, pathSum = 0;
  // find the sums of all sub-paths in the current path list
  for (vector<int>::reverse_iterator itr = currentPath.rbegin(); itr != currentPath.rend();
       ++itr) {
    pathSum += *itr;
    // if the sum of any sub-path is equal to '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 up the recursive call stack.
  currentPath.pop_back();

  return pathCount;
}
```

```
};


int main(int argc, char *argv[]) {
  TreeNode *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);
  cout << "Tree has path: " << CountAllPathSum::countPaths(root, 11) << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(N^2)$ 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 $O(NlogN)$.

Space complexity

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)

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.

Solution

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 current node.
3. The tree diameter at the current node 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.

```
using namespace std;


#include <iostream>

#include <vector>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;
```

```
  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};

class TreeDiameter {
 public:
  static int findDiameter(TreeNode *root) {
    int treeDiameter = 0;
    calculateHeight(root, treeDiameter);
    return treeDiameter;
  }

 private:
  static int calculateHeight(TreeNode *currentNode, int &treeDiameter) {
    if (currentNode == nullptr) {
      return 0;
    }

    int leftTreeHeight = calculateHeight(currentNode->left, treeDiameter);
    int rightTreeHeight = calculateHeight(currentNode->right, treeDiameter);

    // if the current node doesn't have a left or right subtree, 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 current node will be equal to the height of left subtree +
      // the height of right sub-trees + '1' for the current node
      int diameter = leftTreeHeight + rightTreeHeight + 1;
```

```
    // update the global tree diameter
    treeDiameter = max(treeDiameter, diameter);
   }

   // height of the current node will be equal to the maximum of the heights of
   // left or right subtrees plus '1' for the current node
   return max(leftTreeHeight, rightTreeHeight) + 1;
  }
};

int main(int argc, char *argv[]) {
  TreeNode *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);
  cout << "Tree Diameter: " << TreeDiameter::findDiameter(root) << endl;
  root->left->left = nullptr;
  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);
  cout << "Tree Diameter: " << TreeDiameter::findDiameter(root) << endl;
}
```

Time complexity

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.

Space complexity

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)

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.

Solution

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.

```
#include <iostream>

#include <limits>


using namespace std;


class TreeNode {

 public:
```

```
  int val;
  TreeNode *left;
  TreeNode *right;

  TreeNode(int x) {
    val = x;
    left = right = nullptr;
  }
};

class MaximumPathSum {
 public:
  static int findMaximumPathSum(TreeNode *root) {
    int globalMaximumSum = numeric_limits<int>::min();
    findMaximumPathSumRecursive(root, globalMaximumSum);
    return globalMaximumSum;
  }

 private:
  static int findMaximumPathSumRecursive(TreeNode *currentNode, int &globalMaximumSum) {
    if (currentNode == nullptr) {
      return 0;
    }

    int maxPathSumFromLeft = findMaximumPathSumRecursive(currentNode-
>left, globalMaximumSum);
    int maxPathSumFromRight = findMaximumPathSumRecursive(currentNode-
>right, globalMaximumSum);
```

```
    // ignore paths with negative sums, since we need to find the maximum sum we should
    // ignore any path which has an overall negative sum.
    maxPathSumFromLeft = max(maxPathSumFromLeft, 0);
    maxPathSumFromRight = max(maxPathSumFromRight, 0);

    // maximum path sum at the current node will be equal to the sum from the left subtree +
    // the sum from right subtree + val of current node
    int localMaximumSum = maxPathSumFromLeft + maxPathSumFromRight + currentNode->val;

    // update the global maximum sum
    globalMaximumSum = max(globalMaximumSum, localMaximumSum);

    // maximum sum of any path from the current node will be equal to the maximum of
    // the sums from left or right subtrees plus the value of the current node
    return max(maxPathSumFromLeft, maxPathSumFromRight) + currentNode->val;
  }
};

int main(int argc, char *argv[]) {
  TreeNode *root = new TreeNode(1);
  root->left = new TreeNode(2);
  root->right = new TreeNode(3);
  cout << "Maximum Path Sum: " << MaximumPathSum::findMaximumPathSum(root) << endl;

  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);
  cout << "Maximum Path Sum: " << MaximumPathSum::findMaximumPathSum(root) << endl;

  root = new TreeNode(-1);
  root->left = new TreeNode(-3);
  cout << "Maximum Path Sum: " << MaximumPathSum::findMaximumPathSum(root) << endl;
}
```

Time complexity

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.

Space complexity

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).

1. Introduction

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.

Let's jump onto our first problem to see this pattern in action.

Problem Statement

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.

Example 1:

1. insertNum(3)
2. insertNum(1)
3. findMedian() -> output: 2
4. insertNum(5)
5. findMedian() -> output: 3
6. insertNum(4)
7. findMedian() -> output: 3.5

Solution

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?

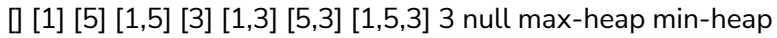
Assume 'x' is the median of a list. This means that half of the numbers in the list will be smaller than (or equal to) 'x' and half will be greater than (or equal to) 'x'. This leads us to an approach where we can divide the list into two halves: one half to store all the smaller numbers (let's call 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.

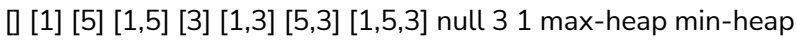
1. We can store the first half of numbers (i.e., `smallNumList`) in a **Max Heap**. We should use a **Max Heap** as we are interested in knowing the largest number in the first half.
2. We can store the second half of numbers (i.e., `largeNumList`) in a **Min Heap**, as we are interested in knowing the smallest number in the second half.
3. Inserting a number in a heap will take $O(\log N)$, 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:

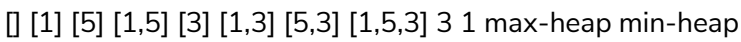
1. `insertNum(3)`: We can insert a number in the **Max Heap** (i.e. first half) if the number is smaller than the top (largest) number of the heap. After every insertion, we will balance the number of elements in both heaps, so that they have an equal number of elements. If the count of numbers is odd, 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 -> $(1+3)/2 = 2.0$
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**.

[] [1] [5] [1,5] [3] [1,3] [5,3] [1,5,3] max-heap min-heap 3 1 5

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**.

   [] [1] [5] [1,5] [3] [1,3] [5,3] [1,5,3] max-heap min-heap 3 1 4 5

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

```
using namespace std;


#include <iostream>
#include <queue>
#include <vector>


class MedianOfAStream {
 public:
  priority_queue<int> maxHeap;                    // containing first half of numbers
  priority_queue<int, vector<int>, greater<int>> minHeap;  // containing second half of numbers


  virtual void insertNum(int num) {
    if (maxHeap.empty() || maxHeap.top() >= num) {
      maxHeap.push(num);
    } else {
      minHeap.push(num);
    }
```

```
    // either both the heaps will have equal number of elements or max-heap will have one
    // more element than the min-heap
    if (maxHeap.size() > minHeap.size() + 1) {
      minHeap.push(maxHeap.top());
      maxHeap.pop();
    } else if (maxHeap.size() < minHeap.size()) {
      maxHeap.push(minHeap.top());
      minHeap.pop();
    }
  }

  virtual double findMedian() {
    if (maxHeap.size() == minHeap.size()) {
      // we have even number of elements, take the average of middle two elements
      return maxHeap.top() / 2.0 + minHeap.top() / 2.0;
    }
    // because max-heap will have one more element than the min-heap
    return maxHeap.top();
  }
};

int main(int argc, char *argv[]) {
  MedianOfAStream medianOfAStream;
  medianOfAStream.insertNum(3);
  medianOfAStream.insertNum(1);
  cout << "The median is: " << medianOfAStream.findMedian() << endl;
  medianOfAStream.insertNum(5);
  cout << "The median is: " << medianOfAStream.findMedian() << endl;
```

```
  medianOfAStream.insertNum(4);
  cout << "The median is: " << medianOfAStream.findMedian() << endl;
}
```

Time complexity

The time complexity of the `insertNum()` will be $O(\log N)$ due to the insertion in the heap. The time complexity of the `findMedian()` will be $O(1)$ as we can find the median from the top elements of the heaps.

Space complexity

The space complexity will be $O(N)$ because, as at any time, we will be storing all the numbers.

Problem Statement

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

Solution

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.

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <queue>

#include <vector>


// extending priority_queue to implement 'remove'

template <typename T, class Container = vector<T>,

        class Compare = less<typename Container::value_type>>

class priority_queue_with_remove : public priority_queue<T, Container, Compare> {

 public:

  bool remove(const T &valueToRemove) {

    auto itr = std::find(this->c.begin(), this->c.end(), valueToRemove);

    if (itr == this->c.end()) {

      return false;

    }

    this->c.erase(itr);

    // ideally we should not be rebuilding the heap as we are removing only one element
```

```
    // a custom implementation of priority queue can adjust only one element in O(logN)
    std::make_heap(this->c.begin(), this->c.end(), this->comp);
    return true;
  }
};

class SlidingWindowMedian {
 public:
  priority_queue_with_remove<int> maxHeap;
  priority_queue_with_remove<int, vector<int>, greater<int>> minHeap;

  virtual vector<double> findSlidingWindowMedian(const vector<int> &nums, int k) {
    vector<double> result(nums.size() - k + 1);
    for (int i = 0; i < nums.size(); i++) {
      if (maxHeap.size() == 0 || maxHeap.top() >= nums[i]) {
        maxHeap.push(nums[i]);
      } else {
        minHeap.push(nums[i]);
      }
      rebalanceHeaps();

      if (i - k + 1 >= 0) {  // if we have at least 'k' elements in the sliding window
        // add the median to the the result array
        if (maxHeap.size() == minHeap.size()) {
          // we have even number of elements, take the average of middle two elements
          result[i - k + 1] = maxHeap.top() / 2.0 + minHeap.top() / 2.0;
        } else {  // because max-heap will have one more element than the min-heap
          result[i - k + 1] = maxHeap.top();
        }
```

```
      // remove the element going out of the sliding window
      int elementToBeRemoved = nums[i - k + 1];
      if (elementToBeRemoved <= maxHeap.top()) {
        maxHeap.remove(elementToBeRemoved);
      } else {
        minHeap.remove(elementToBeRemoved);
      }
      rebalanceHeaps();
    }
  }
  return result;
 }

private:
 void rebalanceHeaps() {
  // either both the heaps will have equal number of elements or max-heap will have
  // one more element than the min-heap
  if (maxHeap.size() > minHeap.size() + 1) {
    minHeap.push(maxHeap.top());
    maxHeap.pop();
  } else if (maxHeap.size() < minHeap.size()) {
    maxHeap.push(minHeap.top());
    minHeap.pop();
  }
 }
};

int main(int argc, char *argv[]) {
```

```
  SlidingWindowMedian slidingWindowMedian;
  vector<double> result =
     slidingWindowMedian.findSlidingWindowMedian(vector<int>{1, 2, -1, 3, 5}, 2);
  cout << "Sliding window medians are: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  slidingWindowMedian = SlidingWindowMedian();
  result = slidingWindowMedian.findSlidingWindowMedian(vector<int>{1, 2, -1, 3, 5}, 3);
  cout << "Sliding window medians are: ";
  for (auto num : result) {
    cout << num << " ";
  }
}
```

Time complexity

The time complexity of our algorithm is $O(N*K)$ where 'N' is the total number of elements in the input array and 'K' is the size of the sliding window. This is due to the fact that we are going through all the 'N' numbers and, while doing so, we are doing two things:

1. Inserting/removing numbers from heaps of size 'K'. This will take $O(logK)$
2. Removing the element going out of the sliding window. This will take $O(K)$ as we will be searching this element in an array of size 'K' (i.e., a heap).

Space complexity

Ignoring the space needed for the output array, the space complexity will be $O(K)$ because, at any time, we will be storing all the numbers within the sliding window.

Problem Statement

Given a set of investment projects with their respective profits, we need to find the **most profitable projects**. We are given an initial capital and are allowed to invest only in a fixed number of projects. Our goal is to choose projects that give us the maximum profit. Write a function that returns the maximum total capital after selecting the most profitable projects.

We can start an investment project only when we have the required capital. Once a project is selected, we can assume that its profit has become our capital.

Example 1:

Input: Project Capitals=[0,1,2], Project Profits=[1,2,3], Initial Capital=1, Number of Projects=2
Output: 6
Explanation:

1. With initial capital of '1', we will start the second project which will give us profit of '2'. Once we selected our first project, our total capital will become 3 (profit + initial capital).
2. With '3' capital, we will select the third project, which will give us '3' profit.

After the completion of the two projects, our total capital will be 6 (1+2+3).

Example 2:

Input: Project Capitals=[0,1,2,3], Project Profits=[1,2,3,5], Initial Capital=0, Number of Projects=3
Output: 8
Explanation:

1. With '0' capital, we can only select the first project, bringing out capital to 1.
2. Next, we will select the second project, which will bring our capital to 3.
3. Next, we will select the fourth project, giving us a profit of 5.

After selecting the three projects, our total capital will be 8 (1+2+5).

Solution

While selecting projects we have two constraints:

1. We can select a project only when we have the required capital.
2. There is a maximum limit on how many projects we can select.

Since we don't have any constraint on time, we should choose a project, among the projects for which we have enough capital, which gives us a maximum profit. Following this greedy approach will give us the best solution.

While selecting a project, we will do two things:

1. Find all the projects that we can choose with the available capital.
2. From the list of projects in the 1st step, choose the project that gives us a maximum profit.

We can follow the **Two Heaps** approach similar to Find the Median of a Number Stream. Here are the steps of our algorithm:

1. Add all project capitals to a min-heap, so that we can select a project with the smallest capital requirement.
2. Go through the top projects of the min-heap and filter the projects that can be completed within our available capital. Insert the profits of all these projects into a max-heap, so that we can choose a project with the maximum profit.
3. Finally, select the top project of the max-heap for investment.
4. Repeat the 2nd and 3rd steps for the required number of projects.

```
using namespace std;


#include <iostream>

#include <queue>
```

```
#include <vector>

class MaximizeCapital {
 public:
  struct capitalCompare {
    bool operator()(const pair<int, int> &x, const pair<int, int> &y) { return x.first > y.first; }
  };

  struct profitCompare {
    bool operator()(const pair<int, int> &x, const pair<int, int> &y) { return y.first > x.first; }
  };

  static int findMaximumCapital(const vector<int> &capital, const vector<int> &profits,
                                int numberOfProjects, int initialCapital) {
    int n = profits.size();
    priority_queue<pair<int, int>, vector<pair<int, int>>, capitalCompare> minCapitalHeap;
    priority_queue<pair<int, int>, vector<pair<int, int>>, profitCompare> maxProfitHeap;

    // insert all project capitals to a min-heap
    for (int i = 0; i < n; i++) {
      minCapitalHeap.push(make_pair(capital[i], i));
    }

    // let's try to find a total of 'numberOfProjects' best projects
    int availableCapital = initialCapital;
    for (int i = 0; i < numberOfProjects; i++) {
      // find all projects that can be selected within the available capital and insert them in a
      // max-heap
      while (!minCapitalHeap.empty() && minCapitalHeap.top().first <= availableCapital) {
```

```
      auto capitalIndex = minCapitalHeap.top().second;
      minCapitalHeap.pop();
      maxProfitHeap.push(make_pair(profits[capitalIndex], capitalIndex));
    }

    // terminate if we are not able to find any project that can be completed within the available
    // capital
    if (maxProfitHeap.empty()) {
      break;
    }

    // select the project with the maximum profit
    availableCapital += maxProfitHeap.top().first;
    maxProfitHeap.pop();
  }

  return availableCapital;
 }
};

int main(int argc, char *argv[]) {
 int result =
    MaximizeCapital::findMaximumCapital(vector<int>{0, 1, 2}, vector<int>{1, 2, 3}, 2, 1);
 cout << "Maximum capital: " << result << endl;
 result =
    MaximizeCapital::findMaximumCapital(vector<int>{0, 1, 2, 3}, vector<int>{1, 2, 3, 5}, 3, 0);
 cout << "Maximum capital: " << result << endl;
}
```

Time complexity

Since, at the most, all the projects will be pushed to both the heaps once, the time complexity of our algorithm is $O(NlogN + KlogN)$, $O(NlogN+KlogN)$, where 'N' is the total number of projects and 'K' is the number of projects we are selecting.

Space complexity

The space complexity will be $O(N)$ $O(N)$ because we will be storing all the projects in the heaps.

Next Interval (hard)

Given an array of intervals, find the next interval of each interval. In a list of intervals, for an interval 'i' its next interval 'j' will have the smallest 'start' greater than or equal to the 'end' of 'i'.

Write a function to return an array containing indices of the next interval of each input interval. If there is no next interval of a given interval, return -1. It is given that none of the intervals have the same start point.

Example 1:

Input: Intervals [[2,3], [3,4], [5,6]]
Output: [1, 2, -1]
Explanation: The next interval of [2,3] is [3,4] having index '1'. Similarly, the next interval of [3,4] is [5,6] having index '2'. There is no next interval for [5,6] hence we have '-1'.

Example 2:

Input: Intervals [[3,4], [1,5], [4,6]]
Output: [2, -1, -1]
Explanation: The next interval of [3,4] is [4,6] which has index '2'. There is no next interval for [1,5] and [4,6].

Solution

A brute force solution could be to take one interval at a time and go through all the other intervals to find the next interval. This algorithm will take $O(N^2)$ where 'N' is the total number of intervals. Can we do better than that?

We can utilize the **Two Heaps** approach. We can push all intervals into two heaps: one heap to sort the intervals on maximum start time (let's call it `maxStartHeap`) and the other on maximum end time (let's call it `maxEndHeap`). We can then iterate through all intervals of the `maxEndHeap' to find their next interval. Our algorithm will have the following steps:

1. Take out the top (having highest end) interval from the `maxEndHeap` to find its next interval. Let's call this interval `topEnd`.
2. Find an interval in the `maxStartHeap` with the closest start greater than or equal to the start of `topEnd`. Since `maxStartHeap` is sorted by 'start' of intervals, it is easy to find the interval with the highest 'start'. Let's call this interval `topStart`.
3. Add the index of `topStart` in the result array as the next interval of `topEnd`. If we can't find the next interval, add '-1' in the result array.
4. Put the `topStart` back in the `maxStartHeap`, as it could be the next interval of other intervals.
5. Repeat the steps 1-4 until we have no intervals left in `maxEndHeap`.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class Interval {

 public:

  int start = 0;
```

```
  int end = 0;

  Interval(int start, int end) {
    this->start = start;
    this->end = end;
  }
};

class NextInterval {
 public:
  struct startCompare {
    bool operator()(const pair<Interval, int> &x, const pair<Interval, int> &y) {
      return y.first.start > x.first.start;
    }
  };

  struct endCompare {
    bool operator()(const pair<Interval, int> &x, const pair<Interval, int> &y) {
      return y.first.end > x.first.end;
    }
  };

  static vector<int> findNextInterval(const vector<Interval> &intervals) {
    int n = intervals.size();
    // heap for finding the maximum start
    priority_queue<pair<Interval, int>, vector<pair<Interval, int>>, startCompare> maxStartHeap;
    // heap for finding the minimum end
    priority_queue<pair<Interval, int>, vector<pair<Interval, int>>, endCompare> maxEndHeap;
```

```
  vector<int> result(n);
  for (int i = 0; i < intervals.size(); i++) {
    maxStartHeap.push(make_pair(intervals[i], i));
    maxEndHeap.push(make_pair(intervals[i], i));
  }

  // go through all the intervals to find each interval's next interval
  for (int i = 0; i < n; i++) {
    // let's find the next interval of the interval which has the highest 'end'
    auto topEnd = maxEndHeap.top();
    maxEndHeap.pop();

    result[topEnd.second] = -1;  // defaults to -1
    if (maxStartHeap.top().first.start >= topEnd.first.end) {
      auto topStart = maxStartHeap.top();
      maxStartHeap.pop();
      // find the the interval that has the closest 'start'
      while (!maxStartHeap.empty() && maxStartHeap.top().first.start >= topEnd.first.end) {
        topStart = maxStartHeap.top();
        maxStartHeap.pop();
      }
      result[topEnd.second] = topStart.second;
      // put the interval back as it could be the next interval of other intervals
      maxStartHeap.push(topStart);
    }
  }

  return result;
}
```

```
};


int main(int argc, char *argv[]) {
  vector<Interval> intervals = {{2, 3}, {3, 4}, {5, 6}};
  vector<int> result = NextInterval::findNextInterval(intervals);
  cout << "Next interval indices are: ";
  for (auto index : result) {
    cout << index << " ";
  }
  cout << endl;


  intervals = {{3, 4}, {1, 5}, {4, 6}};
  result = NextInterval::findNextInterval(intervals);
  cout << "Next interval indices are: ";
  for (auto index : result) {
    cout << index << " ";
  }
}
```

Time complexity

The time complexity of our algorithm will be $O(NlogN)$,$O(NlogN)$, where 'N' is the total number of intervals.

Space complexity

The space complexity will be $O(N)O(N)$ because we will be storing all the intervals in the heaps.

1. Introduction

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.

Let's jump onto our first problem to develop an understanding of this pattern.

Problem Statement

Given a set with distinct elements, find all of its distinct subsets.

Example 1:

Input: [1, 3]
Output: [], [1], [3], [1,3]

Example 2:

Input: [1, 5, 3]
Output: [], [1], [5], [3], [1,5], [1,3], [5,3], [1,5,3]

Solution

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]**].

Here is the visual representation of the above steps:

Since the input set has distinct elements, the above steps will ensure that we will not have any duplicate subsets.

```
using namespace std;


#include <iostream>
#include <vector>


class Subsets {
 public:
  static vector<vector<int>> findSubsets(const vector<int>& nums) {
    vector<vector<int>> subsets;
    // start by adding the empty subset
    subsets.push_back(vector<int>());
    for (auto currentNumber : nums) {
      // we will take all existing subsets and insert the current number in them to create new
      // subsets
      int n = subsets.size();
      for (int i = 0; i < n; i++) {
        // create a new subset from the existing subset and insert the current element to it
        vector<int> set(subsets[i]);
        set.push_back(currentNumber);
        subsets.push_back(set);
      }
    }
    return subsets;
```

```
  }
};

int main(int argc, char* argv[]) {
  vector<vector<int>> result = Subsets::findSubsets(vector<int>{1, 3});
  cout << "Here is the list of subsets: " << endl;
  for (auto vec : result) {
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << endl;
  }

  result = Subsets::findSubsets(vector<int>{1, 5, 3});
  cout << "Here is the list of subsets: " << endl;
  for (auto vec : result) {
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << endl;
  }
}
```

Time complexity

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 $O(N*2^N)$.

Space complexity

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 $O(N*2^N)$.

Problem Statement

Given a set of numbers that might contain duplicates, find all of its distinct subsets.

Example 1:

Input: [1, 3, 3]
Output: [], [1], [3], [1,3], [3,3], [1,3,3]

Example 2:

Input: [1, 5, 3, 3]
Output: [], [1], [5], [3], [1,5], [1,3], [5,3], [1,5,3], [3,3], [1,3,3], [3,3,5], [1,5,3,3]

Solution

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 Example-2 mentioned above 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]]

Here is the visual representation of the above steps:

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <vector>


class SubsetWithDuplicates {

 public:

  static vector<vector<int>> findSubsets(vector<int>& nums) {

    sort(nums.begin(), nums.end());  // sort the numbers to handle duplicates

    vector<vector<int>> subsets;
```

```
    subsets.push_back(vector<int>());
    int startIndex = 0, endIndex = 0;
    for (int i = 0; i < nums.size(); i++) {
      startIndex = 0;
      // if current and the previous elements are same, create new subsets only from the subsets
      // added in the previous step
      if (i > 0 && nums[i] == nums[i - 1]) {
        startIndex = endIndex + 1;
      }
      endIndex = subsets.size() - 1;
      for (int j = startIndex; j <= endIndex; j++) {
        // create a new subset from the existing subset and add the current element to it
        vector<int> set(subsets[j]);
        set.push_back(nums[i]);
        subsets.push_back(set);
      }
    }
    return subsets;
  }
};

int main(int argc, char* argv[]) {
  vector<int> vec = {1, 3, 3};
  vector<vector<int>> result = SubsetWithDuplicates::findSubsets(vec);
  cout << "Here is the list of subsets: " << endl;
  for (auto vec : result) {
    for (auto num : vec) {
      cout << num << " ";
    }
```

```
    cout << endl;
  }


  vec = {1, 5, 3, 3};
  result = SubsetWithDuplicates::findSubsets(vec);
  cout << "Here is the list of subsets: " << endl;
  for (auto vec : result) {
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << endl;
  }
}
```

Time complexity

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 $O(N*2^N)$.

Space complexity

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 $O(N*2^N)$.

Problem Statement

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.

Example 1:

Input: [1,3,5]
Output: [1,3,5], [1,5,3], [3,1,5], [3,5,1], [5,1,3], [5,3,1]

Solution

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-1 mentioned above 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]

Here is the visual representation of this algorithm:

```
using namespace std;


#include <iostream>
#include <queue>
#include <vector>


class Permutations {
 public:
  static vector<vector<int>> findPermutations(const vector<int>& nums) {
    vector<vector<int>> result;
    queue<vector<int>> permutations;
    permutations.push(vector<int>());
    for (auto currentNumber : nums) {
      // we will take all existing permutations and add the current number to create new
      // permutations
      int n = permutations.size();
      for (int i = 0; i < n; i++) {
        vector<int> oldPermutation = permutations.front();
        permutations.pop();
        // create a new permutation by adding the current number at every position
        for (int j = 0; j <= oldPermutation.size(); j++) {
          vector<int> newPermutation(oldPermutation);
          newPermutation.insert(newPermutation.begin() + j, currentNumber);
          if (newPermutation.size() == nums.size()) {
            result.push_back(newPermutation);
```

```
        } else {
          permutations.push(newPermutation);
        }
       }
     }
    }
    return result;
  }
};

int main(int argc, char* argv[]) {
  vector<vector<int>> result = Permutations::findPermutations(vector<int>{1, 3, 5});
  cout << "Here are all the permutations: " << endl;
  for (auto vec : result) {
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << endl;
  }
}
```

Time complexity

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 on line 17 (line 23 for C++ solution). To insert a number into a permutation of size 'N' will take $O(N)$,$O(N)$, which makes the overall time complexity of our algorithm $O(N*N!)$$O(N*N!)$.

Space complexity

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 NN elements will be O(N*N!)$O(N*N!)$.

Recursive Solution

Here is the recursive algorithm following a similar approach:

```
using namespace std;


#include <iostream>

#include <vector>


class PermutationsRecursive {

 public:

  static vector<vector<int>> generatePermutations(const vector<int> &nums) {

    vector<vector<int>> result;

    vector<int> currentPermutation;

    generatePermutationsRecursive(nums, 0, currentPermutation, result);

    return result;

  }


 private:

  static void generatePermutationsRecursive(const vector<int> &nums, int index,

                                            vector<int> &currentPermutation, vector<vector<int>> &result) {

    if (index == nums.size()) {

      result.push_back(currentPermutation);

    } else {
```

```
    // create a new permutation by adding the current number at every position
    for (int i = 0; i <= currentPermutation.size(); i++) {
      vector<int> newPermutation(currentPermutation);
      newPermutation.insert(newPermutation.begin() + i, nums[index]);
      generatePermutationsRecursive(nums, index + 1, newPermutation, result);
    }
  }
 }
};

int main(int argc, char *argv[]) {
 vector<vector<int>> result = PermutationsRecursive::generatePermutations(vector<int>{1, 3, 5});
 cout << "Here are all the permutations: " << endl;
 for (auto vec : result) {
  for (auto num : vec) {
   cout << num << " ";
  }
  cout << endl;
 }
}
```

Problem Statement

Given a string, find all of its permutations preserving the character sequence but changing case.

Example 1:

Input: "ad52"
Output: "ad52", "Ad52", "aD52", "AD52"

Example 2:

Input: "ab7c"
Output: "ab7c", "Ab7c", "aB7c", "AB7c", "ab7C", "Ab7C", "aB7C", "AB7C"

Solution

This problem follows the Subsets pattern and can be mapped to Permutations.

Let's take Example-2 mentioned above to generate all the permutations. Following a **BFS** approach, we will consider one character at a time. Since we need to preserve the character sequence, we can start with the actual string and process each character (i.e., make it upper-case or lower-case) one by one:

1. Starting with the actual string: `"ab7c"`
2. Processing the first character ('a'), we will get two permutations: `"ab7c", "Ab7c"`
3. Processing the second character ('b'), we will get four permutations: `"ab7c", "Ab7c", "aB7c", "AB7c"`
4. Since the third character is a digit, we can skip it.
5. Processing the fourth character ('c'), we will get a total of eight permutations: `"ab7c", "Ab7c", "aB7c", "AB7c", "ab7C", "Ab7C", "aB7C", "AB7C"`

Let's analyze the permutations in the 3rd and the 5th step. How can we generate the permutations in the 5th step from the permutations in the 3rd step?

If we look closely, we will realize that in the 5th step, when we processed the new character ('c'), we took all the permutations of the previous step (3rd) and changed the case of the letter ('c') in them to create four new permutations.

Here is the visual representation of this algorithm:

```
using namespace std;


#include <iostream>

#include <string>

#include <vector>


class LetterCaseStringPermutation {

 public:
```

```
  static vector<string> findLetterCaseStringPermutations(const string& str) {
    vector<string> permutations;
    if (str == "") {
      return permutations;
    }

    permutations.push_back(str);
    // process every character of the string one by one
    for (int i = 0; i < str.length(); i++) {
      if (isalpha(str[i])) {  // only process characters, skip digits
        // we will take all existing permutations and change the letter case appropriately
        int n = permutations.size();
        for (int j = 0; j < n; j++) {
          vector<char> chs(permutations[j].begin(), permutations[j].end());
          // if the current character is in upper case change it to lower case or vice versa
          if (isupper(chs[i])) {
            chs[i] = tolower(chs[i]);
          } else {
            chs[i] = toupper(chs[i]);
          }
          permutations.push_back(string(chs.begin(), chs.end()));
        }
      }
    }
    return permutations;
  }
};

int main(int argc, char* argv[]) {
```

```
  vector<string> result = LetterCaseStringPermutation::findLetterCaseStringPermutations("ad52");
  cout << "String permutations are: ";
  for (auto str : result) {
    cout << str << " ";
  }
  cout << endl;

  result = LetterCaseStringPermutation::findLetterCaseStringPermutations("ab7c");
  cout << "String permutations are: ";
  for (auto str : result) {
    cout << str << " ";
  }
  cout << endl;
}
```

Time complexity

Since we can have 2^N permutations at the most and while processing each permutation we convert it into a character array, the overall time complexity of the algorithm will be $O(N*2^N)$.

Space complexity

All the additional space used by our algorithm is for the output list. Since we can have a total of $O(2^N)$ permutations, the space complexity of our algorithm is $O(N*2^N)$.

Problem Statement

For a given number ‘N’, write a function to generate all combination of ‘N’ pairs of balanced parentheses.

Example 1:

Input: N=2
Output: (()), ()()

Example 2:

Input: N=3
Output: ((())), (()()), (())(), ()(()), ()()()

Solution

This problem follows the Subsets pattern and can be mapped to Permutations. We can follow a similar BFS approach.

Let's take Example-2 mentioned above to generate all the combinations of balanced parentheses. Following a BFS approach, we will keep adding open parentheses `(` or close parentheses `)`. At each step we need to keep two things in mind:

- We can't add more than 'N' open parenthesis.
- To keep the parentheses balanced, we can add a close parenthesis `)` only when we have already added enough open parenthesis `(`. For this, we can keep a count of open and close parenthesis with every combination.

Following this guideline, let's generate parentheses for N=3:

1. Start with an empty combination: ""
2. At every step, let's take all combinations of the previous step and add `(` or `)` keeping the above-mentioned two rules in mind.
3. For the empty combination, we can add `(` since the count of open parenthesis will be less than 'N'. We can't add `)` as we don't have an equivalent open parenthesis, so our list of combinations will now be: "("
4. For the next iteration, let's take all combinations of the previous set. For "(" we can add another `(` to it since the count of open parenthesis will be less than 'N'. We can also add `)` as we do have an equivalent open parenthesis, so our list of combinations will be: "((", "()"
5. In the next iteration, for the first combination "((", we can add another `(` to it as the count of open parenthesis will be less than 'N', we can also add `)` as we do have an equivalent open parenthesis. This gives us two new combinations: "(((" and "(()". For the second combination "()", we can add another `(` to it since the count of open parenthesis will

be less than ‘N’. We can’t add) as we don’t have an equivalent open parenthesis, so our list of combinations will be: “(((“, “(()”, ()("

6. Following the same approach, next we will get the following list of combinations: “((()”, “(()(“, “(())”, “)((“, “)()”
7. Next we will get: “((()))”, “(()()”, “(())(“, “)(()”, “)()(“
8. Finally, we will have the following combinations of balanced parentheses: “((()))”, “(()())”, “(())()”, “)(())”, “)()()”
9. We can’t add more parentheses to any of the combinations, so we stop here.

Here is the visual representation of this algorithm:

```
using namespace std;


#include <iostream>

#include <queue>

#include <string>

#include <vector>


class ParenthesesString {

 public:

  string str;

  int openCount = 0;   // open parentheses count

  int closeCount = 0;  // close parentheses count


  ParenthesesString(const string &s, int openCount, int closeCount) {

    this->str = s;

    this->openCount = openCount;

    this->closeCount = closeCount;

  }

};
```

```
class GenerateParentheses {
 public:
  static vector<string> generateValidParentheses(int num) {
    vector<string> result;
    queue<ParenthesesString> queue;
    queue.push({"", 0, 0});
    while (!queue.empty()) {
      ParenthesesString ps = queue.front();
      queue.pop();
      // if we've reached the maximum number of open and close parentheses, add to the result
      if (ps.openCount == num && ps.closeCount == num) {
        result.push_back(ps.str);
      } else {
        if (ps.openCount < num) {  // if we can add an open parentheses, add it
          queue.push({ps.str + "(", ps.openCount + 1, ps.closeCount});
        }

        if (ps.openCount > ps.closeCount) {  // if we can add a close parentheses, add it
          queue.push({ps.str + ")", ps.openCount, ps.closeCount + 1});
        }
      }
    }
    return result;
  }
};

int main(int argc, char *argv[]) {
  vector<string> result = GenerateParentheses::generateValidParentheses(2);
  cout << "All combinations of balanced parentheses are: ";
```

```
  for (auto str : result) {
    cout << str << " ";
  }
  cout << endl;

  result = GenerateParentheses::generateValidParentheses(3);
  cout << "All combinations of balanced parentheses are: ";
  for (auto str : result) {
    cout << str << " ";
  }
  cout << endl;
}
```

Time complexity

Let's try to estimate how many combinations we can have for 'N' pairs of balanced parentheses. If we don't care for the ordering - *that* `)` *can only come after* `(` - then we have two options for every position, i.e., either put open parentheses or close parentheses. This means we can have a maximum of 2^N combinations. Because of the ordering, the actual number will be less than 2^N.

If you see the visual representation of Example-2 closely you will realize that, in the worst case, it is equivalent to a binary tree, where each node will have two children. This means that we will have 2^N leaf nodes and $2^N - 1$ intermediate nodes. So the total number of elements pushed to the queue will be $2^N + 2^N - 1$, which is asymptotically equivalent to $O(2^N)$. While processing each element, we do need to concatenate the current string with `(` or `)`. This operation will take $O(N)$, so the overall time complexity of our algorithm will be $O(N*2^N)$. This is not completely accurate but reasonable enough to be presented in the interview.

The actual time complexity ($O(4^n/\sqrt{n})$) is bounded by the Catalan number and is beyond the scope of a coding interview. See more details here.

Space complexity

All the additional space used by our algorithm is for the output list. Since we can't have more than $O(2^N)$ combinations, the space complexity of our algorithm is $O(N*2^N)$.

Recursive Solution

Here is the recursive algorithm following a similar approach:

```
using namespace std;


#include <iostream>

#include <string>

#include <vector>


class GenerateParenthesesRecursive {

 public:

  static vector<string> generateValidParentheses(int num) {

    vector<string> result;

    vector<char> parenthesesString(2 * num);

    generateValidParenthesesRecursive(num, 0, 0, parenthesesString, 0, result);

    return result;

  }


 private:

  static void generateValidParenthesesRecursive(int num, int openCount, int closeCount,

                                                vector<char> &parenthesesString, int index,

                                                vector<string> &result) {
```

```
    // if we've reached the maximum number of open and close parentheses, add to the result
    if (openCount == num && closeCount == num) {
      result.push_back(string(parenthesesString.begin(), parenthesesString.end()));
    } else {
      if (openCount < num) {  // if we can add an open parentheses, add it
        parenthesesString[index] = '(';
        generateValidParenthesesRecursive(num, openCount + 1, closeCount, parenthesesString,
                                          index + 1, result);
      }

      if (openCount > closeCount) {  // if we can add a close parentheses, add it
        parenthesesString[index] = ')';
        generateValidParenthesesRecursive(num, openCount, closeCount + 1, parenthesesString,
                                          index + 1, result);
      }
    }
  }
};

int main(int argc, char *argv[]) {
  vector<string> result = GenerateParenthesesRecursive::generateValidParentheses(2);
  cout << "All combinations of balanced parentheses are: ";
  for (auto str : result) {
    cout << str << " ";
  }
  cout << endl;

  result = GenerateParenthesesRecursive::generateValidParentheses(3);
  cout << "All combinations of balanced parentheses are: ";
```

```
  for (auto str : result) {
    cout << str << " ";
  }
  cout << endl;
}
```

Problem Statement

Given a word, write a function to generate all of its unique generalized abbreviations.

Generalized abbreviation of a word can be generated by replacing each substring of the word by the count of characters in the substring. Take the example of “ab” which has four substrings: “”, “a”, “b”, and “ab”. After replacing these substrings in the actual word by the count of characters we get all the generalized abbreviations: “ab”, “1b”, “a1”, and “2”.

Example 1:

Input: "BAT"
Output: "BAT", "BA1", "B1T", "B2", "1AT", "1A1", "2T", "3"

Example 2:

Input: "code"
Output: "code", "cod1", "co1e", "co2", "c1de", "c1d1", "c2e", "c3", "1ode", "1od1", "1o1e", "1o2", "2de", "2d1", "3e", "4"

Solution

This problem follows the Subsets pattern and can be mapped to Balanced Parentheses. We can follow a similar BFS approach.

Let’s take Example-1 mentioned above to generate all unique generalized abbreviations. Following a BFS approach, we will abbreviate one character at a time. At each step we have two options:

- Abbreviate the current character, or
- Add the current character to the output and skip abbreviation.

Following these two rules, let’s abbreviate `BAT`:

1. Start with an empty word: “”
2. At every step, we will take all the combinations from the previous step and apply the two abbreviation rules to the next character.
3. Take the empty word from the previous step and add the first character to it. We can either abbreviate the character or add it (by skipping abbreviation). This gives us two new words: `_`, `B`.
4. In the next iteration, let’s add the second character. Applying the two rules on `_` will give us `_ _` and `1A`. Applying the above rules to the other combination `B` gives us `B_` and `BA`.
5. The next iteration will give us: `_ _ _`, `2T`, `1A_`, `1AT`, `B _ _`, `B1T`, `BA_`, `BAT`
6. The final iteration will give us:`3`, `2T`, `1A1`, `1AT`, `B2`, `B1T`, `BA1`, `BAT`

Here is the visual representation of this algorithm:

```
using namespace std;


#include <iostream>

#include <queue>

#include <string>

#include <vector>


class AbbreviatedWord {

 public:

  string str;

  int start = 0;

  int count = 0;


  AbbreviatedWord(string str, int start, int count) {

    this->str = str;

    this->start = start;

    this->count = count;

  }

};
```

```
class GeneralizedAbbreviation {
 public:
  static vector<string> generateGeneralizedAbbreviation(const string &word) {
    int wordLen = word.length();
    vector<string> result;
    queue<AbbreviatedWord> queue;
    queue.push({"", 0, 0});
    while (!queue.empty()) {
      AbbreviatedWord abWord = queue.front();
      queue.pop();
      if (abWord.start == wordLen) {
        if (abWord.count != 0) {
          abWord.str += to_string(abWord.count);
        }
        result.push_back(abWord.str);
      } else {
        // continue abbreviating by incrementing the current abbreviation count
        queue.push({abWord.str, abWord.start + 1, abWord.count + 1});

        // restart abbreviating, append the count and the current character to the string
        if (abWord.count != 0) {
          abWord.str += to_string(abWord.count);
        }
        abWord.str += word[abWord.start];
        queue.push({abWord.str, abWord.start + 1, 0});
      }
    }
```

```
    return result;
  }
};

int main(int argc, char *argv[]) {
  vector<string> result = GeneralizedAbbreviation::generateGeneralizedAbbreviation("BAT");
  cout << "Generalized abbreviation are: ";
  for (auto str : result) {
    cout << str << " ";
  }
  cout << endl;

  result = GeneralizedAbbreviation::generateGeneralizedAbbreviation("code");
  cout << "Generalized abbreviation are: ";
  for (auto str : result) {
    cout << str << " ";
  }
  cout << endl;
}
```

Time complexity

Since we had two options for each character, we will have a maximum of 2^N combinations. If you see the visual representation of Example-1 closely you will realize that it is equivalent to a binary tree, where each node has two children. This means that we will have 2^N leaf nodes and 2^N-1 intermediate nodes, so the total number of elements pushed to the queue will be $2^N + 2^N-1$, which is asymptotically equivalent to $O(2^N)$. While processing each element, we do need to concatenate the current string with a character. This operation will take $O(N)$, so the overall time complexity of our algorithm will be $O(N*2^N)$.

Space complexity

All the additional space used by our algorithm is for the output list. Since we can't have more than $O(2^N)$ combinations, the space complexity of our algorithm is $O(N*2^N)$.

Recursive Solution

Here is the recursive algorithm following a similar approach:

```
using namespace std;


#include <iostream>

#include <string>

#include <vector>


class GeneralizedAbbreviationRecursive {

 public:

  static vector<string> generateGeneralizedAbbreviation(const string &word) {

    vector<string> result;

    string abWord = "";

    generateAbbreviationRecursive(word, abWord, 0, 0, result);

    return result;

  }


 private:

  static void generateAbbreviationRecursive(const string &word, string &abWord, int start,

                                            int count, vector<string> &result) {

    if (start == word.length()) {

      if (count != 0) {

        abWord += to_string(count);
```

```
    }
    result.push_back(abWord);
   } else {
    // continue abbreviating by incrementing the current abbreviation count
    string newWord(abWord);
    generateAbbreviationRecursive(word, newWord, start + 1, count + 1, result);

    // restart abbreviating, append the count and the current character to the string
    if (count != 0) {
     abWord += to_string(count);
    }
    abWord += word[start];
    generateAbbreviationRecursive(word, abWord, start + 1, 0, result);
   }
  }
};

int main(int argc, char *argv[]) {
  vector<string> result = GeneralizedAbbreviationRecursive::generateGeneralizedAbbreviation("BAT");
  cout << "Generalized abbreviation are: ";
  for (auto str : result) {
   cout << str << " ";
  }
  cout << endl;

  result = GeneralizedAbbreviationRecursive::generateGeneralizedAbbreviation("code");
  cout << "Generalized abbreviation are: ";
  for (auto str : result) {
   cout << str << " ";
```

```
  }
  cout << endl;
}
```

Evaluate Expression (hard)

Given an expression containing digits and operations (+, -, *), find all possible ways in which the expression can be evaluated by grouping the numbers and operators using parentheses.

Example 1:

Input: "1+2*3"
Output: 7, 9
Explanation: 1+(2*3) => 7 and (1+2)*3 => 9

Example 2:

Input: "2*3-4-5"
Output: 8, -12, 7, -7, -3
Explanation: 2*(3-(4-5)) => 8, 2*(3-4-5) => -12, 2*3-(4-5) => 7, 2*(3-4)-5 => -7, (2*3)-4-5 => -3

Solution

This problem follows the Subsets pattern and can be mapped to Balanced Parentheses. We can follow a similar BFS approach.

Let's take Example-1 mentioned above to generate different ways to evaluate the expression.

1. We can iterate through the expression character-by-character.
2. we can break the expression into two halves whenever we get an operator (+, -, *).
3. The two parts can be calculated by recursively calling the function.
4. Once we have the evaluation results from the left and right halves, we can combine them to produce all results.

```
using namespace std;
```

```
#include <cctype>
#include <iostream>
#include <string>
#include <vector>

class EvaluateExpression {
 public:
  static vector<int> diffWaysToEvaluateExpression(const string& input) {
    vector<int> result;
    // base case: if the input string is a number, parse and add it to output.
    if (input.find("+") == string::npos && input.find("-") == string::npos &&
        input.find("*") == string::npos) {
      result.push_back(stoi(input));
    } else {
      for (int i = 0; i < input.length(); i++) {
        char chr = input[i];
        if (!isdigit(chr)) {
          // break the equation here into two parts and make recursively calls
          vector<int> leftParts = diffWaysToEvaluateExpression(input.substr(0, i));
          vector<int> rightParts = diffWaysToEvaluateExpression(input.substr(i + 1));
          for (auto part1 : leftParts) {
            for (auto part2 : rightParts) {
              if (chr == '+') {
                result.push_back(part1 + part2);
              } else if (chr == '-') {
                result.push_back(part1 - part2);
              } else if (chr == '*') {
                result.push_back(part1 * part2);
```

```
          }
        }
      }
    }
  }
  return result;
 }
};

int main(int argc, char* argv[]) {
 vector<int> result = EvaluateExpression::diffWaysToEvaluateExpression("1+2*3");
 cout << "Expression evaluations: ";
 for (auto num : result) {
  cout << num << " ";
 }
 cout << endl;

 result = EvaluateExpression::diffWaysToEvaluateExpression("2*3-4-5");
 cout << "Expression evaluations: ";
 for (auto num : result) {
  cout << num << " ";
 }
 cout << endl;
}
```

Time complexity

The time complexity of this algorithm will be exponential and will be similar to Balanced Parentheses. Estimated time complexity will

be $O(N*2^N)$ but the actual time complexity ($O(4^n/\sqrt{n})$) is bounded by the Catalan number and is beyond the scope of a coding interview. See more details here.

Space complexity

The space complexity of this algorithm will also be exponential, estimated at $O(2^N)$ though the actual will be ($O(4^n/\sqrt{n})$.

Memoized version

The problem has overlapping subproblems, as our recursive calls can be evaluating the same sub-expression multiple times. To resolve this, we can use memoization and store the intermediate results in a **HashMap**. In each function call, we can check our map to see if we have already evaluated this sub-expression before. Here is the memoized version of our algorithm; please see highlighted changes:

```
using namespace std;


#include <iostream>

#include <string>

#include <unordered_map>

#include <vector>


class EvaluateExpression {

 // memoization map

public:

 unordered_map<string, vector<int>> map = unordered_map<string, vector<int>>();


 virtual vector<int> diffWaysToEvaluateExpression(const string& input) {

  if (map.find(input) != map.end()) {

   return map[input];

  }
```

```
    vector<int> result;
    // base case: if the input string is a number, parse and return it.
    if (input.find("+") == string::npos && input.find("-") == string::npos &&
        input.find("*") == string::npos) {
      result.push_back(stoi(input));
    } else {
      for (int i = 0; i < input.length(); i++) {
        char chr = input[i];
        if (!isdigit(chr)) {
          vector<int> leftParts = diffWaysToEvaluateExpression(input.substr(0, i));
          vector<int> rightParts = diffWaysToEvaluateExpression(input.substr(i + 1));
          for (auto part1 : leftParts) {
            for (auto part2 : rightParts) {
              if (chr == '+') {
                result.push_back(part1 + part2);
              } else if (chr == '-') {
                result.push_back(part1 - part2);
              } else if (chr == '*') {
                result.push_back(part1 * part2);
              }
            }
          }
        }
      }
      map[input] = result;
      return result;
    }
};
```

```
int main(int argc, char* argv[]) {
  EvaluateExpression ee;
  vector<int> result = ee.diffWaysToEvaluateExpression("1+2*3");
  cout << "Expression evaluations: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  EvaluateExpression ee1;
  result = ee1.diffWaysToEvaluateExpression("2*3-4-5");
  cout << "Expression evaluations: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Structurally Unique Binary Search Trees (hard)

Given a number ‘n’, write a function to return all structurally unique Binary Search Trees (BST) that can store values 1 to ‘n’?

Example 1:

Input: 2
Output: List containing root nodes of all structurally unique BSTs.
Explanation: Here are the 2 structurally unique BSTs storing all numbers from 1 to 2:

1 2 2 1

Example 2:

Input: 3
Output: List containing root nodes of all structurally unique BSTs.
Explanation: Here are the 5 structurally unique BSTs storing all numbers from 1 to 3:

1 2

Solution

This problem follows the Subsets pattern and is quite similar to Evaluate Expression. Following a similar approach, we can iterate from 1 to 'n' and consider each number as the root of a tree. All smaller numbers will make up the left sub-tree and bigger numbers will make up the right sub-tree. We will make recursive calls for the left and right sub-trees

```
using namespace std;


#include <iostream>

#include <vector>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;


  TreeNode(int x) { val = x; }

};


class UniqueTrees {

 public:

  static vector<TreeNode *> findUniqueTrees(int n) {

    if (n <= 0) {

      return vector<TreeNode *>();

    }
```

```
  return findUniqueTreesRecursive(1, n);
}

static vector<TreeNode *> findUniqueTreesRecursive(int start, int end) {
  vector<TreeNode *> result;
  // base condition, return 'null' for an empty sub-tree
  // consider n=1, in this case we will have start=end=1, this means we should have only one tree
  // we will have two recursive calls, findUniqueTreesRecursive(1, 0) & (2, 1)
  // both of these should return 'null' for the left and the right child
  if (start > end) {
    result.push_back(nullptr);
    return result;
  }

  for (int i = start; i <= end; i++) {
    // making 'i' root of the tree
    vector<TreeNode *> leftSubtrees = findUniqueTreesRecursive(start, i - 1);
    vector<TreeNode *> rightSubtrees = findUniqueTreesRecursive(i + 1, end);
    for (auto leftTree : leftSubtrees) {
      for (auto rightTree : rightSubtrees) {
        TreeNode *root = new TreeNode(i);
        root->left = leftTree;
        root->right = rightTree;
        result.push_back(root);
      }
    }
  }
  return result;
}
```

```
};


int main(int argc, char *argv[]) {
  vector<TreeNode *> result = UniqueTrees::findUniqueTrees(2);
  cout << "Total trees: " << result.size() << endl;


  result = UniqueTrees::findUniqueTrees(3);
  cout << "Total trees: " << result.size() << endl;
}
```

Time complexity

The time complexity of this algorithm will be exponential and will be similar to Balanced Parentheses. Estimated time complexity will be O(n*2^n)$O(n*2^n)$ but the actual time complexity (O(4^n/\sqrt{n})$O(4^n/\sqrt{n})$) is bounded by the Catalan number and is beyond the scope of a coding interview. See more details here.

Space complexity

The space complexity of this algorithm will be exponential too, estimated at O(2^n)$O(2^n)$, but the actual will be (O(4^n/\sqrt{n})$O(4^n/\sqrt{n})$.

Memoized version

Since our algorithm has overlapping subproblems, can we use memoization to improve it? We could, but every time we return the result of a subproblem from the cache, we have to clone the result list because these trees will be used as the left or right child of a tree. This cloning is equivalent to reconstructing the trees, therefore, the overall time complexity of the memoized algorithm will also be the same.

Count of Structurally Unique Binary Search Trees (hard)

Given a number ‘n’, write a function to return the count of structurally unique Binary Search Trees (BST) that can store values 1 to ‘n’.

Example 1:

Input: 2
Output: 2
Explanation: As we saw in the previous problem, there are 2 unique BSTs storing numbers from 1-2.

Example 2:

Input: 3
Output: 5
Explanation: There will be 5 unique BSTs that can store numbers from 1 to 3.

Solution

This problem is similar to Structurally Unique Binary Search Trees. Following a similar approach, we can iterate from 1 to ‘n’ and consider each number as the root of a tree and make two recursive calls to count the number of left and right sub-trees.

```
using namespace std;


#include <iostream>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;


  TreeNode(int x) { val = x; }

};
```

```
class CountUniqueTrees {
 public:
  int countTrees(int n) {
    if (n <= 1) return 1;
    int count = 0;
    for (int i = 1; i <= n; i++) {
      // making 'i' root of the tree
      int countOfLeftSubtrees = countTrees(i - 1);
      int countOfRightSubtrees = countTrees(n - i);
      count += (countOfLeftSubtrees * countOfRightSubtrees);
    }
    return count;
  }
};


int main(int argc, char *argv[]) {
  CountUniqueTrees ct;
  int count = ct.countTrees(3);
  cout << "Total trees: " << count;
}
```

Time complexity

The time complexity of this algorithm will be exponential and will be similar to Balanced Parentheses. Estimated time complexity will be $O(n*2^n)$ but the actual time complexity ($O(4^n/\sqrt{n})$) is bounded by the Catalan number and is beyond the scope of a coding interview. See more details here.

Space complexity

The space complexity of this algorithm will be exponential too, estimated O(2^n)$O(2^n)$ but the actual will be (O(4^n/\sqrt{n})$O(4^n/\sqrt{n})$).

Memoized version

Our algorithm has overlapping subproblems as our recursive call will be evaluating the same sub-expression multiple times. To resolve this, we can use memoization and store the intermediate results in a **HashMap**. In each function call, we can check our map to see if we have already evaluated this sub-expression before. Here is the memoized version of our algorithm, please see highlighted changes:

```
using namespace std;


#include <iostream>

#include <unordered_map>


class TreeNode {

 public:

  int val = 0;

  TreeNode *left;

  TreeNode *right;


  TreeNode(int x) { val = x; }

};


class CountUniqueTrees {

 public:

  unordered_map<int, int> map = unordered_map<int, int>();


  virtual int countTrees(int n) {
```

```
    if (map.find(n) != map.end()) {
      return map[n];
    }

    if (n <= 1) {
      return 1;
    }

    int count = 0;
    for (int i = 1; i <= n; i++) {
      // making 'i' root of the tree
      int countOfLeftSubtrees = countTrees(i - 1);
      int countOfRightSubtrees = countTrees(n - i);
      count += (countOfLeftSubtrees * countOfRightSubtrees);
    }
    map[n] = count;
    return count;
  }
};

int main(int argc, char *argv[]) {
  CountUniqueTrees ct;
  int count = ct.countTrees(3);
  cout << "Total trees: " << count;
}
```

The time complexity of the memoized algorithm will be O(n^2)$O(n2)$, since we are iterating from ‘1’ to ‘n’ and ensuring that each sub-problem is evaluated only once. The space complexity will be O(n)$O(n)$ for the memoization map.

1. Introduction

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.

This pattern describes an efficient way to handle all problems involving **Binary Search**. We will go through a set of problems that will help us build an understanding of this pattern so that we can apply this technique to other problems we might come across in the interviews.

Let's start with our first problem.

Problem Statement

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.

Example 1:

Input: [4, 6, 10], key = 10
Output: 2

Example 2:

Input: [1, 2, 3, 4, 5, 6, 7], key = 5
Output: 4

Example 3:

Input: [10, 6, 4], key = 10
Output: 0

Example 4:

Input: [10, 6, 4], key = 4
Output: 2

Solution

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*middle*=(*start*+*end*)/2. For **Java and C++**, this equation will work for most cases, but when `start` or `end` is large, this equation will give us the wrong result due to integer overflow. Imagine that `end` is equal to the maximum range of an integer (e.g. for Java: `int end = Integer.MAX_VALUE`). Now adding any positive number to `end` will result in an integer overflow. Since we need to add both the numbers first to evaluate our equation, an overflow might occur. The safest way to find the middle of two numbers without getting an overflow is as follows:

```
middle  = start + (end-start)/2
```

The above discussion is not relevant for **Python**, as we don't have the integer overflow problem in pure Python.

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`.

5. 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'.

Here is the visual representation of **Binary Search** for the Example-2:

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.

```
using namespace std;


#include <iostream>

#include <vector>


class BinarySearch {

 public:

  static int search(const vector<int>& arr, int key) {

    int start = 0, end = arr.size() - 1;

    bool isAscending = arr[start] < arr[end];

    while (start <= end) {

      // calculate the middle of the current range

      int mid = start + (end - start) / 2;


      if (key == arr[mid]) {
```

```
      return mid;
    }

    if (isAscending) {  // ascending order
      if (key < arr[mid]) {
        end = mid - 1;    // the 'key' can be in the first half
      } else {            // key > arr[mid]
        start = mid + 1;  // the 'key' can be in the second half
      }
    } else {  // descending order
      if (key > arr[mid]) {
        end = mid - 1;    // the 'key' can be in the first half
      } else {            // key < arr[mid]
        start = mid + 1;  // the 'key' can be in the second half
      }
    }
  }
  return -1;  // element not found
 }
};

int main(int argc, char* argv[]) {
 cout << BinarySearch::search(vector<int>{4, 6, 10}, 10) << endl;
 cout << BinarySearch::search(vector<int>{1, 2, 3, 4, 5, 6, 7}, 5) << endl;
 cout << BinarySearch::search(vector<int>{10, 6, 4}, 10) << endl;
 cout << BinarySearch::search(vector<int>{10, 6, 4}, 4) << endl;
}
```

Time complexity

Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(\log N)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Problem Statement

Given an array of numbers sorted in an ascending order, find the ceiling of a given number 'key'. The ceiling of the 'key' will be the smallest element in the given array greater than or equal to the 'key'.

Write a function to return the index of the ceiling of the 'key'. If there isn't any ceiling return -1.

Example 1:

Input: [4, 6, 10], key = 6
Output: 1
Explanation: The smallest number greater than or equal to '6' is '6' having index '1'.

Example 2:

Input: [1, 3, 8, 10, 15], key = 12
Output: 4
Explanation: The smallest number greater than or equal to '12' is '15' having index '4'.

Example 3:

Input: [4, 6, 10], key = 17
Output: -1
Explanation: There is no number greater than or equal to '17' in the given array.

Example 4:

Input: [4, 6, 10], key = -1
Output: 0
Explanation: The smallest number greater than or equal to '-1' is '4' having index '0'.

Solution

This problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the ceiling of a number.

We can use a similar approach as discussed in Order-agnostic Binary Search. We will try to search for the 'key' in the given array. If we find the 'key', we return its index as the ceiling. If we can't find the 'key', the next big number will be pointed out by the index `start`. Consider Example-2 mentioned above:

As key > arr[middle], therefore start = middle + 1 1 3 8 10 15 As key > arr[middle], therefore start = middle + 1 1 3 8 10 15 1 3 8 10 15 As key < arr[middle], therefore end = middle - 1, and the loop will break as end has become less than start Search 'key' = '12'

Since we are always adjusting our range to find the 'key', when we exit the loop, the start of our range will point to the smallest number greater than the 'key' as shown in the above picture.

We can add a check in the beginning to see if the 'key' is bigger than the biggest number in the input array. If so, we can return '-1'.

```
using namespace std;


#include <iostream>

#include <vector>


class CeilingOfANumber {

 public:

  static int searchCeilingOfANumber(const vector<int>& arr, int key) {

    if (key > arr[arr.size() - 1]) {  // if the 'key' is bigger than the biggest element

      return -1;

    }


    int start = 0, end = arr.size() - 1;

    while (start <= end) {
```

```
    int mid = start + (end - start) / 2;
    if (key < arr[mid]) {
      end = mid - 1;
    } else if (key > arr[mid]) {
      start = mid + 1;
    } else {  // found the key
      return mid;
    }
  }
  // since the loop is running until 'start <= end', so at the end of the while loop, 'start ==
  // end+1' we are not able to find the element in the given array, so the next big number will be
  // arr[start]
  return start;
 }
};

int main(int argc, char* argv[]) {
 cout << CeilingOfANumber::searchCeilingOfANumber(vector<int>{4, 6, 10}, 6) << endl;
 cout << CeilingOfANumber::searchCeilingOfANumber(vector<int>{1, 3, 8, 10, 15}, 12) << endl;
 cout << CeilingOfANumber::searchCeilingOfANumber(vector<int>{4, 6, 10}, 17) << endl;
 cout << CeilingOfANumber::searchCeilingOfANumber(vector<int>{4, 6, 10}, -1) << endl;
}
```

Time complexity

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(\log N)$ or $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Similar Problems

Problem 1

Given an array of numbers sorted in ascending order, find the floor of a given number ‘key’. The floor of the ‘key’ will be the biggest element in the given array smaller than or equal to the ‘key’

Write a function to return the index of the floor of the ‘key’. If there isn’t a floor, return -1.

Example 1:

Input: [4, 6, 10], key = 6
Output: 1
Explanation: The biggest number smaller than or equal to '6' is '6' having index '1'.

Example 2:

Input: [1, 3, 8, 10, 15], key = 12
Output: 3
Explanation: The biggest number smaller than or equal to '12' is '10' having index '3'.

Example 3:

Input: [4, 6, 10], key = 17
Output: 2
Explanation: The biggest number smaller than or equal to '17' is '10' having index '2'.

Example 4:

Input: [4, 6, 10], key = -1
Output: -1
Explanation: There is no number smaller than or equal to '-1' in the given array.

```
using namespace std;


#include <iostream>
```

```
#include <vector>

class FloorOfANumber {
 public:
  static int searchFloorOfANumber(const vector<int>& arr, int key) {
    if (key < arr[0]) {  // if the 'key' is smaller than the smallest element
      return -1;
    }

    int start = 0, end = arr.size() - 1;
    while (start <= end) {
      int mid = start + (end - start) / 2;
      if (key < arr[mid]) {
        end = mid - 1;
      } else if (key > arr[mid]) {
        start = mid + 1;
      } else {  // found the key
        return mid;
      }
    }
    // since the loop is running until 'start <= end', so at the end of the while loop, 'start ==
    // end+1' we are not able to find the element in the given array, so the next smaller number
    // will be arr[end]
    return end;
  }
};

int main(int argc, char* argv[]) {
  cout << FloorOfANumber::searchFloorOfANumber(vector<int>{4, 6, 10}, 6) << endl;
```

```
  cout << FloorOfANumber::searchFloorOfANumber(vector<int>{1, 3, 8, 10, 15}, 12) << endl;
  cout << FloorOfANumber::searchFloorOfANumber(vector<int>{4, 6, 10}, 17) << endl;
  cout << FloorOfANumber::searchFloorOfANumber(vector<int>{4, 6, 10}, -1) << endl;
}
```

Problem Statement

Given an array of lowercase letters sorted in ascending order, find the **smallest letter** in the given array **greater than a given 'key'**.

Assume the given array is a **circular list**, which means that the last letter is assumed to be connected with the first letter. This also means that the smallest letter in the given array is greater than the last letter of the array and is also the first letter of the array.

Write a function to return the next letter of the given 'key'.

Example 1:

Input: ['a', 'c', 'f', 'h'], key = 'f'
Output: 'h'
Explanation: The smallest letter greater than 'f' is 'h' in the given array.

Example 2:

Input: ['a', 'c', 'f', 'h'], key = 'b'
Output: 'c'
Explanation: The smallest letter greater than 'b' is 'c'.

Example 3:

Input: ['a', 'c', 'f', 'h'], key = 'm'
Output: 'a'
Explanation: As the array is assumed to be circular, the smallest letter greater than 'm' is 'a'.

Example 4:

Input: ['a', 'c', 'f', 'h'], key = 'h'
Output: 'a'
Explanation: As the array is assumed to be circular, the smallest letter greater than 'h' is 'a'.

Solution

The problem follows the **Binary Search** pattern. Since **Binary Search** helps us find an element in a sorted array efficiently, we can use a modified version of it to find the next letter.

We can use a similar approach as discussed in Ceiling of a Number. There are a couple of differences though:

1. The array is considered circular, which means if the 'key' is bigger than the last letter of the array or if it is smaller than the first letter of the array, the key's next letter will be the first letter of the array.
2. The other difference is that we have to find the next biggest letter which can't be equal to the 'key'. This means that we will ignore the case where `key == arr[middle]`. To handle this case, we can update our start range to `start = middle +1`.

In the end, instead of returning the element pointed out by `start`, we have to return the letter pointed out by `start % array_length`. This is needed because of point 2 discussed above. Imagine that the last letter of the array is equal to the 'key'. In that case, we have to return the first letter of the input array.

```
using namespace std;


#include <iostream>

#include <vector>


class NextLetter {

 public:

  static char searchNextLetter(const vector<char>& letters, char key) {

    int n = letters.size();

    if (key < letters[0] || key > letters[n - 1]) {

      return letters[0];

    }
```

```
    int start = 0, end = n - 1;
    while (start <= end) {
      int mid = start + (end - start) / 2;
      if (key < letters[mid]) {
        end = mid - 1;
      } else {  // if (key >= letters[mid]) {
        start = mid + 1;
      }
    }
    // since the loop is running until 'start <= end', so at the end of the
    // while loop, 'start == end+1'
    return letters[start % n];
  }
};

int main(int argc, char* argv[]) {
  cout << NextLetter::searchNextLetter(vector<char>{'a', 'c', 'f', 'h'}, 'f') << endl;
  cout << NextLetter::searchNextLetter(vector<char>{'a', 'c', 'f', 'h'}, 'b') << endl;
  cout << NextLetter::searchNextLetter(vector<char>{'a', 'c', 'f', 'h'}, 'm') << endl;
  cout << NextLetter::searchNextLetter(vector<char>{'a', 'c', 'f', 'h'}, 'h') << endl;
}
```

Time complexity

Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(\log N)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Problem Statement

Given an array of numbers sorted in ascending order, find the range of a given number 'key'. The range of the 'key' will be the first and last position of the 'key' in the array.

Write a function to return the range of the 'key'. If the 'key' is not present return [-1, -1].

Example 1:

Input: [4, 6, 6, 6, 9], key = 6
Output: [1, 3]

Example 2:

Input: [1, 3, 8, 10, 15], key = 10
Output: [3, 3]

Example 3:

Input: [1, 3, 8, 10, 15], key = 12
Output: [-1, -1]

Solution

The problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the first and the last position of a number.

We can use a similar approach as discussed in Order-agnostic Binary Search. We will try to search for the 'key' in the given array; if the 'key' is found (i.e. `key == arr[middle]`) we have two options:

1. When trying to find the first position of the 'key', we can update `end = middle - 1` to see if the key is present before `middle`.
2. When trying to find the last position of the 'key', we can update `start = middle + 1` to see if the key is present after `middle`.

In both cases, we will keep track of the last position where we found the 'key'. These positions will be the required range.

```
using namespace std;

#include <iostream>
#include <vector>

class FindRange {
 public:
  static pair<int, int> findRange(const vector<int> &arr, int key) {
    pair<int, int> result(-1, -1);
    result.first = search(arr, key, false);
    if (result.first != -1) {  // no need to search, if 'key' is not present in the input array
      result.second = search(arr, key, true);
    }
    return result;
  }

 private:
  // modified Binary Search
  static int search(const vector<int> &arr, int key, bool findMaxIndex) {
    int keyIndex = -1;
    int start = 0, end = arr.size() - 1;
    while (start <= end) {
      int mid = start + (end - start) / 2;
      if (key < arr[mid]) {
        end = mid - 1;
      } else if (key > arr[mid]) {
        start = mid + 1;
      } else {  // key == arr[mid]
```

```
      keyIndex = mid;
      if (findMaxIndex) {
        start = mid + 1;  // search ahead to find the last index of 'key'
      } else {
        end = mid - 1;  // search behind to find the first index of 'key'
      }
    }
  }
  return keyIndex;
 }
};

int main(int argc, char *argv[]) {
 pair<int, int> result = FindRange::findRange(vector<int>{4, 6, 6, 6, 9}, 6);
 cout << "Range: [" << result.first << ", " << result.second << "]" << endl;
 result = FindRange::findRange(vector<int>{1, 3, 8, 10, 15}, 10);
 cout << "Range: [" << result.first << ", " << result.second << "]" << endl;
 result = FindRange::findRange(vector<int>{1, 3, 8, 10, 15}, 12);
 cout << "Range: [" << result.first << ", " << result.second << "]" << endl;
}
```

Time complexity

Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(\log N)$ or $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Problem Statement

Given an infinite sorted array (or an array with unknown size), find if a given number 'key' is present in the array. Write a function to return the index of the 'key' if it is present in the array, otherwise return -1.

Since it is not possible to define an array with infinite (unknown) size, you will be provided with an interface `ArrayReader` to read elements of the array. ArrayReader.get(index) will return the number at index; if the array's size is smaller than the index, it will return `Integer.MAX_VALUE`.

Example 1:

Input: [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30], key = 16
Output: 6
Explanation: The key is present at index '6' in the array.

Example 2:

Input: [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30], key = 11
Output: -1
Explanation: The key is not present in the array.

Example 3:

Input: [1, 3, 8, 10, 15], key = 15
Output: 4
Explanation: The key is present at index '4' in the array.

Example 4:

Input: [1, 3, 8, 10, 15], key = 200
Output: -1
Explanation: The key is not present in the array.

Solution

The problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the 'key' in an infinite sorted array.

The only issue with applying binary search in this problem is that we don't know the bounds of the array. To handle this situation, we will first find the proper bounds of the array where we can perform a binary search.

An efficient way to find the proper bounds is to start at the beginning of the array with the bound's size as '1' and exponentially increase the bound's size (i.e., double it) until we find the bounds that can have the key.

Consider Example-1 mentioned above:

Once we have searchable bounds we can apply the binary search.

```
using namespace std;


#include <iostream>
#include <limits>
#include <vector>


class ArrayReader {
 public:
  vector<int> arr;


  ArrayReader(const vector<int> &arr) { this->arr = arr; }


  virtual int get(int index) {
    if (index >= arr.size()) {
      return numeric_limits<int>::max();
    }
    return arr[index];
  }
};
```

```
class SearchInfiniteSortedArray {
 public:
  static int search(ArrayReader *reader, int key) {
    // find the proper bounds first
    int start = 0, end = 1;
    while (reader->get(end) < key) {
      int newStart = end + 1;
      end += (end - start + 1) * 2;  // increase to double the bounds size
      start = newStart;
    }
    return binarySearch(reader, key, start, end);
  }

 private:
  static int binarySearch(ArrayReader *reader, int key, int start, int end) {
    while (start <= end) {
      int mid = start + (end - start) / 2;
      if (key < reader->get(mid)) {
        end = mid - 1;
      } else if (key > reader->get(mid)) {
        start = mid + 1;
      } else {  // found the key
        return mid;
      }
    }

    return -1;
  }
};
```

```
int main(int argc, char *argv[]) {
  ArrayReader *reader =
      new ArrayReader(vector<int>{4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30});
  cout << SearchInfiniteSortedArray::search(reader, 16) << endl;
  cout << SearchInfiniteSortedArray::search(reader, 11) << endl;
  reader = new ArrayReader(vector<int>{1, 3, 8, 10, 15});
  cout << SearchInfiniteSortedArray::search(reader, 15) << endl;
  cout << SearchInfiniteSortedArray::search(reader, 200) << endl;
  delete reader;
}
```

Time complexity

There are two parts of the algorithm. In the first part, we keep increasing the bound's size exponentially (double it every time) while searching for the proper bounds. Therefore, this step will take $O(\log N)$ assuming that the array will have maximum 'N' numbers. In the second step, we perform the binary search which will take $O(\log N)$, so the overall time complexity of our algorithm will be $O(\log N + \log N)$ which is asymptotically equivalent to $O(\log N)$.

Space complexity

The algorithm runs in constant space $O(1)$.

Problem Statement

Given an array of numbers sorted in ascending order, find the element in the array that has the minimum difference with the given 'key'.

Example 1:

Input: [4, 6, 10], key = 7
Output: 6
Explanation: The difference between the key '7' and '6' is minimum than any other number in the array

Example 2:

Input: [4, 6, 10], key = 4
Output: 4

Example 3:

Input: [1, 3, 8, 10, 15], key = 12
Output: 10

Example 4:

Input: [4, 6, 10], key = 17
Output: 10

Solution

The problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the number that has the minimum difference with the given 'key'.

We can use a similar approach as discussed in Order-agnostic Binary Search. We will try to search for the 'key' in the given array. If we find the 'key' we will return it as the minimum difference number. If we can't find the 'key', (at the end of the loop) we can find the differences between the 'key' and the numbers pointed out by indices `start` and `end`, as these two numbers will be closest to the 'key'. The number that gives minimum difference will be our required number.

```
using namespace std;


#include <iostream>

#include <vector>


class MinimumDifference {
```

```
public:
 static int searchMinDiffElement(const vector<int>& arr, int key) {
  if (key < arr[0]) {
   return arr[0];
  }
  if (key > arr[arr.size() - 1]) {
   return arr[arr.size() - 1];
  }

  int start = 0, end = arr.size() - 1;
  while (start <= end) {
   int mid = start + (end - start) / 2;
   if (key < arr[mid]) {
    end = mid - 1;
   } else if (key > arr[mid]) {
    start = mid + 1;
   } else {
    return arr[mid];
   }
  }

  // at the end of the while loop, 'start == end+1'
  // we are not able to find the element in the given array
  // return the element which is closest to the 'key'
  if ((arr[start] - key) < (key - arr[end])) {
   return arr[start];
  }
  return arr[end];
 }
```

```
};


int main(int argc, char* argv[]) {
  cout << MinimumDifference::searchMinDiffElement(vector<int>{4, 6, 10}, 7) << endl;
  cout << MinimumDifference::searchMinDiffElement(vector<int>{4, 6, 10}, 4) << endl;
  cout << MinimumDifference::searchMinDiffElement(vector<int>{1, 3, 8, 10, 15}, 12) << endl;
  cout << MinimumDifference::searchMinDiffElement(vector<int>{4, 6, 10}, 17) << endl;
}
```

Time complexity

Since, we are reducing the search range by half at every step, this means the time complexity of our algorithm will be $O(\log N)$ or $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$ or $O(1)$.

Problem Statement

Find the maximum value in a given Bitonic array. An array is considered bitonic if it is monotonically increasing and then monotonically decreasing. Monotonically increasing or decreasing means that for any index `i` in the array `arr[i] != arr[i+1]`.

Example 1:

Input: [1, 3, 8, 12, 4, 2]
Output: 12
Explanation: The maximum number in the input bitonic array is '12'.

Example 2:

Input: [3, 8, 3, 1]
Output: 8

Example 3:

Input: [1, 3, 8, 12]
Output: 12

Example 4:

Input: [10, 9, 8]
Output: 10

Solution

A bitonic array is a sorted array; the only difference is that its first part is sorted in ascending order and the second part is sorted in descending order. We can use a similar approach as discussed in Order-agnostic Binary Search. Since no two consecutive numbers are same (as the array is monotonically increasing or decreasing), whenever we calculate the `middle`, we can compare the numbers pointed out by the index `middle` and `middle+1` to find if we are in the ascending or the descending part. So:

1. If `arr[middle] > arr[middle + 1]`, we are in the second (descending) part of the bitonic array. Therefore, our required number could either be pointed out by `middle` or will be before `middle`. This means we will be doing: `end = middle`.
2. If `arr[middle] < arr[middle + 1]`, we are in the first (ascending) part of the bitonic array. Therefore, the required number will be after `middle`. This means we will be doing: `start = middle + 1`.

We can break when `start == end`. Due to the two points mentioned above, both `start` and `end` will be pointing at the maximum number of the bitonic array.

```
using namespace std;


#include <iostream>

#include <vector>


class MaxInBitonicArray {

 public:
```

```
  static int findMax(const vector<int>& arr) {
    int start = 0, end = arr.size() - 1;
    while (start < end) {
      int mid = start + (end - start) / 2;
      if (arr[mid] > arr[mid + 1]) {
        end = mid;
      } else {
        start = mid + 1;
      }
    }

    // at the end of the while loop, 'start == end'
    return arr[start];
  }
};

int main(int argc, char* argv[]) {
  cout << MaxInBitonicArray::findMax(vector<int>{1, 3, 8, 12, 4, 2}) << endl;
  cout << MaxInBitonicArray::findMax(vector<int>{3, 8, 3, 1}) << endl;
  cout << MaxInBitonicArray::findMax(vector<int>{1, 3, 8, 12}) << endl;
  cout << MaxInBitonicArray::findMax(vector<int>{10, 9, 8}) << endl;
}
```

Time complexity

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(\log N)$ or $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Search Bitonic Array (medium)

Given a Bitonic array, find if a given 'key' is present in it. An array is considered bitonic if it is monotonically increasing and then monotonically decreasing. Monotonically increasing or decreasing means that for any index `i` in the array `arr[i] != arr[i+1]`.

Write a function to return the index of the 'key'. If the 'key' is not present, return -1.

Example 1:

Input: [1, 3, 8, 4, 3], key=4
Output: 3

Example 2:

Input: [3, 8, 3, 1], key=8
Output: 1

Example 3:

Input: [1, 3, 8, 12], key=12
Output: 3

Example 4:

Input: [10, 9, 8], key=10
Output: 0

Solution

The problem follows the **Binary Search** pattern. Since Binary Search helps us efficiently find a number in a sorted array we can use a modified version of the Binary Search to find the 'key' in the bitonic array.

Here is how we can search in a bitonic array:

1. First, we can find the index of the maximum value of the bitonic array, similar to Bitonic Array Maximum. Let's call the index of the maximum number `maxIndex`.
2. Now, we can break the array into two sub-arrays:

- Array from index '0' to `maxIndex`, sorted in ascending order.
- Array from index `maxIndex+1` to `array_length-1`, sorted in descending order.

3. We can then call **Binary Search** separately in these two arrays to search the 'key'. We can use the same Order-agnostic Binary Search for searching.

```
using namespace std;


#include <iostream>

#include <vector>


class SearchBitonicArray {

 public:

  static int search(const vector<int> &arr, int key) {

    int maxIndex = findMax(arr);

    int keyIndex = binarySearch(arr, key, 0, maxIndex);

    if (keyIndex != -1) {

      return keyIndex;

    }

    return binarySearch(arr, key, maxIndex + 1, arr.size() - 1);

  }


  // find index of the maximum value in a bitonic array

  static int findMax(const vector<int> &arr) {

    int start = 0, end = arr.size() - 1;

    while (start < end) {

      int mid = start + (end - start) / 2;

      if (arr[mid] > arr[mid + 1]) {

        end = mid;
```

```
    } else {
      start = mid + 1;
    }
  }

  // at the end of the while loop, 'start == end'
  return start;
 }

private:
 // order-agnostic binary search
 static int binarySearch(const vector<int> &arr, int key, int start, int end) {
  while (start <= end) {
    int mid = start + (end - start) / 2;

    if (key == arr[mid]) {
      return mid;
    }

    if (arr[start] < arr[end]) {  // ascending order
      if (key < arr[mid]) {
        end = mid - 1;
      } else {  // key > arr[mid]
        start = mid + 1;
      }
    } else {  // descending order
      if (key > arr[mid]) {
        end = mid - 1;
      } else {  // key < arr[mid]
```

```
        start = mid + 1;
      }
    }
  }
  return -1;  // element is not found
 }
};

int main(int argc, char *argv[]) {
 cout << SearchBitonicArray::search(vector<int>{1, 3, 8, 4, 3}, 4) << endl;
 cout << SearchBitonicArray::search(vector<int>{3, 8, 3, 1}, 8) << endl;
 cout << SearchBitonicArray::search(vector<int>{1, 3, 8, 12}, 12) << endl;
 cout << SearchBitonicArray::search(vector<int>{10, 9, 8}, 10) << endl;
}
```

Time complexity

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(\log N)$ or $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Search in Rotated Array (medium)

Given an array of numbers which is sorted in ascending order and also rotated by some arbitrary number, find if a given 'key' is present in it.

Write a function to return the index of the 'key' in the rotated array. If the 'key' is not present, return -1. You can assume that the given array does not have any duplicates.

Example 1:

Input: [10, 15, 1, 3, 8], key = 15
Output: 1
Explanation: '15' is present in the array at index '1'.

Example 2:

Input: [4, 5, 7, 9, 10, -1, 2], key = 10
Output: 4
Explanation: '10' is present in the array at index '4'.

Solution

The problem follows the **Binary Search** pattern. We can use a similar approach as discussed in Order-agnostic Binary Search and modify it similar to Search Bitonic Array to search for the 'key' in the rotated array.

After calculating the `middle`, we can compare the numbers at indices `start` and `middle`. This will give us two options:

1. If `arr[start] <= arr[middle]`, the numbers from `start` to `middle` are sorted in ascending order.
2. Else, the numbers from `middle+1` to `end` are sorted in ascending order.

Once we know which part of the array is sorted, it is easy to adjust our ranges. For example, if option-1 is true, we have two choices:

1. By comparing the 'key' with the numbers at index `start` and `middle` we can easily find out if the 'key' lies between indices `start` and `middle`; if it does, we can skip the second part => `end = middle -1`.
2. Else, we can skip the first part => `start = middle + 1`.

Let's visually see this with the above-mentioned Example-2:

Since there are no duplicates in the given array, it is always easy to skip one part of the array in each iteration. However, if there are duplicates, it is not always possible to know which part is sorted. We will look into this case in the 'Similar Problems' section.

```
using namespace std;
```

```
#include <iostream>
#include <vector>

class SearchRotatedArray {
 public:
  static int search(const vector<int>& arr, int key) {
    int start = 0, end = arr.size() - 1;
    while (start <= end) {
      int mid = start + (end - start) / 2;
      if (arr[mid] == key) {
        return mid;
      }

      if (arr[start] <= arr[mid]) {  // left side is sorted in ascending order
        if (key >= arr[start] && key < arr[mid]) {
          end = mid - 1;
        } else {  // key > arr[mid]
          start = mid + 1;
        }
      } else {  // right side is sorted in ascending order
        if (key > arr[mid] && key <= arr[end]) {
          start = mid + 1;
        } else {
          end = mid - 1;
        }
      }
    }
```

```
    // we are not able to find the element in the given array
    return -1;
  }
};

int main(int argc, char* argv[]) {
  cout << SearchRotatedArray::search(vector<int>{10, 15, 1, 3, 8}, 15) << endl;
  cout << SearchRotatedArray::search(vector<int>{4, 5, 7, 9, 10, -1, 2}, 10) << endl;
}
```

Time complexity

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(\log N)$ or $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Similar Problems

Problem 1

How do we search in a sorted and rotated array that also has duplicates?

The code above will fail in the following example!

Example 1:

```
Input: [3, 7, 3, 3, 3], key = 7
Output: 1
Explanation: '7' is present in the array at index '1'.
```

Solution

The only problematic scenario is when the numbers at indices `start`, `middle`, and `end` are the same, as in this case, we can't decide which part of the array is sorted. In such a case, the best we can do is to skip one number from both ends: `start = start + 1` & `end = end - 1`.

```
using namespace std;


#include <iostream>

#include <vector>


class SearchRotatedWithDuplicate {

 public:

  static int search(const vector<int>& arr, int key) {

    int start = 0, end = arr.size() - 1;

    while (start <= end) {

      int mid = start + (end - start) / 2;

      if (arr[mid] == key) {

        return mid;

      }


      // the only difference from the previous solution,

      // if numbers at indexes start, mid, and end are same, we can't choose a side

      // the best we can do, is to skip one number from both ends as key != arr[mid]

      if ((arr[start] == arr[mid]) && (arr[end] == arr[mid])) {

        ++start;

        --end;

      } else if (arr[start] <= arr[mid]) {  // left side is sorted in ascending order

        if (key >= arr[start] && key < arr[mid]) {

          end = mid - 1;
```

```
      } else {  // key > arr[mid]
        start = mid + 1;
      }
    } else {  // right side is sorted in ascending order
      if (key > arr[mid] && key <= arr[end]) {
        start = mid + 1;
      } else {
        end = mid - 1;
      }
    }
  }

  // we are not able to find the element in the given array
  return -1;
 }
};

int main(int argc, char* argv[]) {
 cout << SearchRotatedWithDuplicate::search(vector<int>{3, 7, 3, 3, 3}, 7) << endl;
}
```

Time complexity

This algorithm will run most of the times in $O(\log N)$. However, since we only skip two numbers in case of duplicates instead of half of the numbers, the worst case time complexity will become $O(N)$.

Space complexity

The algorithm runs in constant space $O(1)$.

Rotation Count (medium)

Given an array of numbers which is sorted in ascending order and is rotated ‘k’ times around a pivot, find ‘k’.

You can assume that the array does not have any duplicates.

Example 1:

Input: [10, 15, 1, 3, 8]
Output: 2
Explanation: The array has been rotated 2 times.

Example 2:

Input: [4, 5, 7, 9, 10, -1, 2]
Output: 5
Explanation: The array has been rotated 5 times.

Example 3:

Input: [1, 3, 8, 10]
Output: 0
Explanation: The array has been not been rotated.

Solution

This problem follows the **Binary Search** pattern. We can use a similar strategy as discussed in Search in Rotated Array.

In this problem, actually, we are asked to find the index of the minimum element. The number of times the minimum element is moved to the right will be equal to the number of rotations. An interesting fact about the minimum element is that it is the only element in the given array which is smaller than its previous element. Since the array is sorted in ascending order, all other elements are bigger than their previous element.

After calculating the `middle`, we can compare the number at index `middle` with its previous and next number. This will give us two options:

1. If `arr[middle] > arr[middle + 1]`, then the element at `middle + 1` is the smallest.

2. If `arr[middle - 1] > arr[middle]`, then the element at `middle` is the smallest.

To adjust the ranges we can follow the same approach as discussed in Search in Rotated Array. Comparing the numbers at indices `start` and `middle` will give us two options:

1. If `arr[start] < arr[middle]`, the numbers from `start` to `middle` are sorted.
2. Else, the numbers from `middle + 1` to `end` are sorted.

```
using namespace std;


#include <iostream>

#include <vector>


class RotationCountOfRotatedArray {

 public:

  static int countRotations(const vector<int>& arr) {

    int start = 0, end = arr.size() - 1;

    while (start < end) {

      int mid = start + (end - start) / 2;


      if (mid < end && arr[mid] > arr[mid + 1]) {  // if mid is greater than the next element

        return mid + 1;

      }

      if (mid > start && arr[mid - 1] > arr[mid]) {  // if mid is smaller than the previous element

        return mid;

      }


      if (arr[start] < arr[mid]) {  // left side is sorted, so the pivot is on right side

        start = mid + 1;

      } else {  // right side is sorted, so the pivot is on the left side
```

```
      end = mid - 1;
    }
   }

   return 0;  // the array has not been rotated
  }
};

int main(int argc, char* argv[]) {
  cout << RotationCountOfRotatedArray::countRotations(vector<int>{10, 15, 1, 3, 8}) << endl;
  cout << RotationCountOfRotatedArray::countRotations(vector<int>{4, 5, 7, 9, 10, -1, 2}) << endl;
  cout << RotationCountOfRotatedArray::countRotations(vector<int>{1, 3, 8, 10}) << endl;
}
```

Time complexity

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(\log N)$ or $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Similar Problems

Problem 1

How do we find the rotation count of a sorted and rotated array that has duplicates too?

The above code will fail on the following example!

Example 1:

Input: [3, 3, 7, 3]
Output: 3
Explanation: The array has been rotated 3 times

Solution

We can follow the same approach as discussed in Search in Rotated Array. The only difference is that before incrementing `start` or decrementing `end`, we will check if either of them is the smallest number.

```
using namespace std;


#include <iostream>

#include <vector>


class RotationCountWithDuplicates {

 public:

  static int countRotations(const vector<int>& arr) {

    int start = 0, end = arr.size() - 1;

    while (start < end) {

      int mid = start + (end - start) / 2;

      if (mid < end &&

          arr[mid] > arr[mid + 1]) {  // if element at mid is greater than the next element

        return mid + 1;

      }

      if (mid > start &&

          arr[mid - 1] > arr[mid]) {  // if element at mid is smaller than the previous element

        return mid;

      }


      // this is the only difference from the previous solution

      // if numbers at indices start, mid, and end are same, we can't choose a side
```

```
    // the best we can do is to skip one number from both ends if they are not the smallest number
    if (arr[start] == arr[mid] && arr[end] == arr[mid]) {
      if (arr[start] > arr[start + 1]) {  // if element at start+1 is not the smallest
        return start + 1;
      }
      ++start;
      if (arr[end - 1] > arr[end]) {  // if the element at end is not the smallest
        return end;
      }
      --end;
    // left side is sorted, so the pivot is on right side
    } else if (arr[start] < arr[mid] || (arr[start] == arr[mid] && arr[mid] > arr[end])) {
      start = mid + 1;
    } else {  // right side is sorted, so the pivot is on the left side
      end = mid - 1;
    }
  }

  return 0;  // the array has not been rotated
 }
};

int main(int argc, char* argv[]) {
 cout << RotationCountWithDuplicates::countRotations(vector<int>{3, 3, 7, 3}) << endl;
}
```

Time complexity

This algorithm will run in $O(\log N)$ most of the times, but since we only skip two numbers in case of duplicates instead of the half of the numbers, therefore the worst case time complexity will become $O(N)$.

Space complexity #

The algorithm runs in constant space $O(1)$.

1. Introduction

Any problem that asks us to find the top/smallest/frequent 'K' elements among a given set falls under this pattern.

The best data structure that comes to mind to keep track of 'K' elements is **Heap**. This pattern will make use of the Heap to solve multiple problems dealing with 'K' elements at a time from a set of given elements.

Let's jump onto our first problem to develop an understanding of this pattern.

Problem Statement

Given an unsorted array of numbers, find the 'K' largest numbers in it.

Note: For a detailed discussion about different approaches to solve this problem, take a look at Kth Smallest Number.

Example 1:

Input: [3, 1, 5, 12, 2, 11], K = 3
Output: [5, 12, 11]

Example 2:

Input: [5, 12, 11, -1, 12], K = 3
Output: [12, 11, 12]

Solution

A brute force solution could be to sort the array and return the largest K numbers. The time complexity of such an algorithm will

be $O(N*logN)$ as we need to use a sorting algorithm like Timsort if we use Java's `Collection.sort()`. Can we do better than that?

The best data structure that comes to mind to keep track of top 'K' elements is Heap. Let's see if we can use a heap to find a better algorithm.

If we iterate through the array one element at a time and keep 'K' largest numbers in a heap such that each time we find a larger number than the smallest number in the heap, we do two things:

1. Take out the smallest number from the heap, and
2. Insert the larger number into the heap.

This will ensure that we always have 'K' largest numbers in the heap. The most efficient way to repeatedly find the smallest number among a set of numbers will be to use a min-heap. As we know, we can find the smallest number in a min-heap in constant time $O(1)$, since the smallest number is always at the root of the heap. Extracting the smallest number from a min-heap will take $O(logN)$ (if the heap has 'N' elements) as the heap needs to readjust after the removal of an element.

Let's take Example-1 to go through each step of our algorithm:

Given array: [3, 1, 5, 12, 2, 11], and K=3

1. First, let's insert 'K' elements in the min-heap.
2. After the insertion, the heap will have three numbers [3, 1, 5] with '1' being the root as it is the smallest element.
3. We'll iterate through the remaining numbers and perform the above-mentioned two steps if we find a number larger than the root of the heap.
4. The 4th number is '12' which is larger than the root (which is '1'), so let's take out '1' and insert '12'. Now the heap will have [3, 5, 12] with '3' being the root as it is the smallest element.
5. The 5th number is '2' which is not bigger than the root of the heap ('3'), so we can skip this as we already have top three numbers in the heap.
6. The last number is '11' which is bigger than the root (which is '3'), so let's take out '3' and insert '11'. Finally, the heap has the largest three numbers: [5, 12, 11]

As discussed above, it will take us $O(\log K)$ to extract the minimum number from the min-heap. So the overall time complexity of our algorithm will be $O(K*\log K+(N-K)*\log K)$ since, first, we insert 'K' numbers in the heap and then iterate through the remaining numbers and at every step, in the worst case, we need to extract the minimum number and insert a new number in the heap. This algorithm is better than $O(N*\log N)$.

Here is the visual representation of our algorithm:

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <queue>

#include <vector>


class KLargestNumbers {

 public:

  struct greater {

    bool operator()(const int& a, const int& b) const { return a > b; }

  };

  static vector<int> findKLargestNumbers(const vector<int>& nums, int k) {

    // put first 'K' numbers in the min heap

    vector<int> minHeap(nums.begin(), nums.begin() + k);

    make_heap(minHeap.begin(), minHeap.end(), greater());


    // go through the remaining numbers of the array, if the number from the array is bigger than

    // the top (smallest) number of the min-heap, remove the top number from heap and add the number

    // from array

    for (int i = k; i < nums.size(); i++) {

      if (nums[i] > minHeap.front()) {
```

```
      pop_heap(minHeap.begin(), minHeap.end(), greater());
      minHeap.pop_back();
      minHeap.push_back(nums[i]);
      push_heap(minHeap.begin(), minHeap.end(), greater());
    }
   }

   // the heap has the top 'K' numbers
   return minHeap;
  }
};

int main(int argc, char* argv[]) {
  vector<int> result = KLargestNumbers::findKLargestNumbers(vector<int>{3, 1, 5, 12, 2, 11}, 3);
  cout << "Here are the top K numbers: ";
  for (auto num : result) {
   cout << num << " ";
  }
  cout << endl;

  result = KLargestNumbers::findKLargestNumbers(vector<int>{5, 12, 11, -1, 12}, 3);
  cout << "Here are the top K numbers: ";
  for (auto num : result) {
   cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

As discussed above, the time complexity of this algorithm is $O(K*\log K+(N-K)*\log K)$ $O(K*logK+(N-K)*logK)$, which is asymptotically equal to $O(N*\log K)$ $O(N*logK)$

Space complexity

The space complexity will be $O(K)$ $O(K)$ since we need to store the top 'K' numbers in the heap.

Problem Statement

Given an unsorted array of numbers, find Kth smallest number in it.

Please note that it is the Kth smallest number in the sorted order, not the Kth distinct element.

Note: For a detailed discussion about different approaches to solve this problem, take a look at Kth Smallest Number.

Example 1:

Input: [1, 5, 12, 2, 11, 5], K = 3
Output: 5
Explanation: The 3rd smallest number is '5', as the first two smaller numbers are [1, 2].

Example 2:

Input: [1, 5, 12, 2, 11, 5], K = 4
Output: 5
Explanation: The 4th smallest number is '5', as the first three small numbers are [1, 2, 5].

Example 3:

Input: [5, 12, 11, -1, 12], K = 3
Output: 11
Explanation: The 3rd smallest number is '11', as the first two small numbers are [5, -1].

Solution

This problem follows the Top 'K' Numbers pattern but has two differences:

1. Here we need to find the Kth `smallest` number, whereas in Top 'K' Numbers we were dealing with 'K' `largest` numbers.
2. In this problem, we need to find only one number (Kth smallest) compared to finding all 'K' largest numbers.

We can follow the same approach as discussed in the 'Top K Elements' problem. To handle the first difference mentioned above, we can use a max-heap instead of a min-heap. As we know, the root is the biggest element in the max heap. So, since we want to keep track of the 'K' smallest numbers, we can compare every number with the root while iterating through all numbers, and if it is smaller than the root, we'll take the root out and insert the smaller number.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class KthSmallestNumber {

 public:

  static int findKthSmallestNumber(const vector<int> &nums, int k) {

    priority_queue<int> maxHeap;


    // put first k numbers in the max heap

    for (int i = 0; i < k; i++) {

      maxHeap.push(nums[i]);

    }


    // go through the remaining numbers of the array, if the number from the array is smaller than

    // the top (biggest) number of the heap, remove the top number from heap and add the number from

    // array
```

```
    for (int i = k; i < nums.size(); i++) {
      if (nums[i] < maxHeap.top()) {
        maxHeap.pop();
        maxHeap.push(nums[i]);
      }
    }

    // the root of the heap has the Kth smallest number
    return maxHeap.top();
  }
};

int main(int argc, char *argv[]) {
  int result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 3);
  cout << "Kth smallest number is: " << result << endl;

  // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
  result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 4);
  cout << "Kth smallest number is: " << result << endl;

  result = KthSmallestNumber::findKthSmallestNumber(vector<int>{5, 12, 11, -1, 12}, 3);
  cout << "Kth smallest number is: " << result << endl;
}
```

Time complexity

The time complexity of this algorithm is O(K*logK+(N-K)*logK)$O(K*logK+(N-K)*logK)$, which is asymptotically equal to O(N*logK)$O(N*logK)$

Space complexity

The space complexity will be $O(K)$ because we need to store 'K' smallest numbers in the heap.

An Alternate Approach

Alternatively, we can use a **Min Heap** to find the Kth smallest number. We can insert all the numbers in the min-heap and then extract the top 'K' numbers from the heap to find the Kth smallest number. Initializing the min-heap with all numbers will take $O(N)$ and extracting 'K' numbers will take $O(KlogN)$. Overall, the time complexity of this algorithm will be $O(N+KlogN)$ and the space complexity will be $O(N)$.

Problem Statement

Given an array of points in a $2D$ plane, find 'K' closest points to the origin.

Example 1:

Input: points = [[1,2],[1,3]], K = 1
Output: [[1,2]]
Explanation: The Euclidean distance between (1, 2) and the origin is sqrt(5).
The Euclidean distance between (1, 3) and the origin is sqrt(10).
Since sqrt(5) < sqrt(10), therefore (1, 2) is closer to the origin.

Example 2:

Input: point = [[1, 3], [3, 4], [2, -1]], K = 2
Output: [[1, 3], [2, -1]]

Solution

The Euclidean distance of a point P(x,y) from the origin can be calculated through the following formula:

$\sqrt{x^2 + y^2}\sqrt{x2+y2}$

This problem follows the Top 'K' Numbers pattern. The only difference in this problem is that we need to find the closest point (to the origin) as compared to finding the largest numbers.

Following a similar approach, we can use a **Max Heap** to find 'K' points closest to the origin. While iterating through all points, if a point (say 'P') is closer to the origin than the top point of the max-heap, we will remove that top point from the heap and add 'P' to always keep the closest points in the heap.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class Point {

 public:

  int x = 0;

  int y = 0;


  Point(int x, int y) {

    this->x = x;

    this->y = y;

  }


  int distFromOrigin() const {

    // ignoring sqrt

    return (x * x) + (y * y);

  }


  const bool operator<(const Point& p) { return p.distFromOrigin() > this->distFromOrigin(); }
```

```
};

class KClosestPointsToOrigin {
 public:
  static vector<Point> findClosestPoints(const vector<Point>& points, int k) {
    // put first 'k' points in the vector
    vector<Point> maxHeap(points.begin(), points.begin() + k);
    make_heap(maxHeap.begin(), maxHeap.end());

    // go through the remaining points of the input array, if a point is closer to the origin than
    // the top point of the max-heap, remove the top point from heap and add the point from the
    // input array
    for (int i = k; i < points.size(); i++) {
      if (points[i].distFromOrigin() < maxHeap.front().distFromOrigin()) {
        pop_heap(maxHeap.begin(), maxHeap.end());
        maxHeap.pop_back();
        maxHeap.push_back(points[i]);
        push_heap(maxHeap.begin(), maxHeap.end());
      }
    }

    // the heap has 'k' points closest to the origin
    return maxHeap;
  }
};

int main(int argc, char* argv[]) {
  vector<Point> maxHeap = KClosestPointsToOrigin::findClosestPoints({{1, 3}, {3, 4}, {2, -1}}, 2);
  cout << "Here are the k points closest the origin: ";
```

```
 for (auto p : maxHeap) {
   cout << "[" << p.x << " , " << p.y << "] ";
 }
}
```

Time complexity

The time complexity of this algorithm is $(N*\log K)$ $(N*logK)$ as we iterating all points and pushing them into the heap.

Space complexity

The space complexity will be O(K) because we need to store 'K' point in the heap.

Problem Statement

Given 'N' ropes with different lengths, we need to connect these ropes into one big rope with minimum cost. The cost of connecting two ropes is equal to the sum of their lengths.

Example 1:

Input: [1, 3, 11, 5]
Output: 33
Explanation: First connect 1+3(=4), then 4+5(=9), and then 9+11(=20). So the total cost is 33 (4+9+20)

Example 2:

Input: [3, 4, 5, 6]
Output: 36
Explanation: First connect 3+4(=7), then 5+6(=11), 7+11(=18). Total cost is 36 (7+11+18)

Example 3:

Input: [1, 3, 11, 5, 2]
Output: 42
Explanation: First connect 1+2(=3), then 3+3(=6), 6+5(=11), 11+11(=22). Total cost is 42 (3+6+11+22)

Solution

In this problem, following a greedy approach to connect the smallest ropes first will ensure the lowest cost. We can use a **Min Heap** to find the smallest ropes following a similar approach as discussed in Kth Smallest Number. Once we connect two ropes, we need to insert the resultant rope back in the **Min Heap** so that we can connect it with the remaining ropes.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class ConnectRopes {

 public:

  static int minimumCostToConnectRopes(const vector<int> &ropeLengths) {

    priority_queue<int, vector<int>, greater<int>> minHeap;

    // add all ropes to the min heap

    for (int i = 0; i < ropeLengths.size(); i++) {

      minHeap.push(ropeLengths[i]);

    }


    // go through the values of the heap, in each step take top (lowest) rope lengths from the min

    // heap connect them and push the result back to the min heap. keep doing this until the heap is

    // left with only one rope

    int result = 0, temp = 0;

    while (minHeap.size() > 1) {

      temp = minHeap.top();

      minHeap.pop();

      temp += minHeap.top();
```

```
    minHeap.pop();
    result += temp;
    minHeap.push(temp);
   }

   return result;
  }
};

int main(int argc, char *argv[]) {
  int result = ConnectRopes::minimumCostToConnectRopes(vector<int>{1, 3, 11, 5});
  cout << "Minimum cost to connect ropes: " << result << endl;
  result = ConnectRopes::minimumCostToConnectRopes(vector<int>{3, 4, 5, 6});
  cout << "Minimum cost to connect ropes: " << result << endl;
  result = ConnectRopes::minimumCostToConnectRopes(vector<int>{1, 3, 11, 5, 2});
  cout << "Minimum cost to connect ropes: " << result << endl;
}
```

Time complexity

Given ‘N’ ropes, we need $O(N*logN)$ to insert all the ropes in the heap. In each step, while processing the heap, we take out two elements from the heap and insert one. This means we will have a total of ‘N’ steps, having a total time complexity of $O(N*logN)$.

Space complexity

The space complexity will be $O(N)$ because we need to store all the ropes in the heap.

Problem Statement

Given an unsorted array of numbers, find the top ‘K’ frequently occurring numbers in it.

Example 1:

Input: [1, 3, 5, 12, 11, 12, 11], K = 2
Output: [12, 11]
Explanation: Both '11' and '12' appeared twice.

Example 2:

Input: [5, 12, 11, 3, 11], K = 2
Output: [11, 5] or [11, 12] or [11, 3]
Explanation: Only '11' appeared twice, all other numbers appeared once.

Solution

This problem follows Top ‘K’ Numbers. The only difference is that in this problem, we need to find the most frequently occurring number compared to finding the largest numbers.

We can follow the same approach as discussed in the **Top K Elements** problem. However, in this problem, we first need to know the frequency of each number, for which we can use a **HashMap**. Once we have the frequency map, we can use a **Min Heap** to find the ‘K’ most frequently occurring number. In the **Min Heap**, instead of comparing numbers we will compare their frequencies in order to get frequently occurring numbers

```
using namespace std;


#include <iostream>

#include <queue>

#include <unordered_map>

#include <vector>


class TopKFrequentNumbers {

  struct valueCompare {

    char operator()(const pair<int, int> &x, const pair<int, int> &y) {
```

```
    return x.second > y.second;
  }
 };

public:
 static vector<int> findTopKFrequentNumbers(const vector<int> &nums, int k) {
  // find the frequency of each number
  unordered_map<int, int> numFrequencyMap;
  for (int n : nums) {
   numFrequencyMap[n]++;
  }

  priority_queue<pair<int, int>, vector<pair<int, int>>, valueCompare> minHeap;

  // go through all numbers of the numFrequencyMap and push them in the minHeap, which will have
  // top k frequent numbers. If the heap size is more than k, we remove the smallest (top) number
  for (auto entry : numFrequencyMap) {
   minHeap.push(entry);
   if (minHeap.size() > k) {
    minHeap.pop();
   }
  }

  // create a list of top k numbers
  vector<int> topNumbers;
  while (!minHeap.empty()) {
   topNumbers.push_back(minHeap.top().first);
   minHeap.pop();
  }
```

```
    return topNumbers;
  }
};

int main(int argc, char *argv[]) {
  vector<int> result =
      TopKFrequentNumbers::findTopKFrequentNumbers(vector<int>{1, 3, 5, 12, 11, 12, 11}, 2);
  cout << "Here are the K frequent numbers: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = TopKFrequentNumbers::findTopKFrequentNumbers(vector<int>{5, 12, 11, 3, 11}, 2);
  cout << "Here are the K frequent numbers: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(N+N*\log K)$.

Space complexity

The space complexity will be $O(N)$. Even though we are storing only 'K' numbers in the heap. For the frequency map, however, we need to store all the 'N' numbers.

Problem Statement

Given a string, sort it based on the decreasing frequency of its characters.

Example 1:

Input: "Programming"
Output: "rrggmmPiano"
Explanation: 'r', 'g', and 'm' appeared twice, so they need to appear before any other character.

Example 2:

Input: "abcbab"
Output: "bbbaac"
Explanation: 'b' appeared three times, 'a' appeared twice, and 'c' appeared only once.

Solution

This problem follows the **Top 'K' Elements** pattern, and shares similarities with Top 'K' Frequent Numbers.

We can follow the same approach as discussed in the Top 'K' Frequent Numbers problem. First, we will find the frequencies of all characters, then use a max-heap to find the most occurring characters.

```
using namespace std;


#include <iostream>

#include <queue>

#include <string>

#include <unordered_map>


class FrequencySort {

 public:

  struct valueCompare {

    bool operator()(const pair<char, int> &x, const pair<char, int> &y) {
```

```
    return y.second > x.second;
  }
};

static string sortCharacterByFrequency(const string &str) {
  // find the frequency of each character
  unordered_map<char, int> characterFrequencyMap;
  for (char chr : str) {
    characterFrequencyMap[chr]++;
  }

  priority_queue<pair<char, int>, vector<pair<char, int>>, valueCompare> maxHeap;

  // add all characters to the max heap
  for (auto entry : characterFrequencyMap) {
    maxHeap.push(entry);
  }

  // build a string, appending the most occurring characters first
  string sortedString = "";
  while (!maxHeap.empty()) {
    auto entry = maxHeap.top();
    maxHeap.pop();
    for (int i = 0; i < entry.second; i++) {
      sortedString += entry.first;
    }
  }

  return sortedString;
```

```
  }
};


int main(int argc, char *argv[]) {
  string result = FrequencySort::sortCharacterByFrequency("Programming");
  cout << "Here is the given string after sorting characters by frequency: " << result << endl;

  result = FrequencySort::sortCharacterByFrequency("abcbab");
  cout << "Here is the given string after sorting characters by frequency: " << result << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(D*logD)$ where 'D' is the number of distinct characters in the input string. This means, in the worst case, when all characters are unique the time complexity of the algorithm will be $O(N*logN)$ where 'N' is the total number of characters in the string.

Space complexity

The space complexity will be $O(N)$, as in the worst case, we need to store all the 'N' characters in the HashMap.

Problem Statement

Design a class to efficiently find the Kth largest element in a stream of numbers.

The class should have the following two things:

1. The constructor of the class should accept an integer array containing initial numbers from the stream and an integer 'K'.
2. The class should expose a function `add(int num)` which will store the given number and return the Kth largest number.

Example 1:

Input: [3, 1, 5, 12, 2, 11], K = 4
1. Calling add(6) should return '5'.
2. Calling add(13) should return '6'.
2. Calling add(4) should still return '6'.

Solution

This problem follows the **Top 'K' Elements** pattern and shares similarities with Kth Smallest number.

We can follow the same approach as discussed in the 'Kth Smallest number' problem. However, we will use a **Min Heap** (instead of a **Max Heap**) as we need to find the Kth largest number.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class KthLargestNumberInStream {

 public:

  struct numCompare {

    bool operator()(const int &x, const int &y) { return x > y; }

  };


  priority_queue<int, vector<int>, numCompare> minHeap;

  const int k;


  KthLargestNumberInStream(const vector<int> &nums, int k) : k(k) {

    // add the numbers in the min heap

    for (int i = 0; i < nums.size(); i++) {

      add(nums[i]);
```

```
    }
  }

  virtual int add(int num) {
    // add the new number in the min heap
    minHeap.push(num);

    // if heap has more than 'k' numbers, remove one number
    if ((int)minHeap.size() > this->k) {
      minHeap.pop();
    }

    // return the 'Kth largest number
    return minHeap.top();
  }
};

int main(int argc, char *argv[]) {
  KthLargestNumberInStream kthLargestNumber({3, 1, 5, 12, 2, 11}, 4);
  cout << "4th largest number is: " << kthLargestNumber.add(6) << endl;
  cout << "4th largest number is: " << kthLargestNumber.add(13) << endl;
  cout << "4th largest number is: " << kthLargestNumber.add(4) << endl;
}
```

Time complexity

The time complexity of the `add()` function will be $O(\log K)$ since we are inserting the new number in the heap.

Space complexity

The space complexity will be $O(K)$ for storing numbers in the heap.

Problem Statement

Given a sorted number array and two integers 'K' and 'X', find 'K' closest numbers to 'X' in the array. Return the numbers in the sorted order. 'X' is not necessarily present in the array.

Example 1:

Input: [5, 6, 7, 8, 9], K = 3, X = 7
Output: [6, 7, 8]

Example 2:

Input: [2, 4, 5, 6, 9], K = 3, X = 6
Output: [4, 5, 6]

Example 3:

Input: [2, 4, 5, 6, 9], K = 3, X = 10
Output: [5, 6, 9]

Solution

This problem follows the Top 'K' Numbers pattern. The biggest difference in this problem is that we need to find the closest (to 'X') numbers compared to finding the overall largest numbers. Another difference is that the given array is sorted.

Utilizing a similar approach, we can find the numbers closest to 'X' through the following algorithm:

1. Since the array is sorted, we can first find the number closest to 'X' through **Binary Search**. Let's say that number is 'Y'.
2. The 'K' closest numbers to 'Y' will be adjacent to 'Y' in the array. We can search in both directions of 'Y' to find the closest numbers.
3. We can use a heap to efficiently search for the closest numbers. We will take 'K' numbers in both directions of 'Y' and push them in a **Min**

Heap sorted by their absolute difference from ‘X’. This will ensure that the numbers with the smallest difference from ‘X’ (i.e., closest to ‘X’) can be extracted easily from the **Min Heap**.

4. Finally, we will extract the top ‘K’ numbers from the **Min Heap** to find the required numbers.

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <queue>

#include <vector>


class KClosestElements {

 public:

  struct numCompare {

    bool operator()(const pair<int, int> &x, const pair<int, int> &y) { return x.first > y.first; }

  };


  static vector<int> findClosestElements(const vector<int> &arr, int K, int X) {

    int index = binarySearch(arr, X);

    int low = index - K, high = index + K;

    low = max(low, 0);                    // 'low' should not be less than zero

    high = min(high, (int)arr.size() - 1);  // 'high' should not be greater the size of the array


    priority_queue<pair<int, int>, vector<pair<int, int>>, numCompare> minHeap;

    // add all candidate elements to the min heap, sorted by their absolute difference from 'X'

    for (int i = low; i <= high; i++) {

      minHeap.push(make_pair(abs(arr[i] - X), i));

    }
```

```
  // we need the top 'K' elements having smallest difference from 'X'
  vector<int> result;
  for (int i = 0; i < K; i++) {
    result.push_back(arr[minHeap.top().second]);
    minHeap.pop();
  }

  sort(result.begin(), result.end());
  return result;
 }

private:
 static int binarySearch(const vector<int> &arr, int target) {
  int low = 0;
  int high = (int)arr.size() - 1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (arr[mid] == target) {
      return mid;
    }
    if (arr[mid] < target) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  if (low > 0) {
    return low - 1;
```

```
    }
    return low;
  }
};

int main(int argc, char *argv[]) {
  vector<int> result = KClosestElements::findClosestElements(vector<int>{5, 6, 7, 8, 9}, 3, 7);
  cout << "'K' closest numbers to 'X' are: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = KClosestElements::findClosestElements(vector<int>{2, 4, 5, 6, 9}, 3, 6);
  cout << "'K' closest numbers to 'X' are: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(\log N + K*\log K)$. We need $O(\log N)$ for Binary Search and $O(K*\log K)$ to insert the numbers in the **Min Heap**, as well as to sort the output array.

Space complexity

The space complexity will be $O(K)$, as we need to put a maximum of $2K$ numbers in the heap.

Alternate Solution using Two Pointers

After finding the number closest to 'X' through **Binary Search**, we can use the **Two Pointers** approach to find the 'K' closest numbers. Let's say the closest number is 'Y'. We can have a `left` pointer to move back from 'Y' and a `right` pointer to move forward from 'Y'. At any stage, whichever number pointed out by the `left` or the `right` pointer gives the smaller difference from 'X' will be added to our result list.

To keep the resultant list sorted we can use a **Queue**. So whenever we take the number pointed out by the `left` pointer, we will append it at the beginning of the list and whenever we take the number pointed out by the `right` pointer we will append it at the end of the list.

Here is what our algorithm will look like:

```
using namespace std;


#include <deque>

#include <iostream>

#include <vector>


class KClosestElements {

 public:

  static vector<int> findClosestElements(const vector<int> &arr, int K, int X) {

    deque<int> result;

    int index = binarySearch(arr, X);

    int leftPointer = index;

    int rightPointer = index + 1;

    for (int i = 0; i < K; i++) {

      if (leftPointer >= 0 && rightPointer < (int)arr.size()) {

        int diff1 = abs(X - arr[leftPointer]);

        int diff2 = abs(X - arr[rightPointer]);

        if (diff1 <= diff2) {
```

```
      result.push_front(arr[leftPointer--]);  // append in the beginning
     } else {
      result.push_back(arr[rightPointer++]);  // append at the end
     }
    } else if (leftPointer >= 0) {
     result.push_front(arr[leftPointer--]);
    } else if (rightPointer < (int)arr.size()) {
     result.push_back(arr[rightPointer++]);
    }
  }
  vector<int> resultVec;
  std::move(std::begin(result), std::end(result), std::back_inserter(resultVec));
  return resultVec;
 }

private:
 static int binarySearch(const vector<int> &arr, int target) {
  int low = 0;
  int high = (int)arr.size() - 1;
  while (low <= high) {
   int mid = low + (high - low) / 2;
   if (arr[mid] == target) {
    return mid;
   }
   if (arr[mid] < target) {
    low = mid + 1;
   } else {
    high = mid - 1;
   }
```

```
    }
    if (low > 0) {
      return low - 1;
    }
    return low;
  }
};

int main(int argc, char *argv[]) {
  vector<int> result = KClosestElements::findClosestElements(vector<int>{5, 6, 7, 8, 9}, 3, 7);
  cout << "'K' closest numbers to 'X' are: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = KClosestElements::findClosestElements(vector<int>{2, 4, 5, 6, 9}, 3, 6);
  cout << "'K' closest numbers to 'X' are: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = KClosestElements::findClosestElements(vector<int>{2, 4, 5, 6, 9}, 3, 10);
  cout << "'K' closest numbers to 'X' are: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
```

}

Time complexity

The time complexity of the above algorithm is $O(\log N + K)$. We need $O(\log N)$ for Binary Search and $O(K)$ for finding the 'K' closest numbers using the two pointers.

Space complexity

If we ignoring the space required for the output list, the algorithm runs in constant space $O(1)$.

Problem Statement

Given an array of numbers and a number 'K', we need to remove 'K' numbers from the array such that we are left with maximum distinct numbers.

Example 1:

Input: [7, 3, 5, 8, 5, 3, 3], and K=2
Output: 3
Explanation: We can remove two occurrences of 3 to be left with 3 distinct numbers [7, 3, 8], we have to skip 5 because it is not distinct and occurred twice.
Another solution could be to remove one instance of '5' and '3' each to be left with three distinct numbers [7, 5, 8], in this case, we have to skip 3 because it occurred twice.

Example 2:

Input: [3, 5, 12, 11, 12], and K=3
Output: 2
Explanation: We can remove one occurrence of 12, after which all numbers will become distinct. Then we can delete any two numbers which will leave us 2 distinct numbers in the result.

Example 3:

Input: [1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5], and K=2
Output: 3
Explanation: We can remove one occurrence of '4' to get three distinct numbers.

Solution

This problem follows the Top 'K' Numbers pattern, and shares similarities with Top 'K' Frequent Numbers.

We can following a similar approach as discussed in Top 'K' Frequent Numbers problem:

1. First, we will find the frequencies of all the numbers.
2. Then, push all numbers that are not distinct (i.e., have a frequency higher than one) in a **Min Heap** based on their frequencies. At the same time, we will keep a running count of all the distinct numbers.
3. Following a greedy approach, in a stepwise fashion, we will remove the least frequent number from the heap (i.e., the top element of the min-heap), and try to make it distinct. We will see if we can remove all occurrences of a number except one. If we can, we will increment our running count of distinct numbers. We have to also keep a count of how many removals we have done.
4. If after removing elements from the heap, we are still left with some deletions, we have to remove some distinct elements.

```
using namespace std;


#include <iostream>

#include <queue>

#include <unordered_map>

#include <vector>


class MaximumDistinctElements {

 public:

  struct valueCompare {

    bool operator()(const pair<int, int> &x, const pair<int, int> &y) {

      return x.second > y.second;

    }
```

```
};

static int findMaximumDistinctElements(const vector<int> &nums, int k) {
 int distinctElementsCount = 0;
 if (nums.size() <= k) {
  return distinctElementsCount;
 }

 // find the frequency of each number
 unordered_map<int, int> numFrequencyMap;
 for (auto i : nums) {
  numFrequencyMap[i]++;
 }

 priority_queue<pair<int, int>, vector<pair<int, int>>, valueCompare> minHeap;

 // insert all numbers with frequency greater than '1' into the min-heap
 for (auto entry : numFrequencyMap) {
  if (entry.second == 1) {
   distinctElementsCount++;
  } else {
   minHeap.push(entry);
  }
 }

 // following a greedy approach, try removing the least frequent numbers first from the min-heap
 while (k > 0 && !minHeap.empty()) {
  auto entry = minHeap.top();
  minHeap.pop();
```

```
      // to make an element distinct, we need to remove all of its occurrences except one
      k -= entry.second - 1;
      if (k >= 0) {
        distinctElementsCount++;
      }
    }

    // if k > 0, this means we have to remove some distinct numbers
    if (k > 0) {
      distinctElementsCount -= k;
    }

    return distinctElementsCount;
  }
};

int main(int argc, char *argv[]) {
  int result =
      MaximumDistinctElements::findMaximumDistinctElements(vector<int>{7, 3, 5, 8, 5, 3, 3}, 2);
  cout << "Maximum distinct numbers after removing K numbers: " << result << endl;

  result = MaximumDistinctElements::findMaximumDistinctElements(vector<int>{3, 5, 12, 11, 12}, 3);
  cout << "Maximum distinct numbers after removing K numbers: " << result << endl;

  result = MaximumDistinctElements::findMaximumDistinctElements(
      vector<int>{1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5}, 2);
  cout << "Maximum distinct numbers after removing K numbers: " << result << endl;
}
```

Time complexity

Since we will insert all numbers in a **HashMap** and a **Min Heap**, this will take $O(N*logN)$ where 'N' is the total input numbers. While extracting numbers from the heap, in the worst case, we will need to take out 'K' numbers. This will happen when we have at least 'K' numbers with a frequency of two. Since the heap can have a maximum of 'N/2' numbers, therefore, extracting an element from the heap will take $O(logN)$ and extracting 'K' numbers will take $O(KlogN)$. So overall, the time complexity of our algorithm will be $O(N*logN + KlogN)$.

We can optimize the above algorithm and only push 'K' elements in the heap, as in the worst case we will be extracting 'K' elements from the heap. This optimization will reduce the overall time complexity to $O(N*logK + KlogK)$.

Space complexity

The space complexity will be $O(N)$ as, in the worst case, we need to store all the 'N' characters in the **HashMap**.

Problem Statement

Given an array, find the sum of all numbers between the K1'th and K2'th smallest elements of that array.

Example 1:

Input: [1, 3, 12, 5, 15, 11], and K1=3, K2=6
Output: 23
Explanation: The 3rd smallest number is 5 and 6th smallest number 15. The sum of numbers coming between 5 and 15 is 23 (11+12).

Example 2:

Input: [3, 5, 8, 7], and K1=1, K2=4
Output: 12

Explanation: The sum of the numbers between the 1st smallest number (3) and the 4th smallest number (8) is 12 (5+7).

Solution

This problem follows the Top 'K' Numbers pattern, and shares similarities with Kth Smallest Number.

We can find the sum of all numbers coming between the K1'th and K2'th smallest numbers in the following steps:

1. First, insert all numbers in a min-heap.
2. Remove the first `K1` smallest numbers from the min-heap.
3. Now take the next `K2-K1-1` numbers out of the heap and add them. This sum will be our required output.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class SumOfElements {

 public:

  struct numCompare {

    bool operator()(const int &x, const int &y) { return x > y; }

  };


  static int findSumOfElements(const vector<int> &nums, int k1, int k2) {

    priority_queue<int, vector<int>, numCompare> minHeap;


    // insert all numbers to the min heap
```

```
    for (int i = 0; i < nums.size(); i++) {
      minHeap.push(nums[i]);
    }

    // remove k1 small numbers from the min heap
    for (int i = 0; i < k1; i++) {
      minHeap.pop();
    }

    int elementSum = 0;
    // sum next k2-k1-1 numbers
    for (int i = 0; i < k2 - k1 - 1; i++) {
      elementSum += minHeap.top();
      minHeap.pop();
    }

    return elementSum;
  }
};

int main(int argc, char *argv[]) {
  int result = SumOfElements::findSumOfElements(vector<int>{1, 3, 12, 5, 15, 11}, 3, 6);
  cout << "Sum of all numbers between k1 and k2 smallest numbers: " << result << endl;

  result = SumOfElements::findSumOfElements(vector<int>{3, 5, 8, 7}, 1, 4);
  cout << "Sum of all numbers between k1 and k2 smallest numbers: " << result << endl;
}
```

Time complexity

Since we need to put all the numbers in a min-heap, the time complexity of the above algorithm will be $O(N*logN)$ where 'N' is the total input numbers.

Space complexity

The space complexity will be $O(N)$, as we need to store all the 'N' numbers in the heap.

Alternate Solution

We can iterate the array and use a max-heap to keep track of the top K2 numbers. We can, then, add the top `K2-K1-1` numbers in the max-heap to find the sum of all numbers coming between the K1'th and K2'th smallest numbers. Here is what the algorithm will look like:

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class SumOfElements {

 public:

  static int findSumOfElements(const vector<int> &nums, int k1, int k2) {

    priority_queue<int> maxHeap;

    // keep smallest k2 numbers in the max heap

    for (int i = 0; i < nums.size(); i++) {

      if (i < k2 - 1) {

        maxHeap.push(nums[i]);

      } else if (nums[i] < maxHeap.top()) {

        maxHeap.pop();  // as we are interested only in the smallest k2 numbers
```

```
      maxHeap.push(nums[i]);
    }
   }

   // get the sum of numbers between k1 and k2 indices
   // these numbers will be at the top of the max heap
   int elementSum = 0;
   for (int i = 0; i < k2 - k1 - 1; i++) {
    elementSum += maxHeap.top();
    maxHeap.pop();
   }

   return elementSum;
  }
};

int main(int argc, char *argv[]) {
  int result = SumOfElements::findSumOfElements(vector<int>{1, 3, 12, 5, 15, 11}, 3, 6);
  cout << "Sum of all numbers between k1 and k2 smallest numbers: " << result << endl;

  result = SumOfElements::findSumOfElements(vector<int>{3, 5, 8, 7}, 1, 4);
  cout << "Sum of all numbers between k1 and k2 smallest numbers: " << result << endl;
}
```

Time complexity

Since we need to put only the top K2 numbers in the max-heap at any time, the time complexity of the above algorithm will be O(N*logK2)$O(N*logK2)$.

Space complexity

The space complexity will be $O(K2)$, as we need to store the smallest 'K2' numbers in the heap.

Problem Statement

Given a string, find if its letters can be rearranged in such a way that no two same characters come next to each other.

Example 1:

Input: "aappp"
Output: "papap"
Explanation: In "papap", none of the repeating characters come next to each other.

Example 2:

Input: "Programming"
Output: "rgmrgmPiano" or "gmringmrPoa" or "gmrPagimnor", etc.
Explanation: None of the repeating characters come next to each other.

Example 3:

Input: "aapa"
Output: ""
Explanation: In all arrangements of "aapa", atleast two 'a' will come together e.g., "apaa", "paaa".

Solution

This problem follows the Top 'K' Numbers pattern. We can follow a greedy approach to find an arrangement of the given string where no two same characters come next to each other.

We can work in a stepwise fashion to create a string with all characters from the input string. Following a greedy approach, we should first append the most frequent characters to the output strings, for which we can use a **Max Heap**. By appending the most frequent character first, we have the best chance to find a string where no two same characters come next to each other.

So in each step, we should append one occurrence of the highest frequency character to the output string. We will not put this character back in the heap to ensure that no two same characters are adjacent to each other. In the next step, we should process the next most frequent character from the heap in the same way and then, at the end of this step, insert the character from the previous step back to the heap after decrementing its frequency.

Following this algorithm, if we can append all the characters from the input string to the output string, we would have successfully found an arrangement of the given string where no two same characters appeared adjacent to each other.

```
using namespace std;


#include <iostream>

#include <queue>

#include <string>

#include <unordered_map>

#include <vector>


class RearrangeString {

 public:

  struct valueCompare {

    char operator()(const pair<int, int> &x, const pair<int, int> &y) {

      return y.second > x.second;

    }

  };


  static string rearrangeString(const string &str) {

    unordered_map<char, int> charFrequencyMap;

    for (char chr : str) {

      charFrequencyMap[chr]++;

    }
```

```
    priority_queue<pair<char, int>, vector<pair<char, int>>, valueCompare> maxHeap;

    // add all characters to the max heap
    for (auto entry : charFrequencyMap) {
      maxHeap.push(entry);
    }

    pair<char, int> previousEntry(-1, -1);
    string resultString = "";
    while (!maxHeap.empty()) {
      pair<char, int> currentEntry = maxHeap.top();
      maxHeap.pop();
      // add the previous entry back in the heap if its frequency is greater than zero
      if (previousEntry.second > 0) {
        maxHeap.push(previousEntry);
      }
      // append the current character to the result string and decrement its count
      resultString += currentEntry.first;
      currentEntry.second--;
      previousEntry = currentEntry;
    }

    // if we were successful in appending all the characters to the result string, return it
    return resultString.length() == str.length() ? resultString : "";
  }
};

int main(int argc, char *argv[]) {
```

```
  cout << "Rearranged string: " << RearrangeString::rearrangeString("aappp") << endl;
  cout << "Rearranged string: " << RearrangeString::rearrangeString("Programming") << endl;
  cout << "Rearranged string: " << RearrangeString::rearrangeString("aapa") << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(N*logN)$ where 'N' is the number of characters in the input string.

Space complexity

The space complexity will be $O(N)$, as in the worst case, we need to store all the 'N' characters in the **HashMap**.

Rearrange String K Distance Apart (hard)

Given a string and a number 'K', find if the string can be rearranged such that the same characters are at least 'K' distance apart from each other.

Example 1:

Input: "mmpp", K=2
Output: "mpmp" or "pmpm"
Explanation: All same characters are 2 distance apart.

Example 2:

Input: "Programming", K=3
Output: "rgmPrgmiano" or "gmringmrPoa" or "gmrPagimnor" and a few more
Explanation: All same characters are 3 distance apart.

Example 3:

Input: "aab", K=2
Output: "aba"
Explanation: All same characters are 2 distance apart.

Example 4:

Input: "aappa", K=3
Output: ""
Explanation: We cannot find an arrangement of the string where any two 'a' are 3 distance apart.

Solution

This problem follows the Top 'K' Numbers pattern and is quite similar to Rearrange String. The only difference is that in the 'Rearrange String' the same characters need not be adjacent i.e., they should be at least '2' distance apart (in other words, there should be at least one character between two same characters), while in the current problem, the same characters should be 'K' distance apart.

Following a similar approach, since we were inserting a character back in the heap in the next iteration, in this problem, we will re-insert the character after 'K' iterations. We can keep track of previous characters in a queue to insert them back in the heap after 'K' iterations.

```
using namespace std;


#include <iostream>

#include <queue>

#include <string>

#include <unordered_map>

#include <vector>


class RearrangeStringKDistanceApart {

 public:

  struct valueCompare {

    char operator()(const pair<int, int> &x, const pair<int, int> &y) {

      return y.second > x.second;

    }

  };
```

```
static string reorganizeString(const string &str, int k) {
 if (k <= 1) return str;

 unordered_map<char, int> charFrequencyMap;
 for (char chr : str) {
  charFrequencyMap[chr]++;
 }

 priority_queue<pair<char, int>, vector<pair<char, int>>, valueCompare> maxHeap;

 // add all characters to the max heap
 for (auto entry : charFrequencyMap) {
  maxHeap.push(entry);
 }

 queue<pair<char, int>> queue;
 string resultString;
 while (!maxHeap.empty()) {
  auto currentEntry = maxHeap.top();
  maxHeap.pop();
  // append the current character to the result string and decrement its count
  resultString += currentEntry.first;
  currentEntry.second--;
  queue.push(currentEntry);
  if (queue.size() == k) {
   auto entry = queue.front();
   queue.pop();
   if (entry.second > 0) {
    maxHeap.push(entry);
```

```
      }
     }
    }

    // if we were successful in appending all the characters to the result string, return it
    return resultString.length() == str.length() ? resultString : "";
  }
};

int main(int argc, char *argv[]) {
  cout << "Reorganized string: "
       << RearrangeStringKDistanceApart::reorganizeString("Programming", 3) << endl;
  cout << "Reorganized string: "
       << RearrangeStringKDistanceApart::reorganizeString("mmpp", 2) << endl;
  cout << "Reorganized string: "
       << RearrangeStringKDistanceApart::reorganizeString("aab", 2) << endl;
  cout << "Reorganized string: " << RearrangeStringKDistanceApart::reorganizeString("aappa", 3)
       << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(N*logN)$ where 'N' is the number of characters in the input string.

Space complexity

The space complexity will be $O(N)$, as in the worst case, we need to store all the 'N' characters in the HashMap.

Scheduling Tasks (hard)

You are given a list of tasks that need to be run, in any order, on a server. Each task will take one CPU interval to execute but once a task has finished, it has a cooling period during which it can't be run again. If the cooling period for all tasks is 'K' intervals, find the minimum number of CPU intervals that the server needs to finish all tasks.

If at any time the server can't execute any task then it must stay idle.

Example 1:

Input: [a, a, a, b, c, c], K=2
Output: 7
Explanation: a -> c -> b -> a -> c -> idle -> a

Example 2:

Input: [a, b, a], K=3
Output: 5
Explanation: a -> b -> idle -> idle -> a

Solution

This problem follows the Top 'K' Elements pattern and is quite similar to Rearrange String K Distance Apart. We need to rearrange tasks such that same tasks are 'K' distance apart.

Following a similar approach, we will use a **Max Heap** to execute the highest frequency task first. After executing a task we decrease its frequency and put it in a waiting list. In each iteration, we will try to execute as many as `k+1` tasks. For the next iteration, we will put all the waiting tasks back in the **Max Heap**. If, for any iteration, we are not able to execute `k+1` tasks, the CPU has to remain idle for the remaining time in the next iteration.

```
using namespace std;


#include <iostream>

#include <queue>

#include <unordered_map>

#include <vector>
```

```
class TaskScheduler {
 public:
  struct valueCompare {
    char operator()(const pair<int, int> &x, const pair<int, int> &y) {
      return y.second > x.second;
    }
  };

  static int scheduleTasks(vector<char> &tasks, int k) {
    int intervalCount = 0;
    unordered_map<char, int> taskFrequencyMap;
    for (char chr : tasks) {
      taskFrequencyMap[chr]++;
    }

    priority_queue<pair<char, int>, vector<pair<char, int>>, valueCompare> maxHeap;

    // add all tasks to the max heap
    for (auto entry : taskFrequencyMap) {
      maxHeap.push(entry);
    }

    while (!maxHeap.empty()) {
      vector<pair<char, int>> waitList;
      int n = k + 1;  // try to execute as many as 'k+1' tasks from the max-heap
      for (; n > 0 && !maxHeap.empty(); n--) {
        intervalCount++;
        auto currentEntry = maxHeap.top();
        maxHeap.pop();
```

```
      if (currentEntry.second > 1) {
        currentEntry.second--;
        waitList.push_back(currentEntry);
      }
    }
    // put all the waiting list back on the heap
    for (auto it = waitList.begin(); it != waitList.end(); it++) {
      maxHeap.push(*it);
    }
    if (!maxHeap.empty()) {
      intervalCount += n;  // we'll be having 'n' idle intervals for the next iteration
    }
  }

  return intervalCount;
 }
};

int main(int argc, char *argv[]) {
 vector<char> tasks = {'a', 'a', 'a', 'b', 'c', 'c'};
 cout << "Minimum intervals needed to execute all tasks: "
      << TaskScheduler::scheduleTasks(tasks, 2) << endl;

 tasks = vector<char>{'a', 'b', 'a'};
 cout << "Minimum intervals needed to execute all tasks: "
      << TaskScheduler::scheduleTasks(tasks, 3) << endl;
}
```

Time complexity

The time complexity of the above algorithm is $O(N*logN)$ where 'N' is the number of tasks. Our `while loop` will iterate once for each occurrence of the task in the input (i.e. 'N') and in each iteration we will remove a task from the heap which will take $O(logN)$ time. Hence the overall time complexity of our algorithm is $O(N*logN)$.

Space complexity

The space complexity will be $O(N)$, as in the worst case, we need to store all the 'N' tasks in the **HashMap**.

Frequency Stack (hard)

Design a class that simulates a Stack data structure, implementing the following two operations:

1. `push(int num)`: Pushes the number 'num' on the stack.
2. `pop()`: Returns the most frequent number in the stack. If there is a tie, return the number which was pushed later.

Example:

After following push operations: push(1), push(2), push(3), push(2), push(1), push(2), push(5)

1. pop() should return 2, as it is the most frequent number
2. Next pop() should return 1
3. Next pop() should return 2

Solution

This problem follows the Top 'K' Elements pattern, and shares similarities with Top 'K' Frequent Numbers.

We can use a **Max Heap** to store the numbers. Instead of comparing the numbers we will compare their frequencies so that the root of the heap is always the most frequently occurring number. There are two issues that need to be resolved though:

1. How can we keep track of the frequencies of numbers in the heap? When we are pushing a new number to the **Max Heap**, we don't know how many times the number has already appeared in the **Max Heap**. To resolve this, we will maintain a **HashMap** to store the current frequency of each number. Thus whenever we push a new number in the heap, we will increment its frequency in the HashMap and when we pop, we will decrement its frequency.
2. If two numbers have the same frequency, we will need to return the number which was pushed later while popping. To resolve this, we need to attach a sequence number to every number to know which number came first.

In short, we will keep three things with every number that we push to the heap:

1. number // value of the number
2. frequency // current frequency of the number when it was pushed to the heap
3. sequenceNumber // a sequence number, to know what number came first

```
using namespace std;


#include <iostream>

#include <queue>

#include <string>

#include <unordered_map>


class Element {

 public:

  int number = 0;

  int frequency = 0;

  int sequenceNumber = 0;


  Element(int number, int frequency, int sequenceNumber) {

    this->number = number;

    this->frequency = frequency;
```

```
    this->sequenceNumber = sequenceNumber;
  }
};

class FrequencyStack {
 public:
  struct frequencyCompare {
    bool operator()(const Element &e1, const Element &e2) {
      if (e1.frequency != e2.frequency) {
        return e2.frequency > e1.frequency;
      }
      // if both elements have same frequency, return the one that was pushed later
      return e2.sequenceNumber > e1.sequenceNumber;
    }
  };

  int sequenceNumber = 0;
  priority_queue<Element, vector<Element>, frequencyCompare> maxHeap;
  unordered_map<int, int> frequencyMap;

  virtual void push(int num) {
    frequencyMap[num] += 1;
    maxHeap.push({num, frequencyMap[num], sequenceNumber++});
  }

  virtual int pop() {
    int num = maxHeap.top().number;
    maxHeap.pop();
```

```
    // decrement the frequency or remove if this is the last number
    if (frequencyMap[num] > 1) {
      frequencyMap[num] -= 1;
    } else {
      frequencyMap.erase(num);
    }

    return num;
  }
};

int main(int argc, char *argv[]) {
  FrequencyStack frequencyStack;
  frequencyStack.push(1);
  frequencyStack.push(2);
  frequencyStack.push(3);
  frequencyStack.push(2);
  frequencyStack.push(1);
  frequencyStack.push(2);
  frequencyStack.push(5);
  cout << frequencyStack.pop() << endl;
  cout << frequencyStack.pop() << endl;
  cout << frequencyStack.pop() << endl;
}
```

Time complexity

The time complexity of `push()` and `pop()` is $O(\log N)$ where 'N' is the current number of elements in the heap.

Space complexity

We will need $O(N)$ space for the heap and the map, so the overall space complexity of the algorithm is $O(N)$.

1. Introduction

This pattern helps us solve problems that involve a list of sorted arrays.

Whenever we are given 'K' sorted arrays, we can use a **Heap** to efficiently perform a sorted traversal of all the elements of all arrays. We can push the smallest (first) element of each sorted array in a **Min Heap** to get the overall minimum. While inserting elements to the **Min Heap** we keep track of which array the element came from. We can, then, remove the top element from the heap to get the smallest element and push the next element from the same array, to which this smallest element belonged, to the heap. We can repeat this process to make a sorted traversal of all elements.

Let's see this pattern in action.

Problem Statement

Given an array of 'K' sorted LinkedLists, merge them into one sorted list.

Example 1:

Input: L1=[2, 6, 8], L2=[3, 6, 7], L3=[1, 3, 4]
Output: [1, 2, 3, 3, 4, 6, 6, 7, 8]

Example 2:

Input: L1=[5, 8, 9], L2=[1, 7]
Output: [1, 5, 7, 8, 9]

Solution

A brute force solution could be to add all elements of the given 'K' lists to one list and sort it. If there are a total of 'N' elements in all the input lists, then the brute force solution will have a time complexity of $O(N*logN)$ as

we will need to sort the merged list. Can we do better than this? How can we utilize the fact that the input lists are individually sorted?

If we have to find the smallest element of all the input lists, we have to compare only the smallest (i.e. the first) element of all the lists. Once we have the smallest element, we can put it in the merged list. Following a similar pattern, we can then find the next smallest element of all the lists to add it to the merged list.

The best data structure that comes to mind to find the smallest number among a set of 'K' numbers is a **Heap**. Let's see how can we use a heap to find a better algorithm.

1. We can insert the first element of each array in a **Min Heap**.
2. After this, we can take out the smallest (top) element from the heap and add it to the merged list.
3. After removing the smallest element from the heap, we can insert the next element of the same list into the heap.
4. We can repeat steps 2 and 3 to populate the merged list in sorted order.

Let's take the Example-1 mentioned above to go through each step of our algorithm:

Given lists: `L1=[2, 6, 8]`, `L2=[3, 6, 7]`, `L3=[1, 3, 4]`

1. After inserting the 1st element of each list, the heap will have the following elements:
2. We'll take the top number from the heap, insert it into the merged list and add the next number in the heap.
3. Again, we'll take the top element of the heap, insert it into the merged list and add the next number to the heap.
4. Repeating the above step, take the top element of the heap, insert it into the merged list and add the next number to the heap. As there are two 3s in the heap, we can pick anyone but we need to take the next element from the corresponding list to insert in the heap.

We'll repeat the above step to populate our merged array.

```
using namespace std;
```

```
#include <iostream>
#include <queue>
#include <vector>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    this->next = nullptr;
  }
};

class MergeKSortedLists {
 public:
  struct valueCompare {
    bool operator()(const ListNode *x, const ListNode *y) { return x->value > y->value; }
  };

  static ListNode *merge(const vector<ListNode *> &lists) {
    priority_queue<ListNode *, vector<ListNode *>, valueCompare> minHeap;

    // put the root of each list in the min heap
    for (auto root : lists) {
      if (root != nullptr) {
        minHeap.push(root);
      }
```

```
    }

    // take the smallest (top) element form the min-heap and add it to the result;
    // if the top element has a next element add it to the heap
    ListNode *resultHead = nullptr, *resultTail = nullptr;
    while (!minHeap.empty()) {
      ListNode *node = minHeap.top();
      minHeap.pop();
      if (resultHead == nullptr) {
        resultHead = resultTail = node;
      } else {
        resultTail->next = node;
        resultTail = resultTail->next;
      }
      if (node->next != nullptr) {
        minHeap.push(node->next);
      }
    }

    return resultHead;
  }
};

int main(int argc, char *argv[]) {
  ListNode *l1 = new ListNode(2);
  l1->next = new ListNode(6);
  l1->next->next = new ListNode(8);

  ListNode *l2 = new ListNode(3);
```

```
  l2->next = new ListNode(6);
  l2->next->next = new ListNode(7);

  ListNode *l3 = new ListNode(1);
  l3->next = new ListNode(3);
  l3->next->next = new ListNode(4);

  ListNode *result = MergeKSortedLists::merge(vector<ListNode *>{l1, l2, l3});
  cout << "Here are the elements form the merged list: ";
  while (result != nullptr) {
    cout << result->value << " ";
    result = result->next;
  }
}
```

Time complexity

Since we'll be going through all the elements of all arrays and will be removing/adding one element to the heap in each step, the time complexity of the above algorithm will be $O(N*\log K)$, $O(N*logK)$, where 'N' is the total number of elements in all the 'K' input arrays.

Space complexity

The space complexity will be $O(K)$ $O(K)$ because, at any time, our min-heap will be storing one number from all the 'K' input arrays.

Problem Statement

Given 'M' sorted arrays, find the K'th smallest number among all the arrays.

Example 1:

Input: L1=[2, 6, 8], L2=[3, 6, 7], L3=[1, 3, 4], K=5
Output: 4

Explanation: The 5th smallest number among all the arrays is 4, this can be verified from the merged list of all the arrays: [1, 2, 3, 3, 4, 6, 6, 7, 8]

Example 2:

Input: L1=[5, 8, 9], L2=[1, 7], K=3
Output: 7
Explanation: The 3rd smallest number among all the arrays is 7.

Solution

This problem follows the **K-way merge** pattern and we can follow a similar approach as discussed in Merge K Sorted Lists.

We can start merging all the arrays, but instead of inserting numbers into a merged list, we will keep count to see how many elements have been inserted in the merged list. Once that count is equal to 'K', we have found our required number.

A big difference from Merge K Sorted Lists is that in this problem, the input is a list of arrays compared to LinkedLists. This means that when we want to push the next number in the heap we need to know what the index of the current number in the current array was. To handle this, we will need to keep track of the array and the element indices.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class KthSmallestInMSortedArrays {

 public:

  struct valueCompare {

    bool operator()(const pair<int, pair<int, int>> &x, const pair<int, pair<int, int>> &y) {

      return x.first > y.first;

    }
```

```
};

static int findKthSmallest(const vector<vector<int>> &lists, int k) {
  priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>, valueCompare>
      minHeap;

  // put the 1st element of each array in the min heap
  for (int i = 0; i < lists.size(); i++) {
    if (!lists[i].empty()) {
      minHeap.push(make_pair(lists[i][0], make_pair(i, 0)));
    }
  }

  // take the smallest (top) element form the min heap, if the running count is equal to k return
  // the number if the array of the top element has more elements, add the next element to the
  // heap
  int numberCount = 0, result = 0;
  while (!minHeap.empty()) {
    auto node = minHeap.top();
    minHeap.pop();
    result = node.first;
    if (++numberCount == k) {
      break;
    }
    node.second.second++;
    if (lists[node.second.first].size() > node.second.second) {
      node.first = lists[node.second.first][node.second.second];
      minHeap.push(node);
    }
```

```
    }


    return result;
  }
};


int main(int argc, char *argv[]) {
  vector<vector<int>> lists = {{2, 6, 8}, {3, 6, 7}, {1, 3, 4}};
  int result = KthSmallestInMSortedArrays::findKthSmallest(lists, 5);
  cout << "Kth smallest number is: " << result;
}
```

Time complexity

Since we'll be going through at most 'K' elements among all the arrays, and we will remove/add one element in the heap in each step, the time complexity of the above algorithm will be $O(K*logM)$ where 'M' is the total number of input arrays.

Space complexity

The space complexity will be $O(M)$ because, at any time, our min-heap will be storing one number from all the 'M' input arrays.

Similar Problems

Problem 1: Given 'M' sorted arrays, find the median number among all arrays.

Solution: This problem is similar to our parent problem with K=Median. So if there are 'N' total numbers in all the arrays we need to find the K'th minimum number where $K=N/2$.

Problem 2: Given a list of 'K' sorted arrays, merge them into one sorted list.

Solution: This problem is similar to Merge K Sorted Lists except that the input is a list of arrays compared to **LinkedLists**. To handle this, we can use a similar approach as discussed in our parent problem by keeping a track of the array and the element indices.

Problem Statement

Given an N * N$N*N$ matrix where each row and column is sorted in ascending order, find the Kth smallest element in the matrix.

Example 1:

```
Input: Matrix=[
   [2, 6, 8],
   [3, 7, 10],
   [5, 8, 11]
 ],
 K=5
Output: 7
Explanation: The 5th smallest number in the matrix is 7.
```

Solution

This problem follows the **K-way merge** pattern and can be easily converted to Kth Smallest Number in M Sorted Lists. As each row (or column) of the given matrix can be seen as a sorted list, we essentially need to find the Kth smallest number in 'N' sorted lists.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class KthSmallestInSortedMatrix {

 public:

  struct numCompare {
```

```
  bool operator()(const pair<int, pair<int, int>> &x, const pair<int, pair<int, int>> &y) {
    return x.first > y.first;
  }
};

static int findKthSmallest(vector<vector<int>> &matrix, int k) {
  int n = matrix.size();
  priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>, numCompare>
      minHeap;

  // put the 1st element of each row in the min heap
  // we don't need to push more than 'k' elements in the heap
  for (int i = 0; i < n && i < k; i++) {
    minHeap.push(make_pair(matrix[i][0], make_pair(i, 0)));
  }

  // take the smallest element form the min heap, if the running count is equal to k return the
  // number if the row of the top element has more elements, add the next element to the heap
  int numberCount = 0, result = 0;
  while (!minHeap.empty()) {
    auto heapTop = minHeap.top();
    minHeap.pop();
    result = heapTop.first;
    if (++numberCount == k) {
      break;
    }

    heapTop.second.second++;
    if (n > heapTop.second.second) {
```

```
      heapTop.first = matrix[heapTop.second.first][heapTop.second.second];
      minHeap.push(heapTop);
    }
   }
   return result;
  }
};

int main(int argc, char *argv[]) {
  vector<vector<int>> matrix2 = {vector<int>{2, 6, 8}, vector<int>{3, 7, 10},
                                 vector<int>{5, 8, 11}};
  int result = KthSmallestInSortedMatrix::findKthSmallest(matrix2, 5);
  cout << "Kth smallest number is: " << result << endl;
}
```

Time complexity

First, we inserted at most 'K' or one element from each of the 'N' rows, which will take $O(\min(K, N))$. Then we went through at most 'K' elements in the matrix and remove/add one element in the heap in each step. As we can't have more than 'N' elements in the heap in any condition, therefore, the overall time complexity of the above algorithm will be $O(\min(K, N) + K*\log N)$.

Space complexity

The space complexity will be $O(N)$ because, in the worst case, our min-heap will be storing one number from each of the 'N' rows.

An Alternate Solution

Since each row and column of the matrix is sorted, is it possible to use **Binary Search** to find the Kth smallest number?

The biggest problem to use **Binary Search**, in this case, is that we don't have a straightforward sorted array, instead, we have a matrix. As we remember, in **Binary Search**, we calculate the middle index of the search space ('1' to 'N') and see if our required number is pointed out by the middle index; if not we either search in the lower half or the upper half. In a sorted matrix, we can't really find a middle. Even if we do consider some index as middle, it is not straightforward to find the search space containing numbers bigger or smaller than the number pointed out by the middle index.

An alternative could be to apply the **Binary Search** on the "number range" instead of the "index range". As we know that the smallest number of our matrix is at the top left corner and the biggest number is at the bottom right corner. These two numbers can represent the "range" i.e., the `start` and the `end` for the **Binary Search**. Here is how our algorithm will work:

1. Start the **Binary Search** with `start = matrix[0][0]` and `end = matrix[n-1][n-1]`.
2. Find `middle` of the `start` and the `end`. This `middle` number is NOT necessarily an element in the matrix.
3. Count all the numbers smaller than or equal to `middle` in the matrix. As the matrix is sorted, we can do this in $O(N)$.
4. While counting, we can keep track of the "smallest number greater than the `middle`" (let's call it `n1`) and at the same time the "biggest number less than or equal to the `middle`" (let's call it `n2`). These two numbers will be used to adjust the "number range" for the **Binary Search** in the next iteration.
5. If the count is equal to 'K', `n1` will be our required number as it is the "biggest number less than or equal to the `middle`", and is definitely present in the matrix.
6. If the count is less than 'K', we can update `start = n2` to search in the higher part of the matrix and if the count is greater than 'K', we can update `end = n1` to search in the lower part of the matrix in the next iteration.

Here is the visual representation of our algorithm:

```
using namespace std;


#include <iostream>

#include <queue>
```

```
#include <vector>

class KthSmallestInSortedMatrix {
 public:
  static int findKthSmallest(vector<vector<int>> &matrix, int k) {
    int n = matrix.size();
    int start = matrix[0][0], end = matrix[n - 1][n - 1];
    while (start < end) {
      int mid = start + (end - start) / 2;
      // first number is the smallest and the second number is the largest
      pair<int, int> smallLargePair = {matrix[0][0], matrix[n - 1][n - 1]};
      int count = countLessEqual(matrix, mid, smallLargePair);
      if (count == k) {
        return smallLargePair.first;
      }

      if (count < k) {
        start = smallLargePair.second;  // search higher
      } else {
        end = smallLargePair.first;  // search lower
      }
    }

    return start;
  }

 private:
  static int countLessEqual(vector<vector<int>> &matrix, int mid, pair<int, int> &smallLargePair) {
    int count = 0;
```

```
    int n = matrix.size(), row = n - 1, col = 0;
    while (row >= 0 && col < n) {
      if (matrix[row][col] > mid) {
        // as matrix[row][col] is bigger than the mid, let's keep track of the
        // smallest number greater than the mid
        smallLargePair.second = min(smallLargePair.second, matrix[row][col]);
        row--;
      } else {
        // as matrix[row][col] is less than or equal to the mid, let's keep track of the
        // biggest number less than or equal to the mid
        smallLargePair.first = max(smallLargePair.first, matrix[row][col]);
        count += row + 1;
        col++;
      }
    }
    return count;
  }
};


int main(int argc, char *argv[]) {
  vector<vector<int>> matrix2 = {vector<int>{2, 6, 8}, vector<int>{3, 7, 10},
                                 vector<int>{5, 8, 11}};
  int result = KthSmallestInSortedMatrix::findKthSmallest(matrix2, 5);
  cout << "Kth smallest number is: " << result << endl;
}
```

Time complexity

The **Binary Search** could take $O(\log(\text{max-min}))$ $O(log(max-min))$ iterations where ‘max’ is the largest and ‘min’ is the

smallest element in the matrix and in each iteration we take $O(N)$ for counting, therefore, the overall time complexity of the algorithm will be $O(N*log(max-min))$.

Space complexity #

The algorithm runs in constant space O(1).

Problem Statement

Given 'M' sorted arrays, find the smallest range that includes at least one number from each of the 'M' lists.

Example 1:

Input: L1=[1, 5, 8], L2=[4, 12], L3=[7, 8, 10]
Output: [4, 7]
Explanation: The range [4, 7] includes 5 from L1, 4 from L2 and 7 from L3.

Example 2:

Input: L1=[1, 9], L2=[4, 12], L3=[7, 10, 16]
Output: [9, 12]
Explanation: The range [9, 12] includes 9 from L1, 12 from L2 and 10 from L3.

Solution

This problem follows the **K-way merge** pattern and we can follow a similar approach as discussed in Merge K Sorted Lists.

We can start by inserting the first number from all the arrays in a min-heap. We will keep track of the largest number that we have inserted in the heap (let's call it `currentMaxNumber`).

In a loop, we'll take the smallest (top) element from the min-heap and`currentMaxNumber` has the largest element that we inserted in the heap. If these two numbers give us a smaller range, we'll update our range. Finally, if the array of the top element has more elements, we'll insert the next element to the heap.

We can finish searching the minimum range as soon as an array is completed or, in other terms, the heap has less than 'M' elements.

```
using namespace std;

#include <iostream>
#include <limits>
#include <queue>
#include <vector>

class SmallestRange {
 public:
  struct valueCompare {
    bool operator()(const pair<int, pair<int, int>> &x, const pair<int, pair<int, int>> &y) {
      return x.first > y.first;
    }
  };

  static pair<int, int> findSmallestRange(const vector<vector<int>> &lists) {
    // we will store the actual number, list index and element index in the heap
    priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>, valueCompare>
        minHeap;

    int rangeStart = 0, rangeEnd = numeric_limits<int>::max(),
        currentMaxNumber = numeric_limits<int>::min();
    // put the 1st element of each array in the min heap
    for (int i = 0; i < lists.size(); i++) {
      if (!lists[i].empty()) {
        minHeap.push(make_pair(lists[i][0], make_pair(i, 0)));
        currentMaxNumber = max(currentMaxNumber, lists[i][0]);
      }
    }
```

```
    // take the smallest (top) element form the min heap, if it gives us smaller range, update the
    // ranges if the array of the top element has more elements, insert the next element in the heap
    while (minHeap.size() == lists.size()) {
      auto node = minHeap.top();
      minHeap.pop();
      if (rangeEnd - rangeStart > currentMaxNumber - node.first) {
        rangeStart = node.first;
        rangeEnd = currentMaxNumber;
      }
      node.second.second++;
      if (lists[node.second.first].size() > node.second.second) {
        node.first = lists[node.second.first][node.second.second];
        minHeap.push(node);  // insert the next element in the heap
        currentMaxNumber = max(currentMaxNumber, node.first);
      }
    }

    return make_pair(rangeStart, rangeEnd);
  }
};

int main(int argc, char *argv[]) {
  vector<vector<int>> lists = {{1, 5, 8}, {4, 12}, {7, 8, 10}};
  auto result = SmallestRange::findSmallestRange(lists);
  cout << "Smallest range is: [" << result.first << ", " << result.second << "]";
}
```

Time complexity

Since, at most, we'll be going through all the elements of all the arrays and will remove/add one element in the heap in each step, the time complexity of the above algorithm will be $O(N*logM)$ where 'N' is the total number of elements in all the 'M' input arrays.

Space complexity

The space complexity will be $O(M)$ because, at any time, our min-heap will be store one number from all the 'M' input arrays.

K Pairs with Largest Sums (Hard)

Given two sorted arrays in descending order, find 'K' pairs with the largest sum where each pair consists of numbers from both the arrays.

Example 1:

Input: L1=[9, 8, 2], L2=[6, 3, 1], K=3
Output: [9, 3], [9, 6], [8, 6]
Explanation: These 3 pairs have the largest sum. No other pair has a sum larger than any of these.

Example 2:

Input: L1=[5, 2, 1], L2=[2, -1], K=3
Output: [5, 2], [5, -1], [2, 2]

Solution

This problem follows the **K-way merge** pattern and we can follow a similar approach as discussed in Merge K Sorted Lists.

We can go through all the numbers of the two input arrays to create pairs and initially insert them all in the heap until we have 'K' pairs in **Min Heap**. After that, if a pair is bigger than the top (smallest) pair in the heap, we can remove the smallest pair and insert this pair in the heap.

We can optimize our algorithms in two ways:

1. Instead of iterating over all the numbers of both arrays, we can iterate only the first 'K' numbers from both arrays. Since the arrays are sorted

in descending order, the pairs with the maximum sum will be constituted by the first 'K' numbers from both the arrays.

2. As soon as we encounter a pair with a sum that is smaller than the smallest (top) element of the heap, we don't need to process the next elements of the array. Since the arrays are sorted in descending order, we won't be able to find a pair with a higher sum moving forward.

```
using namespace std;


#include <iostream>

#include <queue>

#include <vector>


class LargestPairs {

 public:

  struct sumCompare {

    bool operator()(const pair<int, int> &x, const pair<int, int> &y) {

      return x.first + x.second > y.first + y.second;

    }

  };


  static vector<pair<int, int>> findKLargestPairs(const vector<int> &nums1,

                                                  const vector<int> &nums2, int k) {

    vector<pair<int, int>> minHeap;

    for (int i = 0; i < nums1.size() && i < k; i++) {

      for (int j = 0; j < nums2.size() && j < k; j++) {

        if (minHeap.size() < k) {

          minHeap.push_back(make_pair(nums1[i], nums2[j]));

          push_heap(minHeap.begin(), minHeap.end(), sumCompare());

        } else {
```

```
        // if the sum of the two numbers from the two arrays is smaller than the smallest (top)
        // element of the heap, we can 'break' here. Since the arrays are sorted in the descending
        // order, we'll not be able to find a pair with a higher sum moving forward.
        if (nums1[i] + nums2[j] < minHeap.front().first + minHeap.front().second) {
          break;
        } else {  // else: we have a pair with a larger sum, remove top and insert this pair in
                  // the heap
          pop_heap(minHeap.begin(), minHeap.end(), sumCompare());
          minHeap.pop_back();
          minHeap.push_back(make_pair(nums1[i], nums2[j]));
          push_heap(minHeap.begin(), minHeap.end(), sumCompare());
        }
      }
    }
  }
  return minHeap;
 }
};

int main(int argc, char *argv[]) {
 auto result = LargestPairs::findKLargestPairs({9, 8, 2}, {6, 3, 1}, 3);
 cout << "Pairs with largest sum are: ";
 for (auto pair : result) {
  cout << "[" << pair.first << ", " << pair.second << "] ";
 }
}
```

Time complexity

Since, at most, we'll be going through all the elements of both arrays and we will add/remove one element in the heap in each step, the time complexity of the above algorithm will be $O(N*M*logK)$ where 'N' and 'M' are the total number of elements in both arrays, respectively.

If we assume that both arrays have at least 'K' elements then the time complexity can be simplified to $O(K^2logK)$, because we are not iterating more than 'K' elements in both arrays.

Space complexity

The space complexity will be $O(K)$ because, at any time, our **Min Heap** will be storing 'K' largest pairs.

1. Introduction

0/1 Knapsack pattern is based on the famous problem with the same name which is efficiently solved using **Dynamic Programming (DP)**.

In this pattern, we will go through a set of problems to develop an understanding of DP. We will always start with a brute-force recursive solution to see the overlapping subproblems, i.e., realizing that we are solving the same problems repeatedly.

After the recursive solution, we will modify our algorithm to apply advanced techniques of **Memoization** and **Bottom-Up Dynamic Programming** to develop a complete understanding of this pattern.

Let's jump onto our first problem.

Introduction

Given the weights and profits of 'N' items, we are asked to put these items in a knapsack with a capacity 'C.' The goal is to get the maximum profit out of the knapsack items. Each item can only be selected once, as we don't have multiple quantities of any item.

Let's take Merry's example, who wants to carry some fruits in the knapsack to get maximum profit. Here are the weights and profits of the fruits:

Items: { Apple, Orange, Banana, Melon }
Weights: { 2, 3, 1, 4 }
Profits: { 4, 5, 3, 7 }
Knapsack capacity: 5

Let's try to put various combinations of fruits in the knapsack, such that their total weight is not more than 5:

Apple + Orange (total weight 5) => 9 profit
Apple + Banana (total weight 3) => 7 profit
Orange + Banana (total weight 4) => 8 profit
Banana + Melon (total weight 5) => 10 profit

This shows that **Banana + Melon** is the best combination as it gives us the maximum profit, and the total weight does not exceed the capacity.

Problem Statement

Given two integer arrays to represent weights and profits of 'N' items, we need to find a subset of these items which will give us maximum profit such that their cumulative weight is not more than a given number 'C.' Each item can only be selected once, which means either we put an item in the knapsack or we skip it.

Basic Solution

A basic brute-force solution could be to try all combinations of the given items (as we did above), allowing us to choose the one with maximum profit and a weight that doesn't exceed 'C'. Take the example of four items (A, B, C, and D), as shown in the diagram below. To try all the combinations, our algorithm will look like:

```
for each item 'i'
  create a new set which INCLUDES item 'i' if the total weight does not exceed the capacity, and
    recursively process the remaining capacity and items
  create a new set WITHOUT item 'i', and recursively process the remaining items
return the set from the above two sets with higher profit
```

```
using namespace std;

#include <iostream>
#include <vector>

class Knapsack {
 public:
  int solveKnapsack(const vector<int> &profits, vector<int> &weights, int capacity) {
    return this->knapsackRecursive(profits, weights, capacity, 0);
  }

 private:
  int knapsackRecursive(const vector<int> &profits, vector<int> &weights, int capacity,
                        int currentIndex) {
    // base checks
    if (capacity <= 0 || currentIndex >= profits.size()) {
      return 0;
    }

    // recursive call after choosing the element at the currentIndex
    // if the weight of the element at currentIndex exceeds the capacity, we shouldn't process this
    int profit1 = 0;
    if (weights[currentIndex] <= capacity) {
      profit1 =
          profits[currentIndex] +
          knapsackRecursive(profits, weights, capacity - weights[currentIndex], currentIndex + 1);
    }
```

```
    // recursive call after excluding the element at the currentIndex
    int profit2 = knapsackRecursive(profits, weights, capacity, currentIndex + 1);

    return max(profit1, profit2);
  }
};

int main(int argc, char *argv[]) {
  Knapsack ks;
  vector<int> profits = {1, 6, 10, 16};
  vector<int> weights = {1, 2, 3, 5};
  int maxProfit = ks.solveKnapsack(profits, weights, 7);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
  maxProfit = ks.solveKnapsack(profits, weights, 6);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
}
```

Time and Space complexity

The above algorithm's time complexity is exponential $O(2^n)$, where 'n' represents the total number of items. This can also be confirmed from the above recursion tree. As we can see, we will have a total of '31' recursive calls – calculated through $(2^n) + (2^n) - 1$, which is asymptotically equivalent to $O(2^n)$.

The space complexity is $O(n)$. This space will be used to store the recursion stack. Since the recursive algorithm works in a depth-first fashion, which means that we can't have more than 'n' recursive calls on the call stack at any time.

Overlapping Sub-problems: Let's visually draw the recursive calls to see if there are any overlapping sub-problems. As we can see, in each recursive call, profits and weights arrays remain constant, and only capacity and

currentIndex change. For simplicity, let's denote capacity with 'c' and currentIndex with 'i':

We can clearly see that **'c:4, i=3'** has been called twice. Hence we have an overlapping sub-problems pattern. We can use Memoization to solve overlapping sub-problems efficiently.

Top-down Dynamic Programming with Memoization

Memoization is when we store the results of all the previously solved sub-problems and return the results from memory if we encounter a problem that has already been solved.

Since we have two changing values (`capacity` and `currentIndex`) in our recursive function `knapsackRecursive()`, we can use a two-dimensional array to store the results of all the solved sub-problems. As mentioned above, we need to store results for every sub-array (i.e., for every possible index 'i') and every possible capacity 'c.'

```
using namespace std;


#include <iostream>

#include <vector>


class Knapsack {

 public:

  virtual int solveKnapsack(const vector<int> &profits, vector<int> &weights, int capacity) {

    vector<vector<int>> dp(profits.size(), vector<int>(capacity + 1, -1));

    return this->knapsackRecursive(dp, profits, weights, capacity, 0);

  }


 private:

  int knapsackRecursive(vector<vector<int>> &dp, const vector<int> &profits, vector<int> &weights,

                        int capacity, int currentIndex) {
```

```
    // base checks
    if (capacity <= 0 || currentIndex >= profits.size()) {
      return 0;
    }

    // if we have already solved a similar problem, return the result from memory
    if (dp[currentIndex][capacity] != -1) {
      return dp[currentIndex][capacity];
    }

    // recursive call after choosing the element at the currentIndex
    // if the weight of the element at currentIndex exceeds the capacity, we shouldn't process this
    int profit1 = 0;
    if (weights[currentIndex] <= capacity) {
      profit1 = profits[currentIndex] + knapsackRecursive(dp, profits, weights,
                                            capacity - weights[currentIndex],
                                            currentIndex + 1);
    }

    // recursive call after excluding the element at the currentIndex
    int profit2 = knapsackRecursive(dp, profits, weights, capacity, currentIndex + 1);

    dp[currentIndex][capacity] = max(profit1, profit2);
    return dp[currentIndex][capacity];
  }
};

int main(int argc, char *argv[]) {
  Knapsack ks;
```

```
  vector<int> profits = {1, 6, 10, 16};
  vector<int> weights = {1, 2, 3, 5};
  int maxProfit = ks.solveKnapsack(profits, weights, 7);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
  maxProfit = ks.solveKnapsack(profits, weights, 6);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
}
```

Time and Space complexity

Since our memoization array `dp[profits.length][capacity+1]` stores the results for all subproblems, we can conclude that we will not have more than N*C$N*C$ subproblems (where 'N' is the number of items and 'C' is the knapsack capacity). This means that our time complexity will be O(N*C)$O(N*C)$.

The above algorithm will use O(N*C)$O(N*C)$ space for the memoization array. Other than that, we will use O(N)$O(N)$ space for the recursion call-stack. So the total space complexity will be O(N*C + N)$O(N*C+N)$, which is asymptotically equivalent to O(N*C).$O(N*C)$.

Bottom-up Dynamic Programming

Let's try to populate our `dp[][]` array from the above solution by working in a bottom-up fashion. Essentially, we want to find the maximum profit for every sub-array and every possible capacity. **This means that `dp[i][c]` will represent the maximum knapsack profit for capacity 'c' calculated from the first 'i' items.**

So, for each item at index 'i' (0 <= i < items.length) and capacity 'c' (0 <= c <= capacity), we have two options:

1. Exclude the item at index 'i.' In this case, we will take whatever profit we get from the sub-array excluding this item => `dp[i-1][c]`
2. Include the item at index 'i' if its weight is not more than the capacity. In this case, we include its profit plus whatever profit we get from the

remaining capacity and from remaining items => `profit[i] + dp[i-1][c-weight[i]]`

Finally, our optimal solution will be maximum of the above two values:

```
dp[i][c] = max (dp[i-1][c], profit[i] + dp[i-1][c-weight[i]])
```

Let's draw this visually and start with our base case of zero capacity:

```
using namespace std;

#include <iostream>
#include <vector>

class Knapsack {
 public:
  int solveKnapsack(const vector<int> &profits, vector<int> &weights, int capacity) {
    // basic checks
    if (capacity <= 0 || profits.empty() || weights.size() != profits.size()) {
      return 0;
    }

    int n = profits.size();
    vector<vector<int>> dp(n, vector<int>(capacity + 1));

    // populate the capacity=0 columns, with '0' capacity we have '0' profit
    for (int i = 0; i < n; i++) {
      dp[i][0] = 0;
    }

    // if we have only one weight, we will take it if it is not more than the capacity
```

```
    for (int c = 0; c <= capacity; c++) {
      if (weights[0] <= c) {
        dp[0][c] = profits[0];
      }
    }

    // process all sub-arrays for all the capacities
    for (int i = 1; i < n; i++) {
      for (int c = 1; c <= capacity; c++) {
        int profit1 = 0, profit2 = 0;
        // include the item, if it is not more than the capacity
        if (weights[i] <= c) {
          profit1 = profits[i] + dp[i - 1][c - weights[i]];
        }
        // exclude the item
        profit2 = dp[i - 1][c];
        // take maximum
        dp[i][c] = max(profit1, profit2);
      }
    }

    // maximum profit will be at the bottom-right corner.
    return dp[n - 1][capacity];
  }
};

int main(int argc, char *argv[]) {
  Knapsack ks;
  vector<int> profits = {1, 6, 10, 16};
```

```
  vector<int> weights = {1, 2, 3, 5};
  int maxProfit = ks.solveKnapsack(profits, weights, 6);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
  maxProfit = ks.solveKnapsack(profits, weights, 7);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
}
```

Time and Space complexity

The above solution has the time and space complexity of $O(N*C)$, where 'N' represents total items, and 'C' is the maximum capacity.

How can we find the selected items?

As we know, the final profit is at the bottom-right corner. Therefore, we will start from there to find the items that will be going into the knapsack.

As you remember, at every step, we had two options: include an item or skip it. If we skip an item, we take the profit from the remaining items (i.e., from the cell right above it); if we include the item, then we jump to the remaining profit to find more items.

Let's understand this from the above example:

1. '22' did not come from the top cell (which is 17); hence we must include the item at index '3' (which is item 'D').
2. Subtract the profit of item 'D' from '22' to get the remaining profit '6'. We then jump to profit '6' on the same row.
3. '6' came from the top cell, so we jump to row '2'.
4. Again, '6' came from the top cell, so we jump to row '1'.
5. '6' is different from the top cell, so we must include this item (which is item 'B').
6. Subtract the profit of 'B' from '6' to get profit '0'. We then jump to profit '0' on the same row. As soon as we hit zero remaining profit, we can finish our item search.
7. Thus, the items going into the knapsack are {B, D}.

Let's write a function to print the set of items included in the knapsack.

```
using namespace std;

#include <iostream>
#include <vector>

class Knapsack {
 public:
  virtual int solveKnapsack(const vector<int> &profits, vector<int> &weights, int capacity) {
    // base checks
    if (capacity <= 0 || profits.empty() || weights.size() != profits.size()) {
      return 0;
    }

    int n = profits.size();
    vector<vector<int>> dp(n, vector<int>(capacity + 1));

    // populate the capacity=0 columns, with '0' capacity we have '0' profit
    for (int i = 0; i < n; i++) {
      dp[i][0] = 0;
    }

    // if we have only one weight, we will take it if it is not more than the
    // capacity
    for (int c = 0; c <= capacity; c++) {
      if (weights[0] <= c) {
        dp[0][c] = profits[0];
      }
    }
```

```
  // process all sub-arrays for all the capacities
  for (int i = 1; i < n; i++) {
    for (int c = 1; c <= capacity; c++) {
      int profit1 = 0, profit2 = 0;
      // include the item, if it is not more than the capacity
      if (weights[i] <= c) {
        profit1 = profits[i] + dp[i - 1][c - weights[i]];
      }
      // exclude the item
      profit2 = dp[i - 1][c];
      // take maximum
      dp[i][c] = max(profit1, profit2);
    }
  }

  printSelectedElements(dp, weights, profits, capacity);
  // maximum profit will be at the bottom-right corner.
  return dp[n - 1][capacity];
 }

private:
 void printSelectedElements(vector<vector<int>> &dp, const vector<int> &weights,
                            const vector<int> &profits, int capacity) {
  cout << "Selected weights:";
  int totalProfit = dp[weights.size() - 1][capacity];
  for (int i = weights.size() - 1; i > 0; i--) {
    if (totalProfit != dp[i - 1][capacity]) {
      cout << " " << weights[i];
      capacity -= weights[i];
```

```
      totalProfit -= profits[i];
     }
    }

    if (totalProfit != 0) {
     cout << " " << weights[0];
    }
    cout << "" << endl;
  }
};

int main(int argc, char *argv[]) {
  Knapsack ks;
  vector<int> profits = {1, 6, 10, 16};
  vector<int> weights = {1, 2, 3, 5};
  int maxProfit = ks.solveKnapsack(profits, weights, 7);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
  maxProfit = ks.solveKnapsack(profits, weights, 6);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
}
```

Challenge

Can we improve our bottom-up DP solution even further? Can you find an algorithm that has $O(C)$ space complexity?

```
using namespace std;

#include <iostream>
#include <vector>
```

```
// space optimization
class Knapsack {
 public:
  int solveKnapsack(const vector<int> &profits, const vector<int> &weights, int capacity) {
    // basic checks
    if (capacity <= 0 || profits.empty() || weights.size() != profits.size()) {
      return 0;
    }

    int n = profits.size();
    // we only need one previous row to find the optimal solution, overall we need '2' rows
    // the above solution is similar to the previous solution, the only difference is that
    // we use `i%2` instead if `i` and `(i-1)%2` instead if `i-1`
    vector<vector<int>> dp(2, vector<int>(capacity + 1));

    // if we have only one weight, we will take it if it is not more than the
    // capacity
    for (int c = 0; c <= capacity; c++) {
      if (weights[0] <= c) {
        dp[0][c] = dp[1][c] = profits[0];
      }
    }

    // process all sub-arrays for all the capacities
    for (int i = 1; i < n; i++) {
      for (int c = 0; c <= capacity; c++) {
        int profit1 = 0, profit2 = 0;
        // include the item, if it is not more than the capacity
```

```
      if (weights[i] <= c) {
        profit1 = profits[i] + dp[(i - 1) % 2][c - weights[i]];
      }
      // exclude the item
      profit2 = dp[(i - 1) % 2][c];
      // take maximum
      dp[i % 2][c] = max(profit1, profit2);
    }
   }

   return dp[(n - 1) % 2][capacity];
 }
};

int main(int argc, char *argv[]) {
  Knapsack ks;
  vector<int> profits = {1, 6, 10, 16};
  vector<int> weights = {1, 2, 3, 5};
  cout << ks.solveKnapsack(profits, weights, 7) << endl;
  cout << ks.solveKnapsack(profits, weights, 6) << endl;
}
```

The solution above is similar to the previous solution; the only difference is that we use `i%2` instead of `i` and `(i-1)%2` instead of `i-1`. This solution has a space complexity of $O(2*C) = O(C)$ $O(2*C)=O(C)$, where 'C' is the knapsack's maximum capacity.

This space optimization solution can also be implemented using a single array. It is a bit tricky, but the intuition is to use the same array for the previous and the next iteration!

If you see closely, we need two values from the previous iteration: `dp[c]` and `dp[c-weight[i]]`

Since our inner loop is iterating over `c:0-->capacity`, let's see how this might affect our two required values:

1. When we access `dp[c],` it has not been overridden yet for the current iteration, so it should be fine.
2. `dp[c-weight[i]]` might be overridden if "weight[i] > 0". Therefore we can't use this value for the current iteration.

To solve the second case, we can change our inner loop to process in the reverse direction: `c:capacity-->0`. This will ensure that whenever we change a value in `dp[],` we will not need it again in the current iteration.

Can you try writing this algorithm?

```
using namespace std;


#include <iostream>

#include <vector>


class Knapsack {

 public:

  int solveKnapsack(const vector<int> &profits, vector<int> &weights, int capacity) {

    // basic checks

    if (capacity <= 0 || profits.empty() || weights.size() != profits.size()) {

      return 0;

    }


    int n = profits.size();

    vector<int> dp(capacity + 1);


    // if we have only one weight, we will take it if it is not more than the

    // capacity
```

```
    for (int c = 0; c <= capacity; c++) {
      if (weights[0] <= c) {
        dp[c] = profits[0];
      }
    }

    // process all sub-arrays for all the capacities
    for (int i = 1; i < n; i++) {
      for (int c = capacity; c >= 0; c--) {
        int profit1 = 0, profit2 = 0;
        // include the item, if it is not more than the capacity
        if (weights[i] <= c) {
          profit1 = profits[i] + dp[c - weights[i]];
        }
        // exclude the item
        profit2 = dp[c];
        // take maximum
        dp[c] = max(profit1, profit2);
      }
    }

    return dp[capacity];
  }
};

int main(int argc, char *argv[]) {
  Knapsack ks;
  vector<int> profits = {1, 6, 10, 16};
  vector<int> weights = {1, 2, 3, 5};
```

```
  cout << ks.solveKnapsack(profits, weights, 7) << endl;
  cout << ks.solveKnapsack(profits, weights, 6) << endl;
}
```

Problem Statement

Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both subsets is equal.

Example 1:

Input: {1, 2, 3, 4}
Output: True
Explanation: The given set can be partitioned into two subsets with equal sum: {1, 4} & {2, 3}

Example 2:

Input: {1, 1, 3, 4, 7}
Output: True
Explanation: The given set can be partitioned into two subsets with equal sum: {1, 3, 4} & {1, 7}

Example 3:

Input: {2, 3, 4, 6}
Output: False
Explanation: The given set cannot be partitioned into two subsets with equal sum.

Basic Solution

This problem follows the **0/1 Knapsack pattern**. A basic brute-force solution could be to try all combinations of partitioning the given numbers into two sets to see if any pair of sets has an equal sum.

Assume that `s` represents the total sum of all the given numbers. Then the two equal subsets must have a sum equal to `s/2`. This essentially transforms our problem to: "Find a subset of the given numbers that has a total sum of `s/2`".

So our brute-force algorithm will look like:

```
for each number 'i'
  create a new set which INCLUDES number 'i' if it does not exceed 'S/2', and recursively
```

```
    process the remaining numbers
  create a new set WITHOUT number 'i', and recursively process the remaining items
return true if any of the above sets has a sum equal to 'S/2', otherwise return false

using namespace std;

#include <iostream>
#include <vector>

class PartitionSet {
 public:
  bool canPartition(const vector<int> &num) {
    int sum = 0;
    for (int i = 0; i < num.size(); i++) {
      sum += num[i];
    }

    // if 'sum' is a an odd number, we can't have two subsets with equal sum
    if (sum % 2 != 0) {
      return false;
    }

    return this->canPartitionRecursive(num, sum / 2, 0);
  }

 private:
  bool canPartitionRecursive(const vector<int> &num, int sum, int currentIndex) {
    // base check
    if (sum == 0) {
```

```
      return true;
    }

    if (num.empty() || currentIndex >= num.size()) {
      return false;
    }

    // recursive call after choosing the number at the currentIndex
    // if the number at currentIndex exceeds the sum, we shouldn't process this
    if (num[currentIndex] <= sum) {
      if (canPartitionRecursive(num, sum - num[currentIndex], currentIndex + 1)) {
        return true;
      }
    }

    // recursive call after excluding the number at the currentIndex
    return canPartitionRecursive(num, sum, currentIndex + 1);
  }
};

int main(int argc, char *argv[]) {
  PartitionSet ps;
  vector<int> num = {1, 2, 3, 4};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 1, 3, 4, 7};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{2, 3, 4, 6};
  cout << ps.canPartition(num) << endl;
}
```

Time and Space complexity

The time complexity of the above algorithm is exponential $O(2^n)$, where 'n' represents the total number. The space complexity is $O(n)$, which will be used to store the recursion stack.

Top-down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping sub-problems. As stated in previous lessons, memoization is when we store the results of all the previously solved sub-problems so we can return the results from memory if we encounter a problem that has already been solved.

Since we need to store the results for every subset and for every possible sum, therefore we will be using a two-dimensional array to store the results of the solved sub-problems. The first dimension of the array will represent different subsets and the second dimension will represent different 'sums' that we can calculate from each subset. These two dimensions of the array can also be inferred from the two changing values (sum and currentIndex) in our recursive function `canPartitionRecursive()`.

```
using namespace std;


#include <iostream>

#include <vector>


class PartitionSet {

 public:

  bool canPartition(const vector<int> &num) {

    int sum = 0;

    for (int i = 0; i < num.size(); i++) {

      sum += num[i];

    }
```

```
  // if 'sum' is a an odd number, we can't have two subsets with equal sum
  if (sum % 2 != 0) {
    return false;
  }

  vector<vector<int>> dp(num.size(), vector<int>(sum / 2 + 1, -1));
  return this->canPartitionRecursive(dp, num, sum / 2, 0);
}

private:
 bool canPartitionRecursive(vector<vector<int>> &dp, const vector<int> &num, int sum,
                            int currentIndex) {
  // base check
  if (sum == 0) {
    return true;
  }

  if (num.empty() || currentIndex >= num.size()) {
    return false;
  }

  // if we have not already processed a similar problem
  if (dp[currentIndex][sum] == -1) {
    // recursive call after choosing the number at the currentIndex
    // if the number at currentIndex exceeds the sum, we shouldn't process this
    if (num[currentIndex] <= sum) {
      if (canPartitionRecursive(dp, num, sum - num[currentIndex], currentIndex + 1)) {
        dp[currentIndex][sum] = 1;
        return true;
```

```
      }
    }

    // recursive call after excluding the number at the currentIndex
    dp[currentIndex][sum] = canPartitionRecursive(dp, num, sum, currentIndex + 1) ? 1 : 0;
  }

  return dp[currentIndex][sum] == 1 ? true : false;
 }
};

int main(int argc, char *argv[]) {
 PartitionSet ps;
 vector<int> num = {1, 2, 3, 4};
 cout << ps.canPartition(num) << endl;
 num = vector<int>{1, 1, 3, 4, 7};
 cout << ps.canPartition(num) << endl;
 num = vector<int>{2, 3, 4, 6};
 cout << ps.canPartition(num) << endl;
}
```

Time and Space complexity

The above algorithm has the time and space complexity of O(N*S)$O(N*S)$, where 'N' represents total numbers and 'S' is the total sum of all the numbers.

Bottom-up Dynamic Programming

Let's try to populate our `dp[][]` array from the above solution by working in a bottom-up fashion. Essentially, we want to find if we can make all possible

sums with every subset. **This means, `dp[i][s]` will be 'true' if we can make the sum 's' from the first 'i' numbers.**

So, for each number at index 'i' (0 <= i < num.length) and sum 's' (0 <= s <= S/2), we have two options:

1. Exclude the number. In this case, we will see if we can get 's' from the subset excluding this number: `dp[i-1][s]`
2. Include the number if its value is not more than 's'. In this case, we will see if we can find a subset to get the remaining sum: `dp[i-1][s-num[i]]`

If either of the two above scenarios is true, we can find a subset of numbers with a sum equal to 's'.

Let's start with our base case of zero capacity:

From the above visualization, we can clearly see that it is possible to partition the given set into two subsets with equal sums, as shown by bottom-right cell: `dp[3][5] => T`

```
using namespace std;


#include <iostream>

#include <vector>


class PartitionSet {

 public:

  bool canPartition(const vector<int> &num) {

    int n = num.size();

    // find the total sum

    int sum = 0;

    for (int i = 0; i < n; i++) {

      sum += num[i];

    }
```

```
// if 'sum' is a an odd number, we can't have two subsets with same total
if (sum % 2 != 0) {
  return false;
}

// we are trying to find a subset of given numbers that has a total sum of ‘sum/2’.
sum /= 2;

vector<vector<bool>> dp(n, vector<bool>(sum + 1));

// populate the sum=0 columns, as we can always for '0' sum with an empty set
for (int i = 0; i < n; i++) {
  dp[i][0] = true;
}

// with only one number, we can form a subset only when the required sum is
// equal to its value
for (int s = 1; s <= sum; s++) {
  dp[0][s] = (num[0] == s ? true : false);
}

// process all subsets for all sums
for (int i = 1; i < n; i++) {
  for (int s = 1; s <= sum; s++) {
    // if we can get the sum 's' without the number at index 'i'
    if (dp[i - 1][s]) {
      dp[i][s] = dp[i - 1][s];
    } else if (s >= num[i]) {  // else if we can find a subset to get the remaining sum
      dp[i][s] = dp[i - 1][s - num[i]];
```

```
      }
     }
    }

    // the bottom-right corner will have our answer.
    return dp[n - 1][sum];
  }
};

int main(int argc, char *argv[]) {
  PartitionSet ps;
  vector<int> num = {1, 2, 3, 4};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 1, 3, 4, 7};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{2, 3, 4, 6};
  cout << ps.canPartition(num) << endl;
}
```

Time and Space complexity

The above solution the has time and space complexity of $O(N*S)$, where 'N' represents total numbers and 'S' is the total sum of all the numbers.

Problem Statement

Given a set of positive numbers, determine if a subset exists whose sum is equal to a given number 'S'.

Example 1:

Input: {1, 2, 3, 7}, S=6
Output: True
The given set has a subset whose sum is '6': {1, 2, 3}

Example 2:

Input: {1, 2, 7, 1, 5}, S=10
Output: True
The given set has a subset whose sum is '10': {1, 2, 7}

Example 3:

Input: {1, 3, 4, 8}, S=6
Output: False
The given set does not have any subset whose sum is equal to '6'.

Basic Solution

This problem follows the **0/1 Knapsack pattern** and is quite similar to Equal Subset Sum Partition. A basic brute-force solution could be to try all subsets of the given numbers to see if any set has a sum equal to 'S'.

So our brute-force algorithm will look like:

```
for each number 'i'
  create a new set which INCLUDES number 'i' if it does not exceed 'S', and recursively
    process the remaining numbers
  create a new set WITHOUT number 'i', and recursively process the remaining numbers
return true if any of the above two sets has a sum equal to 'S', otherwise return false
```

Since this problem is quite similar to Equal Subset Sum Partition, let's jump directly to the bottom-up dynamic programming solution.

Bottom-up Dynamic Programming

We'll try to find if we can make all possible sums with every subset to populate the array `dp[TotalNumbers][S+1]`.

For every possible sum ‘s’ (where 0 <= s <= S), we have two options:

1. Exclude the number. In this case, we will see if we can get the sum ‘s’ from the subset excluding this number => `dp[index-1][s]`
2. Include the number if its value is not more than ‘s’. In this case, we will see if we can find a subset to get the remaining sum => `dp[index-1][s-num[index]]`

If either of the above two scenarios returns true, we can find a subset with a sum equal to ‘s’.

Let’s draw this visually, with the example input {1, 2, 3, 7}, and start with our base case of size zero:

```
using namespace std;


#include <iostream>

#include <vector>


class SubsetSum {

 public:

  virtual bool canPartition(const vector<int> &num, int sum) {

    int n = num.size();

    vector<vector<bool>> dp(n, vector<bool>(sum + 1));


    // populate the sum=0 columns, as we can always form '0' sum with an empty set

    for (int i = 0; i < n; i++) {

      dp[i][0] = true;

    }


    // with only one number, we can form a subset only when the required sum is equal to its value

    for (int s = 1; s <= sum; s++) {

      dp[0][s] = (num[0] == s ? true : false);
```

```
    }

    // process all subsets for all sums
    for (int i = 1; i < num.size(); i++) {
      for (int s = 1; s <= sum; s++) {
        // if we can get the sum 's' without the number at index 'i'
        if (dp[i - 1][s]) {
          dp[i][s] = dp[i - 1][s];
        } else if (s >= num[i]) {
          // else include the number and see if we can find a subset to get the remaining sum
          dp[i][s] = dp[i - 1][s - num[i]];
        }
      }
    }

    // the bottom-right corner will have our answer.
    return dp[num.size() - 1][sum];
  }
};

int main(int argc, char *argv[]) {
  SubsetSum ss;
  vector<int> num = {1, 2, 3, 7};
  cout << ss.canPartition(num, 6) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ss.canPartition(num, 10) << endl;
  num = vector<int>{1, 3, 4, 8};
  cout << ss.canPartition(num, 6) << endl;
}
```

Time and Space complexity

The above solution has the time and space complexity of $O(N*S)$, where 'N' represents total numbers and 'S' is the required sum.

Challenge

Can we improve our bottom-up DP solution even further? Can you find an algorithm that has $O(S)$ space complexity?

```
using namespace std;


#include <iostream>

#include <vector>


class SubsetSum {

public:

 bool canPartition(const vector<int> &num, int sum) {

  int n = num.size();

  vector<bool> dp(sum + 1);


  // handle sum=0, as we can always have '0' sum with an empty set

  dp[0] = true;


  // with only one number, we can have a subset only when the required sum is equal to its value

  for (int s = 1; s <= sum; s++) {

   dp[s] = (num[0] == s ? true : false);

  }


  // process all subsets for all sums
```

```
    for (int i = 1; i < n; i++) {
      for (int s = sum; s >= 0; s--) {
        // if dp[s]==true, this means we can get the sum 's' without num[i], hence we can move on to
        // the next number else we can include num[i] and see if we can find a subset to get the
        // remaining sum
        if (!dp[s] && s >= num[i]) {
          dp[s] = dp[s - num[i]];
        }
      }
    }

    return dp[sum];
  }
};

int main(int argc, char *argv[]) {
  SubsetSum ss;
  vector<int> num = {1, 2, 3, 7};
  cout << ss.canPartition(num, 6) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ss.canPartition(num, 10) << endl;
  num = vector<int>{1, 3, 4, 8};
  cout << ss.canPartition(num, 6) << endl;
}
```

Problem Statement

Given a set of positive numbers, partition the set into two subsets with minimum difference between their subset sums.

Example 1:

Input: {1, 2, 3, 9}
Output: 3
Explanation: We can partition the given set into two subsets where minimum absolute difference between the sum of numbers is '3'. Following are the two subsets: {1, 2, 3} & {9}.

Example 2:

Input: {1, 2, 7, 1, 5}
Output: 0
Explanation: We can partition the given set into two subsets where minimum absolute difference between the sum of number is '0'. Following are the two subsets: {1, 2, 5} & {7, 1}.

Example 3:

Input: {1, 3, 100, 4}
Output: 92
Explanation: We can partition the given set into two subsets where minimum absolute difference between the sum of numbers is '92'. Here are the two subsets: {1, 3, 4} & {100}.

Basic Solution

This problem follows the **0/1 Knapsack pattern** and can be converted into a Subset Sum problem.

Let's assume S1 and S2 are the two desired subsets. A basic brute-force solution could be to try adding each element either in S1 or S2 in order to find the combination that gives the minimum sum difference between the two sets.

So our brute-force algorithm will look like:

```
for each number 'i'
  add number 'i' to S1 and recursively process the remaining numbers
  add number 'i' to S2 and recursively process the remaining numbers
return the minimum absolute difference of the above two sets
```

```
using namespace std;

#include <iostream>
#include <vector>

class PartitionSet {
 public:
  int canPartition(const vector<int> &num) { return this->canPartitionRecursive(num, 0, 0, 0); }

 private:
  int canPartitionRecursive(const vector<int> &num, int currentIndex, int sum1, int sum2) {
    // base check
    if (currentIndex == num.size()) {
      return abs(sum1 - sum2);
    }

    // recursive call after including the number at the currentIndex in the first set
    int diff1 = canPartitionRecursive(num, currentIndex + 1, sum1 + num[currentIndex], sum2);

    // recursive call after including the number at the currentIndex in the second set
    int diff2 = canPartitionRecursive(num, currentIndex + 1, sum1, sum2 + num[currentIndex]);

    return min(diff1, diff2);
  }
};

int main(int argc, char *argv[]) {
```

```
  PartitionSet ps;
  vector<int> num = {1, 2, 3, 9};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 3, 100, 4};
  cout << ps.canPartition(num) << endl;
}
```

Time and Space complexity

Because of the two recursive calls, the time complexity of the above algorithm is exponential $O(2^n)$, where 'n' represents the total number. The space complexity is $O(n)$ which is used to store the recursion stack.

Top-down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping sub-problems.

We will be using a two-dimensional array to store the results of the solved sub-problems. We can uniquely identify a sub-problem from 'currentIndex' and 'Sum1' as 'Sum2' will always be the sum of the remaining numbers.

```
using namespace std;


#include <iostream>
#include <vector>


class PartitionSet {
 public:
  int canPartition(const vector<int> &num) {
    int sum = 0;
    for (int i = 0; i < num.size(); i++) {
      sum += num[i];
```

```
    }

    vector<vector<int>> dp(num.size(), vector<int>(sum + 1, -1));
    return this->canPartitionRecursive(dp, num, 0, 0, 0);
  }

 private:
  int canPartitionRecursive(vector<vector<int>> &dp, const vector<int> &num, int currentIndex,
                            int sum1, int sum2) {
    // base check
    if (currentIndex == num.size()) {
      return abs(sum1 - sum2);
    }

    // check if we have not already processed similar problem
    if (dp[currentIndex][sum1] == -1) {
      // recursive call after including the number at the currentIndex in the first set
      int diff1 = canPartitionRecursive(dp, num, currentIndex + 1, sum1 + num[currentIndex], sum2);

      // recursive call after including the number at the currentIndex in the second set
      int diff2 = canPartitionRecursive(dp, num, currentIndex + 1, sum1, sum2 + num[currentIndex]);

      dp[currentIndex][sum1] = min(diff1, diff2);
    }

    return dp[currentIndex][sum1];
  }
};
```

```
int main(int argc, char *argv[]) {
  PartitionSet ps;
  vector<int> num = {1, 2, 3, 9};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 3, 100, 4};
  cout << ps.canPartition(num) << endl;
}
```

Bottom-up Dynamic Programming

Let's assume 'S' represents the total sum of all the numbers. So, in this problem, we are trying to find a subset whose sum is as close to 'S/2' as possible, because if we can partition the given set into two subsets of an equal sum, we get the minimum difference, i.e. zero. This transforms our problem to Subset Sum, where we try to find a subset whose sum is equal to a given number-- 'S/2' in our case. If we can't find such a subset, then we will take the subset which has the sum closest to 'S/2'. This is easily possible, as we will be calculating all possible sums with every subset.

Essentially, we need to calculate all the possible sums up to 'S/2' for all numbers. So how can we populate the array `db[TotalNumbers][S/2+1]` in the bottom-up fashion?

For every possible sum 's' (where 0 <= s <= S/2), we have two options:

1. Exclude the number. In this case, we will see if we can get the sum 's' from the subset excluding this number => `dp[index-1][s]`
2. Include the number if its value is not more than 's'. In this case, we will see if we can find a subset to get the remaining sum => `dp[index-1][s-num[index]]`

If either of the two above scenarios is true, we can find a subset with a sum equal to 's'. We should dig into this before we can learn how to find the closest subset.

Let's draw this visually, with the example input {1, 2, 3, 9}. Since the total sum is '15', we will try to find a subset whose sum is equal to the half of it, i.e. '7'.

The above visualization tells us that it is not possible to find a subset whose sum is equal to ‘7’. So what is the closest subset we can find? We can find the subset if we start moving backwards in the last row from the bottom right corner to find the first ‘T’. The first “T” in the diagram above is the sum ‘6’, which means that we can find a subset whose sum is equal to ‘6’. This means the other set will have a sum of ‘9’ and the minimum difference will be ‘3’.

```
using namespace std;


#include <iostream>
#include <vector>


class PartitionSet {
 public:
  virtual int canPartition(const vector<int> &num) {
    int sum = 0;
    for (int i = 0; i < num.size(); i++) {
      sum += num[i];
    }

    int n = num.size();
    vector<vector<bool>> dp(n, vector<bool>(sum / 2 + 1, false));

    // populate the sum=0 columns, as we can always form '0' sum with an empty set
    for (int i = 0; i < n; i++) {
      dp[i][0] = true;
    }

    // with only one number, we can form a subset only when the sum is equal to that number
    for (int s = 0; s <= sum / 2; s++) {
```

```
    dp[0][s] = (num[0] == s ? true : false);
  }

  // process all subsets for all sums
  for (int i = 1; i < num.size(); i++) {
    for (int s = 1; s <= sum / 2; s++) {
      // if we can get the sum 's' without the number at index 'i'
      if (dp[i - 1][s]) {
        dp[i][s] = dp[i - 1][s];
      } else if (s >= num[i]) {
        // else include the number and see if we can find a subset to get the remaining sum
        dp[i][s] = dp[i - 1][s - num[i]];
      }
    }
  }

  int sum1 = 0;
  // Find the largest index in the last row which is true
  for (int i = sum / 2; i >= 0; i--) {
    if (dp[n - 1][i] == true) {
      sum1 = i;
      break;
    }
  }

  int sum2 = sum - sum1;
  return abs(sum2 - sum1);
 }
};
```

```
int main(int argc, char *argv[]) {
  PartitionSet ps;
  vector<int> num = {1, 2, 3, 9};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 3, 100, 4};
  cout << ps.canPartition(num) << endl;
}
```

Time and Space complexity

The above solution has the time and space complexity of $O(N*S)$, where 'N' represents total numbers and 'S' is the total sum of all the numbers.

Count of Subset Sum (hard)

Given a set of positive numbers, find the total number of subsets whose sum is equal to a given number 'S'.

Example 1:

Input: {1, 1, 2, 3}, S=4
Output: 3
The given set has '3' subsets whose sum is '4': {1, 1, 2}, {1, 3}, {1, 3}
Note that we have two similar sets {1, 3}, because we have two '1' in our input.

Example 2:

Input: {1, 2, 7, 1, 5}, S=9
Output: 3
The given set has '3' subsets whose sum is '9': {2, 7}, {1, 7, 1}, {1, 2, 1, 5}

Count of Subset Sum (hard)

Given a set of positive numbers, find the total number of subsets whose sum is equal to a given number ‘S’.

Example 1:

Input: {1, 1, 2, 3}, S=4
Output: 3
The given set has '3' subsets whose sum is '4': {1, 1, 2}, {1, 3}, {1, 3}
Note that we have two similar sets {1, 3}, because we have two '1' in our input.

Example 2:

Input: {1, 2, 7, 1, 5}, S=9
Output: 3
The given set has '3' subsets whose sum is '9': {2, 7}, {1, 7, 1}, {1, 2, 1, 5}

Basic Solution

This problem follows the **0/1 Knapsack pattern** and is quite similar to Subset Sum. The only difference in this problem is that we need to count the number of subsets, whereas in Subset Sum we only wanted to know if a subset with the given sum existed.

A basic brute-force solution could be to try all subsets of the given numbers to count the subsets that have a sum equal to ‘S’. So our brute-force algorithm will look like:

```
for each number 'i'
  create a new set which includes number 'i' if it does not exceed 'S', and recursively
     process the remaining numbers and sum
  create a new set without number 'i', and recursively process the remaining numbers
return the count of subsets who has a sum equal to 'S'
```

```
using namespace std;

#include <iostream>
#include <vector>
```

```
// divide and conquer

class SubsetSum {
 public:
  virtual int countSubsets(const vector<int> &num, int sum) {
    return this->countSubsetsRecursive(num, sum, 0);
  }

 private:
  int countSubsetsRecursive(const vector<int> &num, int sum, int currentIndex) {
    // base checks
    if (sum == 0) {
      return 1;
    }

    if (num.empty() || currentIndex >= num.size()) {
      return 0;
    }

    // recursive call after selecting the number at the currentIndex
    // if the number at currentIndex exceeds the sum, we shouldn't process this
    int sum1 = 0;
    if (num[currentIndex] <= sum) {
      sum1 = countSubsetsRecursive(num, sum - num[currentIndex], currentIndex + 1);
    }

    // recursive call after excluding the number at the currentIndex
    int sum2 = countSubsetsRecursive(num, sum, currentIndex + 1);
```

```
    return sum1 + sum2;
  }
};

int main(int argc, char *argv[]) {
  SubsetSum ss;
  vector<int> num = {1, 1, 2, 3};
  cout << ss.countSubsets(num, 4) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ss.countSubsets(num, 9) << endl;
}
```

Time and Space complexity

The time complexity of the above algorithm is exponential $O(2^n)$, where 'n' represents the total number. The space complexity is $O(n)$, this memory is used to store the recursion stack.

Top-down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping sub-problems. We will be using a two-dimensional array to store the results of solved sub-problems. As mentioned above, we need to store results for every subset and for every possible sum.

```
using namespace std;

#include <iostream>
#include <vector>

class SubsetSum {
```

```
public:
 virtual int countSubsets(const vector<int> &num, int sum) {
  vector<vector<int>> dp(num.size(), vector<int>(sum + 1, -1));
  return this->countSubsetsRecursive(dp, num, sum, 0);
 }

private:
 int countSubsetsRecursive(vector<vector<int>> &dp, const vector<int> &num, int sum,
                           int currentIndex) {
  // base checks
  if (sum == 0) {
   return 1;
  }

  if (num.empty() || currentIndex >= num.size()) {
   return 0;
  }

  // check if we have not already processed a similar problem
  if (dp[currentIndex][sum] == -1) {
   // recursive call after choosing the number at the currentIndex
   // if the number at currentIndex exceeds the sum, we shouldn't process this
   int sum1 = 0;
   if (num[currentIndex] <= sum) {
    sum1 = countSubsetsRecursive(dp, num, sum - num[currentIndex], currentIndex + 1);
   }

   // recursive call after excluding the number at the currentIndex
   int sum2 = countSubsetsRecursive(dp, num, sum, currentIndex + 1);
```

```
      dp[currentIndex][sum] = sum1 + sum2;
    }

    return dp[currentIndex][sum];
  }
};

int main(int argc, char *argv[]) {
  SubsetSum ss;
  vector<int> num = {1, 1, 2, 3};
  cout << ss.countSubsets(num, 4) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ss.countSubsets(num, 9) << endl;
}
```

Bottom-up Dynamic Programming

We will try to find if we can make all possible sums with every subset to populate the array `db[TotalNumbers][S+1]`.

So, at every step we have two options:

1. Exclude the number. Count all the subsets without the given number up to the given sum => `dp[index-1][sum]`
2. Include the number if its value is not more than the 'sum'. In this case, we will count all the subsets to get the remaining sum => `dp[index-1][sum-num[index]]`

To find the total sets, we will add both of the above two values:

```
dp[index][sum] = dp[index-1][sum] + dp[index-1][sum-num[index]])
```

Let's start with our base case of size zero:

```
using namespace std;

#include <iostream>
#include <vector>

class SubsetSum {
 public:
  virtual int countSubsets(const vector<int> &num, int sum) {
    int n = num.size();
    vector<vector<int>> dp(n, vector<int>(sum + 1, 0));

    // populate the sum=0 columns, as we will always have an empty set for zero sum
    for (int i = 0; i < n; i++) {
      dp[i][0] = 1;
    }

    // with only one number, we can form a subset only when the required sum is
    // equal to its value
    for (int s = 1; s <= sum; s++) {
      dp[0][s] = (num[0] == s ? 1 : 0);
    }

    // process all subsets for all sums
    for (int i = 1; i < num.size(); i++) {
      for (int s = 1; s <= sum; s++) {
        // exclude the number
        dp[i][s] = dp[i - 1][s];
        // include the number, if it does not exceed the sum
        if (s >= num[i]) {
```

```
        dp[i][s] += dp[i - 1][s - num[i]];
      }
     }
    }

    // the bottom-right corner will have our answer.
    return dp[num.size() - 1][sum];
  }
};

int main(int argc, char *argv[]) {
  SubsetSum ss;
  vector<int> num = {1, 1, 2, 3};
  cout << ss.countSubsets(num, 4) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ss.countSubsets(num, 9) << endl;
}
```

Time and Space complexity

The above solution has the time and space complexity of $O(N*S)$, where 'N' represents total numbers and 'S' is the desired sum.

Challenge

Can we improve our bottom-up DP solution even further? Can you find an algorithm that has $O(S)$ space complexity?

```
using namespace std;

#include <iostream>
```

```
#include <vector>

class SubsetSum {
 public:
  static int countSubsets(const vector<int> &num, int sum) {
    int n = num.size();
    vector<int> dp(sum + 1);
    dp[0] = 1;

    // with only one number, we can form a subset only when the required sum is
    // equal to its value
    for (int s = 1; s <= sum; s++) {
      dp[s] = (num[0] == s ? 1 : 0);
    }

    // process all subsets for all sums
    for (int i = 1; i < num.size(); i++) {
      for (int s = sum; s >= 0; s--) {
        if (s >= num[i]) {
          dp[s] += dp[s - num[i]];
        }
      }
    }

    return dp[sum];
  }
};

int main(int argc, char *argv[]) {
```

```
  SubsetSum ss;
  vector<int> num = {1, 1, 2, 3};
  cout << ss.countSubsets(num, 4) << endl;
  num = vector<int>{1, 2, 7, 1, 5};
  cout << ss.countSubsets(num, 9) << endl;
}
```

Target Sum (hard)

You are given a set of positive numbers and a target sum ‘S’. Each number should be assigned either a ‘+’ or ‘-’ sign. We need to find the total ways to assign symbols to make the sum of the numbers equal to the target ‘S’.

Example 1:

Input: {1, 1, 2, 3}, S=1
Output: 3
Explanation: The given set has '3' ways to make a sum of '1': {+1-1-2+3} & {-1+1-2+3} & {+1+1+2-3}

Example 2:

Input: {1, 2, 7, 1}, S=9
Output: 2
Explanation: The given set has '2' ways to make a sum of '9': {+1+2+7-1} & {-1+2+7+1}

Solution

This problem follows the **0/1 Knapsack pattern** and can be converted into Count of Subset Sum. Let’s dig into this.

We are asked to find two subsets of the given numbers whose difference is equal to the given target ‘S’. Take the first example above. As we saw, one solution is {+1-1-2+3}. So, the two subsets we are asked to find are {1, 3} & {1, 2} because,

```
(1 + 3) - (1 + 2 ) = 1
```

Now, let's say 'Sum(s1)' denotes the total sum of set 's1', and 'Sum(s2)' denotes the total sum of set 's2'. So the required equation is:

```
Sum(s1) - Sum(s2) = S
```

This equation can be reduced to the subset sum problem. Let's assume that 'Sum(num)' denotes the total sum of all the numbers, therefore:

```
Sum(s1) + Sum(s2) = Sum(num)
```

Let's add the above two equations:

```
=> Sum(s1) - Sum(s2) + Sum(s1) + Sum(s2) = S + Sum(num)
=> 2 * Sum(s1) =  S + Sum(num)
=> Sum(s1) = (S + Sum(num)) / 2
```

Which means that one of the set 's1' has a sum equal to `(S + Sum(num)) / 2`. This essentially converts our problem to: "Find the count of subsets of the given numbers whose sum is equal to `(S + Sum(num)) / 2`"

```
using namespace std;


#include <iostream>

#include <vector>


class TargetSum {

 public:

  int findTargetSubsets(const vector<int> &num, int s) {

    int totalSum = 0;

    for (auto n : num) {

      totalSum += n;

    }


    // if 's + totalSum' is odd, we can't find a subset with sum equal to '(s + totalSum) / 2'

    if (totalSum < s || (s + totalSum) % 2 == 1) {

      return 0;

    }
```

```
  return countSubsets(num, (s + totalSum) / 2);
 }

 // this function is exactly similar to what we have in 'Count of Subset Sum' problem.
private:
 int countSubsets(const vector<int> &num, int sum) {
  int n = num.size();
  vector<vector<int>> dp(n, vector<int>(sum + 1, 0));

  // populate the sum=0 columns, as we will always have an empty set for zero sum
  for (int i = 0; i < n; i++) {
   dp[i][0] = 1;
  }

  // with only one number, we can form a subset only when the required sum is
  // equal to the number
  for (int s = 1; s <= sum; s++) {
   dp[0][s] = (num[0] == s ? 1 : 0);
  }

  // process all subsets for all sums
  for (int i = 1; i < num.size(); i++) {
   for (int s = 1; s <= sum; s++) {
    dp[i][s] = dp[i - 1][s];
    if (s >= num[i]) {
     dp[i][s] += dp[i - 1][s - num[i]];
    }
   }
```

```
    }

    // the bottom-right corner will have our answer.
    return dp[num.size() - 1][sum];
  }
};

int main(int argc, char *argv[]) {
  TargetSum ts;
  vector<int> num = {1, 1, 2, 3};
  cout << ts.findTargetSubsets(num, 1) << endl;
  num = vector<int>{1, 2, 7, 1};
  cout << ts.findTargetSubsets(num, 9) << endl;
}
```

Time and Space complexity

The above solution has time and space complexity of $O(N*S)$, where 'N' represents total numbers and 'S' is the desired sum.

We can further improve the solution to use only $O(S)$ space.

Space Optimized Solution

Here is the code for the space-optimized solution, using only a single array:

```
using namespace std;

#include <iostream>
#include <vector>

class TargetSum {
```

```
public:
 int findTargetSubsets(const vector<int> &num, int s) {
  int totalSum = 0;
  for (auto n : num) {
   totalSum += n;
  }

  // if 's + totalSum' is odd, we can't find a subset with sum equal to '(s + totalSum) / 2'
  if (totalSum < s || (s + totalSum) % 2 == 1) {
   return 0;
  }

  return countSubsets(num, (s + totalSum) / 2);
 }

 // this function is exactly similar to what we have in 'Count of Subset Sum' problem.
private:
 int countSubsets(const vector<int> &num, int sum) {
  int n = num.size();
  vector<int> dp(sum + 1);
  dp[0] = 1;

  // with only one number, we can form a subset only when the required sum is
  // equal to the number
  for (int s = 1; s <= sum; s++) {
   dp[s] = (num[0] == s ? 1 : 0);
  }

  // process all subsets for all sums
```

```
    for (int i = 1; i < num.size(); i++) {
      for (int s = sum; s >= 0; s--) {
        if (s >= num[i]) {
          dp[s] += dp[s - num[i]];
        }
      }
    }

    return dp[sum];
  }
};

int main(int argc, char *argv[]) {
  TargetSum ts;
  vector<int> num = {1, 1, 2, 3};
  cout << ts.findTargetSubsets(num, 1) << endl;
  num = vector<int>{1, 2, 7, 1};
  cout << ts.findTargetSubsets(num, 9) << endl;
}
```

1. Introduction

Topological Sort is used to find a linear ordering of elements that have dependencies on each other. For example, if event ‘B’ is dependent on event ‘A’, ‘A’ comes before ‘B’ in topological ordering.

This pattern defines an easy way to understand the technique for performing topological sorting of a set of elements and then solves a few problems using it.

Let’s see this pattern in action.

Problem Statement

Topological Sort of a directed graph (a graph with unidirectional edges) is a linear ordering of its vertices such that for every directed edge (U, V) from vertex `u` to vertex `v`, `u` comes before `v` in the ordering.

Given a directed graph, find the topological ordering of its vertices.

Example 1:

Input: Vertices=4, Edges=[3, 2], [3, 0], [2, 0], [2, 1]
Output: Following are the two valid topological sorts for the given graph:
1) 3, 2, 0, 1
2) 3, 2, 1, 0

Example 2:

Input: Vertices=5, Edges=[4, 2], [4, 3], [2, 0], [2, 1], [3, 1]
Output: Following are all valid topological sorts for the given graph:
1) 4, 2, 3, 0, 1
2) 4, 3, 2, 0, 1
3) 4, 3, 2, 1, 0
4) 4, 2, 3, 1, 0
5) 4, 2, 0, 3, 1

Example 3:

Input: Vertices=7, Edges=[6, 4], [6, 2], [5, 3], [5, 4], [3, 0], [3, 1], [3, 2], [4, 1]
Output: Following are all valid topological sorts for the given graph:
1) 5, 6, 3, 4, 0, 1, 2
2) 6, 5, 3, 4, 0, 1, 2
3) 5, 6, 4, 3, 0, 2, 1
4) 6, 5, 4, 3, 0, 1, 2
5) 5, 6, 3, 4, 0, 2, 1
6) 5, 6, 3, 4, 1, 2, 0

There are other valid topological ordering of the graph too.

Solution

The basic idea behind the topological sort is to provide a partial ordering among the vertices of the graph such that if there is an edge from `u` to `v` then

U≤V i.e., `u` comes before `v` in the ordering. Here are a few fundamental concepts related to topological sort:

1. **Source:** Any node that has no incoming edge and has only outgoing edges is called a source.

2. **Sink:** Any node that has only incoming edges and no outgoing edge is called a sink.

3. So, we can say that a topological ordering starts with one of the sources and ends at one of the sinks.

4. A topological ordering is possible only when the graph has no directed cycles, i.e. if the graph is a **Directed Acyclic Graph (DAG)**. If the graph has a cycle, some vertices will have cyclic dependencies which makes it impossible to find a linear ordering among vertices.

To find the topological sort of a graph we can traverse the graph in a **Breadth First Search (BFS)** way. We will start with all the sources, and in a stepwise fashion, save all sources to a sorted list. We will then remove all sources and their edges from the graph. After the removal of the edges, we will have new sources, so we will repeat the above process until all vertices are visited.

Here is the visual representation of this algorithm for Example-3:

This is how we can implement this algorithm:

a. Initialization

1. We will store the graph in **Adjacency Lists**, which means each parent vertex will have a list containing all of its children. We will do this using a **HashMap** where the 'key' will be the parent vertex number and the value will be a **List** containing children vertices.
2. To find the sources, we will keep a **HashMap** to count the in-degrees i.e., count of incoming edges of each vertex. Any vertex with '0' in-degree will be a source.

b. Build the graph and find in-degrees of all vertices

1. We will build the graph from the input and populate the in-degrees **HashMap**.

c. Find all sources

1. All vertices with ‘0’ in-degrees will be our sources and we will store them in a **Queue**.

d. Sort

1. For each source, do the following things:
    - Add it to the sorted list.
    - Get all of its children from the graph.
    - Decrement the in-degree of each child by 1.
    - If a child’s in-degree becomes ‘0’, add it to the sources **Queue**.
2. Repeat step 1, until the source **Queue** is empty.

```
using namespace std;


#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>


class TopologicalSort {
 public:
  static vector<int> sort(int vertices, const vector<vector<int>>& edges) {
    vector<int> sortedOrder;
    if (vertices <= 0) {
      return sortedOrder;
    }

    // a. Initialize the graph
    unordered_map<int, int> inDegree;       // count of incoming edges for every vertex
    unordered_map<int, vector<int>> graph;  // adjacency list graph
    for (int i = 0; i < vertices; i++) {
```

```
  inDegree[i] = 0;
  graph[i] = vector<int>();
}

// b. Build the graph
for (int i = 0; i < edges.size(); i++) {
  int parent = edges[i][0], child = edges[i][1];
  graph[parent].push_back(child);  // put the child into it's parent's list
  inDegree[child]++;               // increment child's inDegree
}

// c. Find all sources i.e., all vertices with 0 in-degrees
queue<int> sources;
for (auto entry : inDegree) {
  if (entry.second == 0) {
    sources.push(entry.first);
  }
}

// d. For each source, add it to the sortedOrder and subtract one from all of its children's
// in-degrees if a child's in-degree becomes zero, add it to the sources queue
while (!sources.empty()) {
  int vertex = sources.front();
  sources.pop();
  sortedOrder.push_back(vertex);
  vector<int> children =
      graph[vertex];  // get the node's children to decrement their in-degrees
  for (auto child : children) {
    inDegree[child]--;
```

```
      if (inDegree[child] == 0) {
        sources.push(child);
      }
    }
  }

  if (sortedOrder.size() !=
      vertices) {  // topological sort is not possible as the graph has a cycle
    return vector<int>();
  }

  return sortedOrder;
 }
};

int main(int argc, char* argv[]) {
 vector<int> result =
    TopologicalSort::sort(4, vector<vector<int>>{vector<int>{3, 2}, vector<int>{3, 0},
                                        vector<int>{2, 0}, vector<int>{2, 1}});
 for (auto num : result) {
  cout << num << " ";
 }
 cout << endl;

 result = TopologicalSort::sort(
    5, vector<vector<int>>{vector<int>{4, 2}, vector<int>{4, 3}, vector<int>{2, 0},
                           vector<int>{2, 1}, vector<int>{3, 1}});
 for (auto num : result) {
  cout << num << " ";
```

```
  }
  cout << endl;

  result = TopologicalSort::sort(
      7, vector<vector<int>>{vector<int>{6, 4}, vector<int>{6, 2}, vector<int>{5, 3},
                             vector<int>{5, 4}, vector<int>{3, 0}, vector<int>{3, 1},
                             vector<int>{3, 2}, vector<int>{4, 1}});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time Complexity

In step ‘d’, each vertex will become a source only once and each edge will be accessed and removed once. Therefore, the time complexity of the above algorithm will be $O(V+E)$, where ‘V’ is the total number of vertices and ‘E’ is the total number of edges in the graph.

Space Complexity

The space complexity will be $O(V+E)$, since we are storing all of the edges for each vertex in an adjacency list.

Similar Problems

Problem 1: Find if a given **Directed Graph** has a cycle in it or not.

Solution: If we can’t determine the topological ordering of all the vertices of a directed graph, the graph has a cycle in it. This was also referred to in the above code:

```
  if (sortedOrder.size() != vertices) // topological sort is not possible as the graph has a cycle
    return new ArrayList<>();
```

Problem Statement

There are ‘N’ tasks, labeled from ‘0’ to ‘N-1’. Each task can have some prerequisite tasks which need to be completed before it can be scheduled. Given the number of tasks and a list of prerequisite pairs, find out if it is possible to schedule all the tasks.

Example 1:

Input: Tasks=3, Prerequisites=[0, 1], [1, 2]
Output: true
Explanation: To execute task '1', task '0' needs to finish first. Similarly, task '1' needs to finish before '2' can be scheduled. A possible sceduling of tasks is: [0, 1, 2]

Example 2:

Input: Tasks=3, Prerequisites=[0, 1], [1, 2], [2, 0]
Output: false
Explanation: The tasks have cyclic dependency, therefore they cannot be sceduled.

Example 3:

Input: Tasks=6, Prerequisites=[2, 5], [0, 5], [0, 4], [1, 4], [3, 2], [1, 3]
Output: true
Explanation: A possible sceduling of tasks is: [0 1 4 3 2 5]

Solution

This problem is asking us to find out if it is possible to find a topological ordering of the given tasks. The tasks are equivalent to the vertices and the prerequisites are the edges.

We can use a similar algorithm as described in Topological Sort to find the topological ordering of the tasks. If the ordering does not include all the tasks, we will conclude that some tasks have cyclic dependencies.

```
using namespace std;

#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>

class TaskScheduling {
 public:
  static bool isSchedulingPossible(int tasks, const vector<vector<int>>& prerequisites) {
    vector<int> sortedOrder;
    if (tasks <= 0) {
      return false;
    }

    // a. Initialize the graph
    unordered_map<int, int> inDegree;       // count of incoming edges for every vertex
    unordered_map<int, vector<int>> graph;  // adjacency list graph
    for (int i = 0; i < tasks; i++) {
      inDegree[i] = 0;
      graph[i] = vector<int>();
    }

    // b. Build the graph
    for (int i = 0; i < prerequisites.size(); i++) {
      int parent = prerequisites[i][0], child = prerequisites[i][1];
      graph[parent].push_back(child);  // put the child into it's parent's list
      inDegree[child]++;               // increment child's inDegree
    }
```

```
  // c. Find all sources i.e., all vertices with 0 in-degrees
  queue<int> sources;
  for (auto entry : inDegree) {
    if (entry.second == 0) {
      sources.push(entry.first);
    }
  }

  // d. For each source, add it to the sortedOrder and subtract one from all of its children's
  // in-degrees if a child's in-degree becomes zero, add it to the sources queue
  while (!sources.empty()) {
    int vertex = sources.front();
    sources.pop();
    sortedOrder.push_back(vertex);
    vector<int> children =
        graph[vertex];  // get the node's children to decrement their in-degrees
    for (auto child : children) {
      inDegree[child]--;
      if (inDegree[child] == 0) {
        sources.push(child);
      }
    }
  }

  // if sortedOrder doesn't contain all tasks, there is a cyclic dependency between tasks,
  // therefore, we will not be able to schedule all tasks
  return sortedOrder.size() == tasks;
}
```

```
};


int main(int argc, char* argv[]) {
  bool result = TaskScheduling::isSchedulingPossible(
      3, vector<vector<int>>{vector<int>{0, 1}, vector<int>{1, 2}});
  cout << "Tasks execution possible: " << result << endl;

  result = TaskScheduling::isSchedulingPossible(
      3, vector<vector<int>>{vector<int>{0, 1}, vector<int>{1, 2}, vector<int>{2, 0}});
  cout << "Tasks execution possible: " << result << endl;

  result = TaskScheduling::isSchedulingPossible(
      6, vector<vector<int>>{vector<int>{2, 5}, vector<int>{0, 5}, vector<int>{0, 4},
                             vector<int>{1, 4}, vector<int>{3, 2}, vector<int>{1, 3}});
  cout << "Tasks execution possible: " << result << endl;
}
```

Time complexity

In step ‘d’, each task can become a source only once and each edge (prerequisite) will be accessed and removed once. Therefore, the time complexity of the above algorithm will be $O(V+E)$, where ‘V’ is the total number of tasks and ‘E’ is the total number of prerequisites.

Space complexity

The space complexity will be $O(V+E)$,), since we are storing all of the prerequisites for each task in an adjacency list.

Similar Problems

Course Schedule: There are ‘N’ courses, labeled from ‘0’ to ‘N-1’. Each course can have some prerequisite courses which need to be completed before

it can be taken. Given the number of courses and a list of prerequisite pairs, find if it is possible for a student to take all the courses.

Solution: This problem is exactly similar to our parent problem. In this problem, we have courses instead of tasks.

Problem Statement

There are ‘N’ tasks, labeled from ‘0’ to ‘N-1’. Each task can have some prerequisite tasks which need to be completed before it can be scheduled. Given the number of tasks and a list of prerequisite pairs, write a method to find the ordering of tasks we should pick to finish all tasks.

Example 1:

Input: Tasks=3, Prerequisites=[0, 1], [1, 2]
Output: [0, 1, 2]
Explanation: To execute task '1', task '0' needs to finish first. Similarly, task '1' needs to finish before '2' can be scheduled. A possible scheduling of tasks is: [0, 1, 2]

Example 2:

Input: Tasks=3, Prerequisites=[0, 1], [1, 2], [2, 0]
Output: []
Explanation: The tasks have cyclic dependency, therefore they cannot be scheduled.

Example 3:

Input: Tasks=6, Prerequisites=[2, 5], [0, 5], [0, 4], [1, 4], [3, 2], [1, 3]
Output: [0 1 4 3 2 5]
Explanation: A possible scheduling of tasks is: [0 1 4 3 2 5]

Solution

This problem is similar to Tasks Scheduling, the only difference being that we need to find the best ordering of tasks so that it is possible to schedule them all.

```
using namespace std;


#include <iostream>
```

```
#include <queue>
#include <unordered_map>
#include <vector>

class TaskSchedulingOrder {
 public:
  static vector<int> findOrder(int tasks, const vector<vector<int>>& prerequisites) {
    vector<int> sortedOrder;
    if (tasks <= 0) {
      return sortedOrder;
    }

    // a. Initialize the graph
    unordered_map<int, int> inDegree;       // count of incoming edges for every vertex
    unordered_map<int, vector<int>> graph;  // adjacency list graph
    for (int i = 0; i < tasks; i++) {
      inDegree[i] = 0;
      graph[i] = vector<int>();
    }

    // b. Build the graph
    for (int i = 0; i < prerequisites.size(); i++) {
      int parent = prerequisites[i][0], child = prerequisites[i][1];
      graph[parent].push_back(child);  // put the child into it's parent's list
      inDegree[child]++;               // increment child's inDegree
    }

    // c. Find all sources i.e., all vertices with 0 in-degrees
    queue<int> sources;
```

```
for (auto entry : inDegree) {
  if (entry.second == 0) {
    sources.push(entry.first);
  }
}

// d. For each source, add it to the sortedOrder and subtract one from all of its children's
// in-degrees if a child's in-degree becomes zero, add it to the sources queue
while (!sources.empty()) {
  int vertex = sources.front();
  sources.pop();
  sortedOrder.push_back(vertex);
  vector<int> children =
      graph[vertex];  // get the node's children to decrement their in-degrees
  for (auto child : children) {
    inDegree[child]--;
    if (inDegree[child] == 0) {
      sources.push(child);
    }
  }
}

// if sortedOrder doesn't contain all tasks, there is a cyclic dependency between tasks,
// therefore, we will not be able to schedule all tasks
if (sortedOrder.size() != tasks) {
  return vector<int>();
}

return sortedOrder;
```

```
  }
};

int main(int argc, char* argv[]) {
  vector<int> result =
      TaskSchedulingOrder::findOrder(3, vector<vector<int>>{vector<int>{0, 1}, vector<int>{1, 2}});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = TaskSchedulingOrder::findOrder(
      3, vector<vector<int>>{vector<int>{0, 1}, vector<int>{1, 2}, vector<int>{2, 0}});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = TaskSchedulingOrder::findOrder(
      6, vector<vector<int>>{vector<int>{2, 5}, vector<int>{0, 5}, vector<int>{0, 4},
                             vector<int>{1, 4}, vector<int>{3, 2}, vector<int>{1, 3}});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

In step ‘d’, each task can become a source only once and each edge (prerequisite) will be accessed and removed once. Therefore, the time complexity of the above algorithm will be $O(V+E)$, where ‘V’ is the total number of tasks and ‘E’ is the total number of prerequisites.

Space complexity

The space complexity will be $O(V+E)$, since we are storing all of the prerequisites for each task in an adjacency list.

Similar Problems

Course Schedule: There are ‘N’ courses, labeled from ‘0’ to ‘N-1’. Each course has some prerequisite courses which need to be completed before it can be taken. Given the number of courses and a list of prerequisite pairs, write a method to find the best ordering of the courses that a student can take in order to finish all courses.

Solution: This problem is exactly similar to our parent problem. In this problem, we have courses instead of tasks.

Problem Statement

There are ‘N’ tasks, labeled from ‘0’ to ‘N-1’. Each task can have some prerequisite tasks which need to be completed before it can be scheduled. Given the number of tasks and a list of prerequisite pairs, write a method to print all possible ordering of tasks meeting all prerequisites.

Example 1:

Input: Tasks=3, Prerequisites=[0, 1], [1, 2]
Output: [0, 1, 2]
Explanation: There is only possible ordering of the tasks.

Example 2:

Input: Tasks=4, Prerequisites=[3, 2], [3, 0], [2, 0], [2, 1]
Output:
1) [3, 2, 0, 1]

2) [3, 2, 1, 0]
Explanation: There are two possible orderings of the tasks meeting all prerequisites.

Example 3:

Input: Tasks=6, Prerequisites=[2, 5], [0, 5], [0, 4], [1, 4], [3, 2], [1, 3]
Output:
1) [0, 1, 4, 3, 2, 5]
2) [0, 1, 3, 4, 2, 5]
3) [0, 1, 3, 2, 4, 5]
4) [0, 1, 3, 2, 5, 4]
5) [1, 0, 3, 4, 2, 5]
6) [1, 0, 3, 2, 4, 5]
7) [1, 0, 3, 2, 5, 4]
8) [1, 0, 4, 3, 2, 5]
9) [1, 3, 0, 2, 4, 5]
10) [1, 3, 0, 2, 5, 4]
11) [1, 3, 0, 4, 2, 5]
12) [1, 3, 2, 0, 5, 4]
13) [1, 3, 2, 0, 4, 5]

Solution

This problem is similar to Tasks Scheduling Order, the only difference is that we need to find all the topological orderings of the tasks.

At any stage, if we have more than one source available and since we can choose any source, therefore, in this case, we will have multiple orderings of the tasks. We can use a recursive approach with **Backtracking** to consider all sources at any step.

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <queue>

#include <string>

#include <unordered_map>

#include <vector>
```

```
class AllTaskSchedulingOrders {
 public:
  static void printOrders(int tasks, vector<vector<int>> &prerequisites) {
    vector<int> sortedOrder;
    if (tasks <= 0) {
      return;
    }

    // a. Initialize the graph
    unordered_map<int, int> inDegree;       // count of incoming edges for every vertex
    unordered_map<int, vector<int>> graph;  // adjacency list graph
    for (int i = 0; i < tasks; i++) {
      inDegree[i] = 0;
      graph[i] = vector<int>();
    }

    // b. Build the graph
    for (int i = 0; i < prerequisites.size(); i++) {
      int parent = prerequisites[i][0], child = prerequisites[i][1];
      graph[parent].push_back(child);  // put the child into it's parent's list
      inDegree[child]++;               // increment child's inDegree
    }

    // c. Find all sources i.e., all vertices with 0 in-degrees
    vector<int> sources;
    for (auto entry : inDegree) {
      if (entry.second == 0) {
        sources.push_back(entry.first);
```

```
    }
  }

  printAllTopologicalSorts(graph, inDegree, sources, sortedOrder);
 }

private:
 static void printAllTopologicalSorts(unordered_map<int, vector<int>> &graph,
                                      unordered_map<int, int> &inDegree,
                                      const vector<int> &sources, vector<int> &sortedOrder) {
  if (!sources.empty()) {
    for (int vertex : sources) {
      sortedOrder.push_back(vertex);
      vector<int> sourcesForNextCall = sources;
      // only remove the current source, all other sources should remain in the queue for the next
      // call
      sourcesForNextCall.erase(
          find(sourcesForNextCall.begin(), sourcesForNextCall.end(), vertex));

      vector<int> children =
          graph[vertex];  // get the node's children to decrement their in-degrees
      for (auto child : children) {
        inDegree[child]--;
        if (inDegree[child] == 0) {
          sourcesForNextCall.push_back(child);  // save the new source for the next call
        }
      }

      // recursive call to print other orderings from the remaining (and new) sources
```

```
      printAllTopologicalSorts(graph, inDegree, sourcesForNextCall, sortedOrder);

      // backtrack, remove the vertex from the sorted order and put all of its
      // children back to consider the next source instead of the current vertex
      sortedOrder.erase(find(sortedOrder.begin(), sortedOrder.end(), vertex));
      for (auto child : children) {
        inDegree[child]++;
      }
    }
  }

  // if sortedOrder doesn't contain all tasks, either we've a cyclic dependency between tasks, or
  // we have not processed all the tasks in this recursive call
  if (sortedOrder.size() == inDegree.size()) {
    for (int num : sortedOrder) {
      cout << num << " ";
    }
    cout << endl;
  }
 }
};

int main(int argc, char *argv[]) {
 vector<vector<int>> vec = {{0, 1}, {1, 2}};
 AllTaskSchedulingOrders::printOrders(3, vec);
 cout << endl;

 vec = {{3, 2}, {3, 0}, {2, 0}, {2, 1}};
 AllTaskSchedulingOrders::printOrders(4, vec);
```

```
  cout << endl;


  vec = {{2, 5}, {0, 5}, {0, 4}, {1, 4}, {3, 2}, {1, 3}};
  AllTaskSchedulingOrders::printOrders(6, vec);
  cout << endl;
}
```

Time and Space Complexity

If we don't have any prerequisites, all combinations of the tasks can represent a topological ordering. As we know, that there can be N! combinations for 'N' numbers, therefore the time and space complexity of our algorithm will be O(V! * E)$O(V! * E)$ where 'V' is the total number of tasks and 'E' is the total prerequisites. We need the 'E' part because in each recursive call, at max, we remove (and add back) all the edges.

Problem Statement

There is a dictionary containing words from an alien language for which we don't know the ordering of the alphabets. Write a method to find the correct order of the alphabets in the alien language. It is given that the input is a valid dictionary and there exists an ordering among its alphabets.

Example 1:

Input: Words: ["ba", "bc", "ac", "cab"]
Output: bac
Explanation: Given that the words are sorted lexicographically by the rules of the alien language, so from the given words we can conclude the following ordering among its characters:

1. From "ba" and "bc", we can conclude that 'a' comes before 'c'.
2. From "bc" and "ac", we can conclude that 'b' comes before 'a'

From the above two points, we can conclude that the correct character order is: "bac"

Example 2:

Input: Words: ["cab", "aaa", "aab"]
Output: cab
Explanation: From the given words we can conclude the following ordering among its characters:

1. From "cab" and "aaa", we can conclude that 'c' comes before 'a'.
2. From "aaa" and "aab", we can conclude that 'a' comes before 'b'

From the above two points, we can conclude that the correct character order is: "cab"

Example 3:

Input: Words: ["ywx", "wz", "xww", "xz", "zyy", "zwz"]
Output: ywxz
Explanation: From the given words we can conclude the following ordering among its characters:

1. From "ywx" and "wz", we can conclude that 'y' comes before 'w'.
2. From "wz" and "xww", we can conclude that 'w' comes before 'x'.
3. From "xww" and "xz", we can conclude that 'w' comes before 'z'
4. From "xz" and "zyy", we can conclude that 'x' comes before 'z'
5. From "zyy" and "zwz", we can conclude that 'y' comes before 'w'

From the above five points, we can conclude that the correct character order is: "ywxz"

Solution

Since the given words are sorted lexicographically by the rules of the alien language, we can always compare two adjacent words to determine the ordering of the characters. Take Example-1 above: [“ba”, “bc”, “ac”, “cab”]

1. Take the first two words “ba” and “bc”. Starting from the beginning of the words, find the first character that is different in both words: it would be ‘a’ from “ba” and ‘c’ from “bc”. Because of the sorted order of words (i.e. the dictionary!), we can conclude that ‘a’ comes before ‘c’ in the alien language.
2. Similarly, from “bc” and “ac”, we can conclude that ‘b’ comes before ‘a’.

These two points tell us that we are actually asked to find the topological ordering of the characters, and that the ordering rules should be inferred from adjacent words from the alien dictionary.

This makes the current problem similar to Tasks Scheduling Order, the only difference being that we need to build the graph of the characters by comparing adjacent words first, and then perform the topological sort for the graph to determine the order of the characters.

```
using namespace std;

#include <iostream>
#include <queue>
#include <string>
#include <unordered_map>
#include <vector>

class AlienDictionary {
 public:
  static string findOrder(const vector<string> &words) {
    if (words.empty() || words.empty()) {
      return "";
    }

    // a. Initialize the graph
    unordered_map<char, int> inDegree;       // count of incoming edges for every vertex
    unordered_map<char, vector<char>> graph;  // adjacency list graph
    for (auto word : words) {
      for (char character : word) {
        inDegree[character] = 0;
        graph[character] = vector<char>();
      }
    }

    // b. Build the graph
    for (int i = 0; i < words.size() - 1; i++) {
      string w1 = words[i], w2 = words[i + 1];  // find ordering of characters from adjacent words
      for (int j = 0; j < min(w1.length(), w2.length()); j++) {
```

```
    char parent = w1[j], child = w2[j];
    if (parent != child) {          // if the two characters are different
      graph[parent].push_back(child);  // put the child into it's parent's list
      inDegree[child]++;            // increment child's inDegree
      break;  // only the first different character between the two words will help us find the
              // order
    }
  }
}

// c. Find all sources i.e., all vertices with 0 in-degrees
queue<char> sources;
for (auto entry : inDegree) {
  if (entry.second == 0) {
    sources.push(entry.first);
  }
}

// d. For each source, add it to the sortedOrder and subtract one from all of its children's
// in-degrees if a child's in-degree becomes zero, add it to the sources queue
string sortedOrder;
while (!sources.empty()) {
  char vertex = sources.front();
  sources.pop();
  sortedOrder += vertex;
  vector<char> children =
      graph[vertex];  // get the node's children to decrement their in-degrees
  for (char child : children) {
    inDegree[child]--;
```

```
        if (inDegree[child] == 0) {
          sources.push(child);
        }
      }
    }

    // if sortedOrder doesn't contain all characters, there is a cyclic dependency between
    // characters, therefore, we will not be able to find the correct ordering of the characters
    if (sortedOrder.length() != inDegree.size()) {
      return "";
    }

    return sortedOrder;
  }
};

int main(int argc, char *argv[]) {
  string result = AlienDictionary::findOrder(vector<string>{"ba", "bc", "ac", "cab"});
  cout << "Character order: " << result << endl;

  result = AlienDictionary::findOrder(vector<string>{"cab", "aaa", "aab"});
  cout << "Character order: " << result << endl;

  result = AlienDictionary::findOrder(vector<string>{"ywx", "wz", "xww", "xz", "zyy", "zwz"});
  cout << "Character order: " << result << endl;
}
```

Time complexity

In step ‘d’, each task can become a source only once and each edge (a rule) will be accessed and removed once. Therefore, the time complexity of the above algorithm will be $O(V+E)$, where ‘V’ is the total number of different characters and ‘E’ is the total number of the rules in the alien language. Since, at most, each pair of words can give us one rule, therefore, we can conclude that the upper bound for the rules is $O(N)$ where ‘N’ is the number of words in the input. So, we can say that the time complexity of our algorithm is $O(V+N)$.

Space complexity

The space complexity will be $O(V+N)$, since we are storing all of the rules for each character in an adjacency list.

Reconstructing a Sequence (hard)

Given a sequence `originalSeq` and an array of sequences, write a method to find if `originalSeq` can be uniquely reconstructed from the array of sequences.

Unique reconstruction means that we need to find if `originalSeq` is the only sequence such that all sequences in the array are subsequences of it.

Example 1:

Input: originalSeq: [1, 2, 3, 4], seqs: [[1, 2], [2, 3], [3, 4]]
Output: true
Explanation: The sequences [1, 2], [2, 3], and [3, 4] can uniquely reconstruct
[1, 2, 3, 4], in other words, all the given sequences uniquely define the order of numbers
in the 'originalSeq'.

Example 2:

Input: originalSeq: [1, 2, 3, 4], seqs: [[1, 2], [2, 3], [2, 4]]
Output: false
Explanation: The sequences [1, 2], [2, 3], and [2, 4] cannot uniquely reconstruct
[1, 2, 3, 4]. There are two possible sequences we can construct from the given sequences:
1) [1, 2, 3, 4]
2) [1, 2, 4, 3]

Example 3:

Input: originalSeq: [3, 1, 4, 2, 5], seqs: [[3, 1, 5], [1, 4, 2, 5]]
Output: true

Explanation: The sequences [3, 1, 5] and [1, 4, 2, 5] can uniquely reconstruct [3, 1, 4, 2, 5].

Solution

Since each sequence in the given array defines the ordering of some numbers, we need to combine all these ordering rules to find two things:

1. Is it possible to construct the `originalSeq` from all these rules?
2. Are these ordering rules not sufficient enough to define the unique ordering of all the numbers in the `originalSeq`? In other words, can these rules result in more than one sequence?

Take Example-1:

originalSeq: [1, 2, 3, 4], seqs:[[1, 2], [2, 3], [3, 4]]

The first sequence tells us that '1' comes before '2'; the second sequence tells us that '2' comes before '3'; the third sequence tells us that '3' comes before '4'. Combining all these sequences will result in a unique sequence: [1, 2, 3, 4].

The above explanation tells us that we are actually asked to find the topological ordering of all the numbers and also to verify that there is only one topological ordering of the numbers possible from the given array of the sequences.

This makes the current problem similar to Tasks Scheduling Order with two differences:

1. We need to build the graph of the numbers by comparing each pair of numbers in the given array of sequences.
2. We must perform the topological sort for the graph to determine two things:
    - Can the topological ordering construct the `originalSeq`?
    - That there is only one topological ordering of the numbers possible. This can be confirmed if we do not have more than one source at any time while finding the topological ordering of numbers.

using namespace std;

```
#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>

class SequenceReconstruction {
 public:
  static bool canConstruct(const vector<int> &originalSeq, const vector<vector<int>> &sequences) {
    vector<int> sortedOrder;
    if (originalSeq.size() <= 0) {
      return false;
    }

    // a. Initialize the graph
    unordered_map<int, int> inDegree;       // count of incoming edges for every vertex
    unordered_map<int, vector<int>> graph;  // adjacency list graph
    for (auto seq : sequences) {
      for (int i = 0; i < seq.size(); i++) {
        inDegree[seq[i]] = 0;
        graph[seq[i]] = vector<int>();
      }
    }

    // b. Build the graph
    for (auto seq : sequences) {
      for (int i = 1; i < seq.size(); i++) {
        int parent = seq[i - 1], child = seq[i];
        graph[parent].push_back(child);
        inDegree[child]++;
```

```
  }
}

// if we don't have ordering rules for all the numbers we'll not able to uniquely construct the
// sequence
if (inDegree.size() != originalSeq.size()) {
  return false;
}

// c. Find all sources i.e., all vertices with 0 in-degrees
queue<int> sources;
for (auto entry : inDegree) {
  if (entry.second == 0) {
    sources.push(entry.first);
  }
}

// d. For each source, add it to the sortedOrder and subtract one from all of its children's
// in-degrees if a child's in-degree becomes zero, add it to the sources queue
while (!sources.empty()) {
  if (sources.size() > 1) {
    return false;  // more than one sources mean, there is more than one way to reconstruct the
                   // sequence
  }
  if (originalSeq[sortedOrder.size()] != sources.front()) {
    return false;  // the next source (or number) is different from the original sequence
  }
  int vertex = sources.front();
  sources.pop();
```

```
      sortedOrder.push_back(vertex);
      vector<int> children =
          graph[vertex];  // get the node's children to decrement their in-degrees
      for (auto child : children) {
        inDegree[child]--;
        if (inDegree[child] == 0) {
          sources.push(child);
        }
      }
    }

    // if sortedOrder's size is not equal to original sequence's size, there is no unique way to
    // construct
    return sortedOrder.size() == originalSeq.size();
  }
};

int main(int argc, char *argv[]) {
  bool result = SequenceReconstruction::canConstruct(
      vector<int>{1, 2, 3, 4},
      vector<vector<int>>{vector<int>{1, 2}, vector<int>{2, 3}, vector<int>{3, 4}});
  cout << "Can we uniquely construct the sequence: " << result << endl;

  result = SequenceReconstruction::canConstruct(
      vector<int>{1, 2, 3, 4},
      vector<vector<int>>{vector<int>{1, 2}, vector<int>{2, 3}, vector<int>{2, 4}});
  cout << "Can we uniquely construct the sequence: " << result << endl;

  result = SequenceReconstruction::canConstruct(
```

```
    vector<int>{3, 1, 4, 2, 5},
    vector<vector<int>>{vector<int>{3, 1, 5}, vector<int>{1, 4, 2, 5}});
  cout << "Can we uniquely construct the sequence: " << result << endl;
}
```

Time complexity

In step ‘d’, each number can become a source only once and each edge (a rule) will be accessed and removed once. Therefore, the time complexity of the above algorithm will be $O(V+E)$, where ‘V’ is the count of distinct numbers and ‘E’ is the total number of the rules. Since, at most, each pair of numbers can give us one rule, we can conclude that the upper bound for the rules is $O(N)$ where ‘N’ is the count of numbers in all sequences. So, we can say that the time complexity of our algorithm is $O(V+N)$.

Space complexity

The space complexity will be $O(V+N)$, since we are storing all of the rules for each number in an adjacency list.

Minimum Height Trees (hard)

We are given an undirected graph that has characteristics of a k-ary tree. In such a graph, we can choose any node as the root to make a k-ary tree. The root (or the tree) with the minimum height will be called **Minimum Height Tree (MHT)**. There can be multiple MHTs for a graph. In this problem, we need to find all those roots which give us MHTs. Write a method to find all MHTs of the given graph and return a list of their roots.

Example 1:

Input: vertices: 5, Edges: [[0, 1], [1, 2], [1, 3], [2, 4]]
Output:[1, 2]
Explanation: Choosing '1' or '2' as roots give us MHTs. In the below diagram, we can see that the height of the trees with roots '1' or '2' is three which is minimum.

Example 2:

Input: vertices: 4, Edges: [[0, 1], [0, 2], [2, 3]]
Output:[0, 2]

Explanation: Choosing '0' or '2' as roots give us MHTs. In the below diagram, we can see that the height of the trees with roots '0' or '2' is three which is minimum.

Example 3:

Input: vertices: 4, Edges: [[0, 1], [1, 2], [1, 3]]
Output:[1]

Solution

From the above-mentioned examples, we can clearly see that any leaf node (i.e., node with only one edge) can never give us an MHT because its adjacent non-leaf nodes will always give an MHT with a smaller height. All the adjacent non-leaf nodes will consider the leaf node as a subtree. Let's understand this with another example. Suppose we have a tree with root 'M' and height '5'. Now, if we take another node, say 'P', and make the 'M' tree as its subtree, then the height of the overall tree with root 'P' will be '6' (=5+1). Now, this whole tree can be considered a graph, where 'P' is a leaf as it has only one edge (connection with 'M'). This clearly shows that the leaf node ('P') gives us a tree of height '6' whereas its adjacent non-leaf node ('M') gives us a tree with smaller height '5' - since 'P' will be a child of 'M'.

This gives us a strategy to find MHTs. Since leaves can't give us MHT, we can remove them from the graph and remove their edges too. Once we remove the leaves, we will have new leaves. Since these new leaves can't give us MHT, we will repeat the process and remove them from the graph too. We will prune the leaves until we are left with one or two nodes which will be our answer and the roots for MHTs.

We can implement the above process using the topological sort. Any node with only one edge (i.e., a leaf) can be our source and, in a stepwise fashion, we can remove all sources from the graph to find new sources. We will repeat this process until we are left with one or two nodes in the graph, which will be our answer.

```
using namespace std;


#include <deque>

#include <iostream>

#include <string>
```

```
#include <unordered_map>
#include <vector>

class MinimumHeightTrees {
 public:
  static vector<int> findTrees(int nodes, vector<vector<int>>& edges) {
    vector<int> minHeightTrees;
    if (nodes <= 0) {
      return minHeightTrees;
    }

    // with only one node, since its in-degree will be 0, therefore, we need to handle it separately
    if (nodes == 1) {
      minHeightTrees.push_back(0);
      return minHeightTrees;
    }

    // a. Initialize the graph
    unordered_map<int, int> inDegree;       // count of incoming edges for every vertex
    unordered_map<int, vector<int>> graph;  // adjacency list graph
    for (int i = 0; i < nodes; i++) {
      inDegree[i] = 0;
      graph[i] = vector<int>();
    }

    // b. Build the graph
    for (int i = 0; i < edges.size(); i++) {
      int n1 = edges[i][0], n2 = edges[i][1];
      // since this is an undirected graph, therefore, add a link for both the nodes
```

```
  graph[n1].push_back(n2);
  graph[n2].push_back(n1);
  // increment the in-degrees of both the nodes
  inDegree[n1]++;
  inDegree[n2]++;
}

// c. Find all leaves i.e., all nodes with only 1 in-degree
deque<int> leaves;
for (auto entry : inDegree) {
  if (entry.second == 1) {
    leaves.push_back(entry.first);
  }
}

// d. Remove leaves level by level and subtract each leave's children's in-degrees.
// Repeat this until we are left with 1 or 2 nodes, which will be our answer.
// Any node that has already been a leaf cannot be the root of a minimum height tree, because
// its adjacent non-leaf node will always be a better candidate.
int totalNodes = nodes;
while (totalNodes > 2) {
  int leavesSize = leaves.size();
  totalNodes -= leavesSize;
  for (int i = 0; i < leavesSize; i++) {
    int vertex = leaves.front();
    leaves.pop_front();
    vector<int> children = graph[vertex];
    for (auto child : children) {
      inDegree[child]--;
```

```
        if (inDegree[child] == 1) {  // if the child has become a leaf
          leaves.push_back(child);
        }
       }
     }
    }

    std::move(std::begin(leaves), std::end(leaves), std::back_inserter(minHeightTrees));
    return minHeightTrees;
  }
};

int main(int argc, char* argv[]) {
  vector<vector<int>> vec = {{0, 1}, {1, 2}, {1, 3}, {2, 4}};
  vector<int> result = MinimumHeightTrees::findTrees(5, vec);
  cout << "Roots of MHTs: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  vec = {{0, 1}, {0, 2}, {2, 3}};
  result = MinimumHeightTrees::findTrees(4, vec);
  cout << "Roots of MHTs: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
```

```
  vec = {{0, 1}, {1, 2}, {1, 3}};
  result = MinimumHeightTrees::findTrees(4, vec);
  cout << "Roots of MHTs: ";
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Time complexity

In step ‘d’, each node can become a source only once and each edge will be accessed and removed once. Therefore, the time complexity of the above algorithm will be $O(V+E)$, where ‘V’ is the total nodes and ‘E’ is the total number of the edges.

Space complexity

The space complexity will be $O(V+E)$, since we are storing all of the edges for each node in an adjacency list.

Problem Statement

Given an unsorted array of numbers, find Kth smallest number in it.

Please note that it is the Kth smallest number in the sorted order, not the Kth distinct element.

Example 1:

Input: [1, 5, 12, 2, 11, 5], K = 3
Output: 5
Explanation: The 3rd smallest number is '5', as the first two smaller numbers are [1, 2].

Example 2:

Input: [1, 5, 12, 2, 11, 5], K = 4
Output: 5

Explanation: The 4th smallest number is '5', as the first three smaller numbers are [1, 2, 5].

Example 3:

Input: [5, 12, 11, -1, 12], K = 3
Output: 11
Explanation: The 3rd smallest number is '11', as the first two small numbers are [5, -1].

This is a well-known problem and there are multiple solutions available to solve this. A few other similar problems are:

1. Find the Kth largest number in an unsorted array.
2. Find the median of an unsorted array.
3. Find the 'K' smallest or largest numbers in an unsorted array.

Let's discuss different algorithms to solve this problem and understand their time and space complexity. Similar solutions can be devised for the above-mentioned three problems.

1. Brute-force

The simplest brute-force algorithm will be to find the Kth smallest number in a step by step fashion. This means that, first, we will find the smallest element, then 2nd smallest, then 3rd smallest and so on, until we have found the Kth smallest element. Here is what the algorithm will look like:

```
using namespace std;


#include <iostream>

#include <limits>

#include <vector>


class KthSmallestNumber {

 public:

  static int findKthSmallestNumber(const vector<int> &nums, int k) {

    // to handle duplicates, we will keep track of previous smallest number and its index

    int previousSmallestNum = numeric_limits<int>::min();
```

```
    int previousSmallestIndex = -1;
    int currentSmallestNum = numeric_limits<int>::max();
    int currentSmallestIndex = -1;
    for (int i = 0; i < k; i++) {
      for (int j = 0; j < nums.size(); j++) {
        if (nums[j] > previousSmallestNum && nums[j] < currentSmallestNum) {
          // found the next smallest number
          currentSmallestNum = nums[j];
          currentSmallestIndex = j;
        } else if (nums[j] == previousSmallestNum && j > previousSmallestIndex) {
          // found a number which is equal to the previous smallest number; since numbers can
          // repeat, we will consider 'nums[j]' only if it has a different index than previous
          // smallest
          currentSmallestNum = nums[j];
          currentSmallestIndex = j;
          break;  // break here as we have found our definitive next smallest number
        }
      }
      // current smallest number becomes previous smallest number for the next iteration
      previousSmallestNum = currentSmallestNum;
      previousSmallestIndex = currentSmallestIndex;
      currentSmallestNum = numeric_limits<int>::max();
    }

    return previousSmallestNum;
  }
};

int main(int argc, char *argv[]) {
```

```
  int result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 3);
  cout << "Kth smallest number is: " << result << endl;

  // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
  result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 4);
  cout << "Kth smallest number is: " << result << endl;

  result = KthSmallestNumber::findKthSmallestNumber(vector<int>{5, 12, 11, -1, 12}, 3);
  cout << "Kth smallest number is: " << result << endl;
}
```

Time & Space Complexity

The time complexity of the above algorithm will be $O(N*K)$. The algorithm runs in constant space $O(1)$.

2. Brute-force using Sorting

We can use an in-place sort like a **HeapSort** to sort the input array to get the Kth smallest number. Following is the code for this solution:

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <vector>

class KthSmallestNumber {
 public:
  static int findKthSmallestNumber(vector<int> &nums, int k) {
    sort(nums.begin(), nums.end());
    return nums[k - 1];
```

```
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {1, 5, 12, 2, 11, 5};
  int result = KthSmallestNumber::findKthSmallestNumber(vec, 3);
  cout << "Kth smallest number is: " << result << endl;

  // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
  result = KthSmallestNumber::findKthSmallestNumber(vec, 4);
  cout << "Kth smallest number is: " << result << endl;

  vec = {5, 12, 11, -1, 12};
  result = KthSmallestNumber::findKthSmallestNumber(vec, 3);
  cout << "Kth smallest number is: " << result << endl;
}
```

Time & Space Complexity

Sorting will take $O(NlogN)$ and if we are not using an in-place sorting algorithm, we will need $O(N)$ space.

3. Using Max-Heap

As discussed in Kth Smallest Number, we can iterate the array and use a **Max Heap** to keep track of 'K' smallest number. In the end, the root of the heap will have the Kth smallest number.

Here is what this algorithm will look like:

```
using namespace std;

#include <iostream>
```

```
#include <queue>
#include <vector>

class KthSmallestNumber {
 public:
  static int findKthSmallestNumber(const vector<int> &nums, int k) {
    priority_queue<int> maxHeap;

    // put first k numbers in the max heap
    for (int i = 0; i < k; i++) {
      maxHeap.push(nums[i]);
    }

    // go through the remaining numbers of the array, if the number from the array is smaller than
    // the top (biggest) number of the heap, remove the top number from heap and add the number from
    // array
    for (int i = k; i < nums.size(); i++) {
      if (nums[i] < maxHeap.top()) {
        maxHeap.pop();
        maxHeap.push(nums[i]);
      }
    }

    // the root of the heap has the Kth smallest number
    return maxHeap.top();
  }
};

int main(int argc, char *argv[]) {
```

```
  int result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 3);
  cout << "Kth smallest number is: " << result << endl;

  // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
  result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 4);
  cout << "Kth smallest number is: " << result << endl;

  result = KthSmallestNumber::findKthSmallestNumber(vector<int>{5, 12, 11, -1, 12}, 3);
  cout << "Kth smallest number is: " << result << endl;
}
```

Time & Space Complexity

The time complexity of the above algorithm is O(K*logK + (N-K)*logK)$O(K*logK+(N-K)*logK)$ which is asymptotically equal to O(N*logK)$O(N*logK)$. The space complexity will be O(K)$O(K)$ because we need to store 'K' smallest numbers in the heap.

4. Using Min-Heap

Also discussed in Kth Smallest Number, we can use a **Min Heap** to find the Kth smallest number. We can insert all the numbers in the min-heap and then extract the top 'K' numbers from the heap to find the Kth smallest number.

Time & Space Complexity

Building a heap with NN elements will take O(N)$O(N)$ and extracting 'K' numbers will take O(K*logN)$O(K*logN)$. Overall, the time complexity of this algorithm will be O(N+K*logN)$O(N+K*logN)$ and the space complexity will be O(N)$O(N)$.

5. Using Partition Scheme of Quicksort

Quicksort picks a number called **pivot** and partition the input array around it. After partitioning, all numbers smaller than the pivot are to the left of the

pivot, and all numbers greater than or equal to the pivot are to the right of the pivot. This ensures that the pivot has reached its correct sorted position.

We can use this partitioning scheme to find the Kth smallest number. We will recursively partition the input array and if, after partitioning, our pivot is at the `K-1` index we have found our required number; if not, we will choose one the following option:

1. If pivot's position is larger than `K-1`, we will recursively partition the array on numbers lower than the pivot.
2. If pivot's position is smaller than `K-1`, we will recursively partition the array on numbers greater than the pivot.

Here is what our algorithm will look like:

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <vector>


class KthSmallestNumber {

 public:

  static int findKthSmallestNumber(vector<int> &nums, int k) {

    return findKthSmallestNumberRec(nums, k, 0, nums.size() - 1);

  }


  static int findKthSmallestNumberRec(vector<int> &nums, int k, int start, int end) {

    int p = partition(nums, start, end);


    if (p == k - 1) {

      return nums[p];

    }
```

```
  if (p > k - 1)  // search lower part
  {
    return findKthSmallestNumberRec(nums, k, start, p - 1);
  }

  // search higher part
  return findKthSmallestNumberRec(nums, k, p + 1, end);
 }

private:
 static int partition(vector<int> &nums, int low, int high) {
  if (low == high) {
    return low;
  }

  int pivot = nums[high];
  for (int i = low; i < high; i++) {
    // all elements less than 'pivot' will be before the index 'low'
    if (nums[i] < pivot) {
      swap(nums, low++, i);
    }
  }
  // put the pivot in its correct place
  swap(nums, low, high);
  return low;
 }

 static void swap(vector<int> &nums, int i, int j) {
  int temp = nums[i];
```

```
    nums[i] = nums[j];
    nums[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {1, 5, 12, 2, 11, 5};
  int result = KthSmallestNumber::findKthSmallestNumber(vec, 3);
  cout << "Kth smallest number is: " << result << endl;

  // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
  result = KthSmallestNumber::findKthSmallestNumber(vec, 4);
  cout << "Kth smallest number is: " << result << endl;

  vec = {5, 12, 11, -1, 12};
  result = KthSmallestNumber::findKthSmallestNumber(vec, 3);
  cout << "Kth smallest number is: " << result << endl;
}
```

Time & Space Complexity

The above algorithm is known as QuickSelect and has a Worst case time complexity of O(N^2)$O(N2)$. The best and average case is O(N)$O(N)$, which is better than the best and average case of QuickSort. Overall, QuickSelect uses the same approach as QuickSort i.e., partitioning the data into two parts based on a pivot. However, contrary to QuickSort, instead of recursing into both sides QuickSelect only recurses into one side – the side with the element it is searching for. This reduces the average and best case time complexity from O(N*logN)$O(N*logN)$ to O(N)$O(N)$.

The worst-case occurs when, at every step, the partition procedure splits the N-length array into arrays of size ‘11’ and ‘N−1’. This can only happen when

the input array is sorted or if all of its elements are the same. This “unlucky” selection of pivot elements requires $O(N)$ recursive calls, leading to an $O(N^2)$ worst-case.

Worst-case space complexity will be $O(N)$ used for the recursion stack. See details under Quicksort.

6. Using Randomized Partitioning Scheme of Quicksort

As mentioned above, the worst case for **Quicksort** occurs when the partition procedure splits the N-length array into arrays of size ‘11’ and ‘N−1’. To mitigate this, instead of always picking a fixed index for pivot (e.g., in the above algorithm we always pick `nums[high]` as the pivot), we can randomly select an element as pivot. After randomly choosing the pivot element, we expect the split of the input array to be reasonably well balanced on average.

Here is what our algorithm will look like (only the highlighted lines have changed):

```
using namespace std;


#include <iostream>

#include <vector>


class KthSmallestNumber {

 public:

  static int findKthSmallestNumber(vector<int> &nums, int k) {

    return findKthSmallestNumberRec(nums, k, 0, nums.size() - 1);

  }


  static int findKthSmallestNumberRec(vector<int> &nums, int k, int start, int end) {

    int p = partition(nums, start, end);
```

```
  if (p == k - 1) {
    return nums[p];
  }

  if (p > k - 1)  // search lower part
  {
    return findKthSmallestNumberRec(nums, k, start, p - 1);
  }

  // search higher part
  return findKthSmallestNumberRec(nums, k, p + 1, end);
 }

private:
 static int partition(vector<int> &nums, int low, int high) {
  if (low == high) {
    return low;
  }

  srand(time(0));
  int pivotIndex = low + rand() % (high - low);
  swap(nums, pivotIndex, high);

  int pivot = nums[high];
  for (int i = low; i < high; i++) {
    // all elements less than 'pivot' will be before the index 'low'
    if (nums[i] < pivot) {
      swap(nums, low++, i);
    }
```

```
    }
    // put the pivot in its correct place
    swap(nums, low, high);

    return low;
  }

  static void swap(vector<int> &nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {1, 5, 12, 2, 11, 5};
  int result = KthSmallestNumber::findKthSmallestNumber(vec, 3);
  cout << "Kth smallest number is: " << result << endl;

  // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
  result = KthSmallestNumber::findKthSmallestNumber(vec, 4);
  cout << "Kth smallest number is: " << result << endl;

  vec = {5, 12, 11, -1, 12};
  result = KthSmallestNumber::findKthSmallestNumber(vec, 3);
  cout << "Kth smallest number is: " << result << endl;
}
```

Time & Space Complexity

The above algorithm has the same worst and average case time complexities as mentioned for the previous algorithm. But choosing the pivot randomly has the effect of rendering the worst-case very unlikely, particularly for large arrays. Therefore, the **expected** time complexity of the above algorithm will be O(N)$O(N)$, but the absolute worst case is still O(N^2)$O(N2)$. Practically, this algorithm is a lot faster than the non-randomized version.

7. Using the Median of Medians

We can use the Median of Medians algorithm to choose a **good pivot** for the partitioning algorithm of the **Quicksort**. This algorithm finds an approximate median of an array in linear time O(N)$O(N)$. When this approximate median is used as the pivot, the worst-case complexity of the partitioning procedure reduces to linear O(N)$O(N)$, which is also the asymptotically optimal worst-case complexity of any sorting/selection algorithm. This algorithm was originally developed by Blum, Floyd, Pratt, Rivest, and Tarjan and was describe in their 1973 paper.

This is how the partitioning algorithm works:

1. If we have 5 or less than 5 elements in the input array, we simply take its first element as the pivot. If not then we divide the input array into subarrays of five elements (for simplicity we can ignore any subarray having less than five elements).
2. Sort each subarray to determine its median. Sorting a small and fixed numbered array takes constant time. At the end of this step, we have an array containing medians of all the subarray.
3. Recursively call the partitioning algorithm on the array containing medians until we get our pivot.
4. Every time the partition procedure needs to find a pivot, it will follow the above three steps.

Here is what this algorithm will look like:

```
using namespace std;


#include <algorithm>

#include <iostream>

#include <vector>
```

```
class KthSmallestNumber {
public:
  static int findKthSmallestNumber(vector<int> &nums, int k) {
    return findKthSmallestNumberRec(nums, k, 0, nums.size() - 1);
  }

  static int findKthSmallestNumberRec(vector<int> &nums, int k, int start, int end) {
    int p = partition(nums, start, end);

    if (p == k - 1) {
      return nums[p];
    }

    if (p > k - 1) { // search lower part
      return findKthSmallestNumberRec(nums, k, start, p - 1);
    }

    // search higher part
    return findKthSmallestNumberRec(nums, k, p + 1, end);
  }

private:
  static int partition(vector<int> &nums, int low, int high) {
    if (low == high) {
      return low;
    }

    int median = medianOfMedians(nums, low, high);
```

```
  // find the median in the array and swap it with 'nums[high]' which will become our pivot
  for (int i = low; i < high; i++) {
    if (nums[i] == median) {
      swap(nums, i, high);
      break;
    }
  }

  int pivot = nums[high];
  for (int i = low; i < high; i++) {
    // all elements less than 'pivot' will be before the index 'low'
    if (nums[i] < pivot) {
      swap(nums, low++, i);
    }
  }
  // put the pivot in its correct place, remember nums[high] is our pivot
  swap(nums, low, high);
  return low;
}

static void swap(vector<int> &nums, int i, int j) {
  int temp = nums[i];
  nums[i] = nums[j];
  nums[j] = temp;
}

static int medianOfMedians(vector<int> &nums, int low, int high) {
  int n = high - low + 1;
  // if we have less than 5 elements, ignore the partitioning algorithm
```

```
    if (n < 5) {
      return nums[low];
    }

    // for simplicity, lets ignore any partition with less than 5 elements
    int numOfPartitions = n / 5; // represents total number of 5 elements partitions
    vector<int> medians(numOfPartitions);
    for (int i = 0; i < numOfPartitions; i++) {
      int partitionStart = low + i * 5; // starting index of the current partition
      sort(nums.begin() + partitionStart,
          nums.begin() + partitionStart + 5); // sort the 5 elements array
      medians[i] = nums[partitionStart + 2];   // get the middle element (or the median)
    }

    return partition(medians, 0, numOfPartitions - 1);
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {1, 5, 12, 2, 11, 5};
  int result = KthSmallestNumber::findKthSmallestNumber(vec, 3);
  cout << "Kth smallest number is: " << result << endl;

  // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
  result = KthSmallestNumber::findKthSmallestNumber(vec, 4);
  cout << "Kth smallest number is: " << result << endl;

  vec = {5, 12, 11, -1, 12};
  result = KthSmallestNumber::findKthSmallestNumber(vec, 3);
```

```
  cout << "Kth smallest number is: " << result << endl;
}
```

Time & Space Complexity

The above algorithm has a guaranteed $O(N)$ worst-case time. Please see the proof of its running time here and under "Selection-based pivoting". The worst-case space complexity is $O(N)$.

Conclusion

Theoretically, the Median of Medians algorithm gives the best time complexity of $O(N)$ but practically both the Median of Medians and the Randomized Partitioning algorithms nearly perform equally.

In the context of **Quicksort**, given an $O(N)$ selection algorithm using the Median of Medians, one can use it to find the ideal pivot (the median) at every step of quicksort and thus produce a sorting algorithm with $O(NlogN)$ running time in the worst-case. Though practical implementations of this variant are considerably slower on average, they are of theoretical interest because they show that an optimal selection algorithm can yield an optimal sorting algorithm.

Problem Statement

In a non-empty array of integers, every number appears twice except for one, find that single number.

Example 1:

Input: 1, 4, 2, 1, 3, 2, 3
Output: 4

Example 2:

Input: 7, 9, 7
Output: 9

Solution

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.

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**?

Solution with XOR

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.

```
using namespace std;


#include <iostream>

#include <vector>


class SingleNumber {

 public:

  static int findSingleNumber(const vector<int>& arr) {

    int num = 0;

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

      num = num ^ arr[i];
```

```
    }
    return num;
  }
};

int main(int argc, char* argv[]) {
 cout << SingleNumber::findSingleNumber(vector<int>{1, 4, 2, 1, 3, 2, 3}) << endl;
}
```

Time Complexity: Time complexity of this solution is $O(n)$ as we iterate through all numbers of the input once.

Space Complexity: The algorithm runs in constant space O(1).

Problem Statement

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.

Example 1:

Input: [1, 4, 2, 1, 3, 5, 6, 2, 3, 5]
Output: [4, 6]

Example 2:

Input: [2, 1, 3, 2]
Output: [1, 3]

Solution

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.

```
using namespace std;


#include <iostream>

#include <vector>


class TwoSingleNumbers {

public:

  static vector<int> findSingleNumbers(vector<int> &nums) {

    // get the XOR of the all the numbers

    int n1xn2 = 0;

    for (int num : nums) {
```

```
    n1xn2 ^= num;
  }

  // get the rightmost bit that is '1'
  int rightmostSetBit = 1;
  while ((rightmostSetBit & n1xn2) == 0) {
    rightmostSetBit = rightmostSetBit << 1;
  }
  int num1 = 0, num2 = 0;
  for (int num : nums) {
    if ((num & rightmostSetBit) != 0) // the bit is set
      num1 ^= num;
    else // the bit is not set
      num2 ^= num;
  }
  return vector<int>{num1, num2};
 }
};

int main(int argc, char *argv[]) {
 vector<int> v1 = {1, 4, 2, 1, 3, 5, 6, 2, 3, 5};
 vector<int> result = TwoSingleNumbers::findSingleNumbers(v1);
 cout << "Single numbers are: " << result[0] << ", " << result[1] << endl;

 v1 = {2, 1, 3, 2};
 result = TwoSingleNumbers::findSingleNumbers(v1);
 cout << "Single numbers are: " << result[0] << ", " << result[1] << endl;
}
```

Time Complexity

The time complexity of this solution is $O(n)$ where 'n' is the number of elements in the input array.

Space Complexity

The algorithm runs in constant space O(1).

Problem Statement

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.

Example 1:

Input: 8
Output: 7
Explanation: 8 is 1000 in binary, its complement is 0111 in binary, which is 7 in base-10.

Example 2:

Input: 10
Output: 5
Explanation: 10 is 1010 in binary, its complement is 0101 in binary, which is 5 in base-10.

Solution

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'.

```
using namespace std;


#include <iostream>

#include <math.h>


class CalculateComplement {

public:

  static int bitwiseComplement(int num) {

    // count number of total bits in 'num'
```

```
    int bitCount = 0;
    int n = num;
    while (n > 0) {
      bitCount++;
      n = n >> 1;
    }

    // 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'
    int all_bits_set = pow(2, bitCount) - 1;

    // from the solution description: complement = number ^ all_bits_set
    return num ^ all_bits_set;
  }
};

int main(int argc, char *argv[]) {
  cout << "Bitwise complement is: " << CalculateComplement::bitwiseComplement(8) << endl;
  cout << "Bitwise complement is: " << CalculateComplement::bitwiseComplement(10) << endl;
}
```

Time Complexity

Time complexity of this solution is $O(b)$ where 'b' is the number of bits required to store the given number.

Space Complexity

Space complexity of this solution is $O(1)$.

Problem Statement (hard)

Given a binary matrix representing an image, we want to flip the image horizontally, then invert it.

To flip an image horizontally means that each row of the image is reversed. For example, flipping [0, 1, 1] horizontally results in [1, 1, 0].

To invert an image means that each 0 is replaced by 1, and each 1 is replaced by 0. For example, inverting [1, 1, 0] results in [0, 0, 1].

Example 1:

```
Input: [
  [1,0,1],
  [1,1,1],
  [0,1,1]
]
Output: [
  [0,1,0],
  [0,0,0],
  [0,0,1]
]
```

Explanation: First reverse each row: `[[1,0,1],[1,1,1],[1,1,0]]`. Then, invert the image: `[[0,1,0],[0,0,0],[0,0,1]]`

Example 2:

```
Input: [
  [1,1,0,0],
  [1,0,0,1],
  [0,1,1,1],
  [1,0,1,0]
]
Output: [
  [1,1,0,0],
  [0,1,1,0],
  [0,0,0,1],
  [1,0,1,0]
]
```

Explanation: First reverse each row: `[[0,0,1,1],[1,0,0,1],[1,1,1,0],[0,1,0,1]]`. Then invert the image: `[[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]`

Solution

- **Flip**: We can flip the image in place by replacing *ith* element from left with the *ith* element from the right.
- **Invert**: We can take XOR of each element with 1. If it is 1 then it will become 0 and if it is 0 then it will become 1.

```
#include <iostream>

#include <vector>


using namespace std;


class Solution {

 public:

  static vector<vector<int>> flipAndInvertImage(vector<vector<int>> arr) {

   int s = arr[0].size();

   for (int row = 0; row < arr.size(); row++) {

     for (int col = 0; col < (s + 1) / 2; ++col) {

       int tmp = arr[row][col] ^ 1;

       arr[row][col] = arr[row][s - 1 - col] ^ 1;

       arr[row][s - 1 - col] = tmp;

     }

   }

   return arr;

  }


  static void print(const vector<vector<int>> arr) {
```

```
    for(int i=0; i < arr.size(); i++) {
      for (int j=0; j < arr[i].size(); j++) {
        cout << arr[i][j] << " ";
      }
      cout << "\n";
    }
  }
};

int main(int argc, char* argv[]) {
  vector<vector<int>> arr = vector<vector<int>>{{1, 0, 1}, {1, 1, 1}, {0, 1, 1}};
  Solution::print(Solution::flipAndInvertImage(arr));
  cout << "\n";

  vector<vector<int>> arr2 = vector<vector<int>>{{1,1,0,0},{1,0,0,1},{0,1,1,1},{1,0,1,0}};
  Solution::print(Solution::flipAndInvertImage(arr2));
}
```

Time Complexity

The time complexity of this solution is $O(n)$ as we iterate through all elements of the input.

Space Complexity

The space complexity of this solution is $O(1)$.