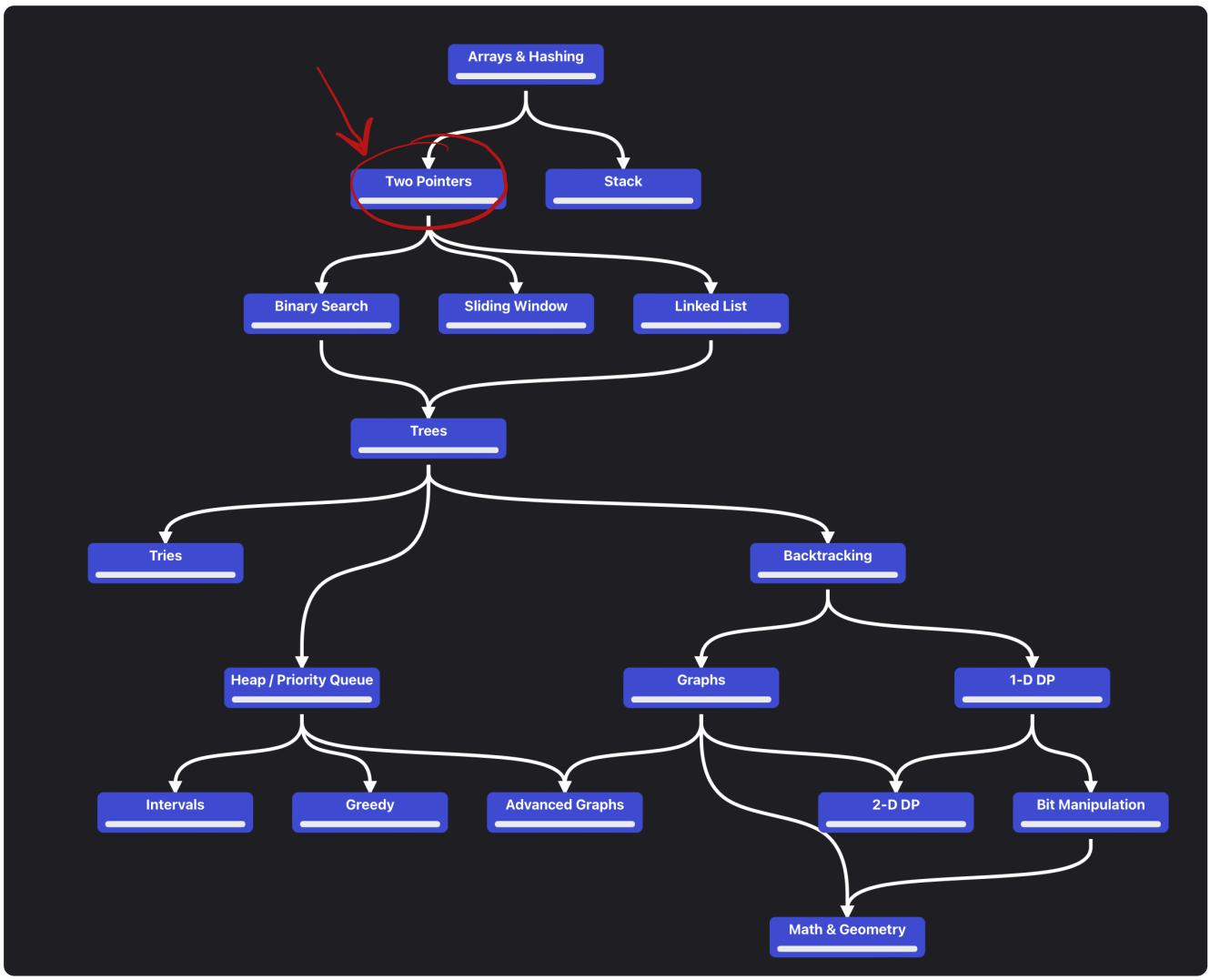
Two-Pointers



125. Valid Palindrome

Easy 8.7K 8.1K

Companies

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string `s`, return `true` *if it is a* ***palindrome****, or* `false` *otherwise.*

Example 1:

```
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.
```

Example 2:

```
Input: s = "race a car"
Output: false
Explanation: "raceacar" is not a palindrome.
```

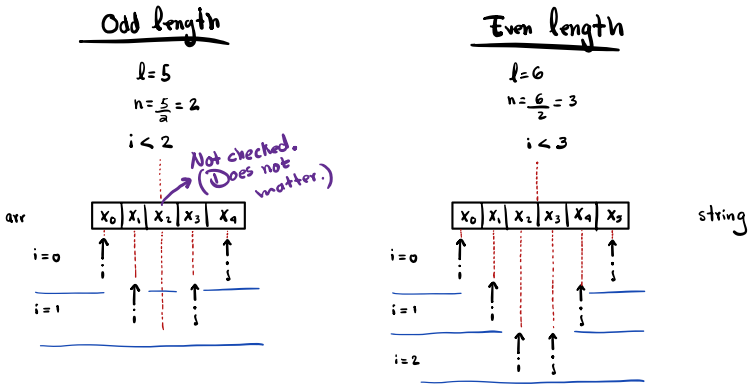
Example 3:

```
Input: s = " "
Output: true
Explanation: s is an empty string "" after removing non-alphanumeric characters.
Since an empty string reads the same forward and backward, it is a palindrome.
```

125. Valid Palindrome

Clean up String (Remove non-alpha numeric characters)

Traverse half the string. Using two pointers to compare characters in a symmetric fashion.



Note: For:

$l = 0 \Rightarrow n = \frac{0}{2} = 0$

$l = 1 \Rightarrow n = \frac{1}{2} = 0$

For loop is NOT executed $\Rightarrow$ return TRUE

```
for ( i < l/2 ) {
    if (arr[i] ≠ arr[j]) {
        return false;
    }
}
```

```cpp
bool isPalindrome(string s) {

    string cleaned;
    for (char c : s) { // Create an entirely new string by
                       // filtering out alpha numeric characters
                       // and transforming them lower case
        if (isalnum(c)) {
            cleaned += tolower(c);
        }
    }

    long left = 0, right = cleaned.size() - 1;

    while (left < right) {
        if (cleaned[left] != cleaned[right]) {
            return false;
        }
        left++;
        right--;
    }

    return true;
}
```

167. Two Sum

167. Two Sum II - Input Array Is Sorted

Medium 11.2K 1.4K

Companies

Given a **1-indexed** array of integers `numbers` that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific `target` number. Let these two numbers be `numbers[index1]` and `numbers[index2]` where `1 <= index1 < index2 <= numbers.length`.

Return *the indices of the two numbers,* `index1` *and* `index2`*, **added by one** as an integer array* `[index1, index2]` *of length 2.*

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

Example 1:

Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore, $index_1 = 1$, $index_2 = 2$. We return [1, 2].

Example 2:

Input: numbers = [2,3,4], target = 6
Output: [1,3]
Explanation: The sum of 2 and 4 is 6. Therefore $index_1 = 1$, $index_2 = 3$. We return [1, 3].

Example 3:

Input: numbers = [-1,0], target = -1
Output: [1,2]
Explanation: The sum of -1 and 0 is -1. Therefore $index_1 = 1$, $index_2 = 2$. We return [1, 2].

Target = 15

1	3	4	5	8	10	16	a[i] + a[j]
i						j	1 + 16 = 17 > 15 ⇒ Decrease j
i					j		1 + 10 = 11 < 15 ⇒ Increase i
	i				j		3 + 10 = 13 < 15 ⇒ Increase i
		i			j		4 + 10 = 14 < 15 ⇒ Increase i
			i		j		5 + 10 = 15 = 15

Note: This might be an example of a greedy algorithm.

```cpp
vector<int> twoSum(vector<int>& numbers, int target) {
    vector<int> ans;

    int i = 0;  // Pointer to the first element
    long j = numbers.size() - 1; // Pointer to the last element

    while (i < j) {

        int num1 = numbers[i];
        int num2 = numbers[j];

        int dif = target - (num1 + num2); // compute difference

        if (dif > 0) { // We need a larger number -> increase left pointer
            i++;
        } else if (dif < 0) { // We need a smaller number -> decrease right pointer
            j--;
        } else {  // dif == 0, then we've found our answer.
            ans.push_back(i + 1);
            ans.push_back(j + 1);
            break;
        }
    }
    return ans;
}
```

15. 3Sum

Medium 29.5K 2.7K

Companies

Given an integer array nums, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

> **Input:** nums = [-1,0,1,2,-1,-4]
> **Output:** [[-1,-1,2],[-1,0,1]]
> **Explanation:**
> nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
> nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
> nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
> The distinct triplets are [-1,0,1] and [-1,-1,2].
> Notice that the order of the output and the order of the triplets does not matter.

Example 2:

> **Input:** nums = [0,1,1]
> **Output:** []
> **Explanation:** The only possible triplet does not sum up to 0.

Example 3:

> **Input:** nums = [0,0,0]
> **Output:** [[0,0,0]]
> **Explanation:** The only possible triplet sums up to 0.

```cpp
vector<vector<int>> threeSum(vector<int>& nums) {

    vector<vector<int>> ans;
    sort(nums.begin(), nums.end()); // First sort

    long size = nums.size();

    for (int i = 0; i < size - 2; i++) { // i only gets up to third to last element

        int j = i + 1;       // middle pointer
        long k = size - 1;   // right pointer

        if (i > 0 && (nums[i] == nums[i - 1]) ) { // Checked only after first iteration
            continue; // We already found all the triples containing nums[i - 1], we must skip,
        }             // otherwise, this leads to repeated triple. Example: [-4,-1,-1,0,1,2]

        while (j < k) {

            int first = nums[i];
            int second = nums[j];
            int third = nums[k];

            int sum = first + second + third;

            if (sum < 0) { // We need a larger number, so increase left pointer.
                j++;
            } else if (sum > 0) { // We need a smaller number, so decrease right pointer.
                k--;
            } else { // sum == 0, we've found a triple

                ans.push_back({first, second, third});

                // Must update values right away.
                j++;
                k--;

                // Check if current pointer values are the same as before. Skip all repeated values.
                // We also have to check that j does not surpass k.
                // Example [0,0,0,0]. Here, j goes out of bounds if we don't check j < k.
                while(j < k && (nums[j] == nums[j - 1]) ) j++;
                while(j < k && (nums[k] == nums[k + 1]) ) k--;

            }
        }
    }
    return ans;
}
```

11. Container With Most Water

11. Container With Most Water

Hint

Medium 27.7K 1.6K

Companies

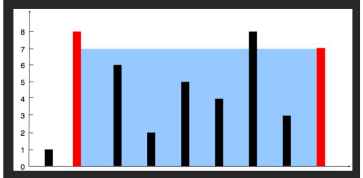
You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the i^{th} line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.

Example 1:




```
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.
```

Example 2:

```
Input: height = [1,1]
Output: 1
```

```cpp
long maxArea(vector<int>& height) {

    long size = height.size();

    long maxArea = 0;

    long i = 0, j = size - 1;  // Prepare pointers

    while (i < j) {

        long left = height[i];
        long right = height[j];

        long A = 0;

        if (left < right) {         // Calculate area depending on the smallest 'wall'
            A = (j - i) * left;
            i++;                    // Move pointer corresponding to the lower 'wall'
        } else {                    // left >= right
            A = (j - i) * right;
            j--;                    // Move pointer corresponding to the lower 'wall'
        }

        if (maxArea < A) {  // Update maxArea if necessary
            maxArea = A;
        }
    }

    return maxArea;
}
```

42. Trapping Rain Water

42. Trapping Rain Water

Hard 30.2K 449

Companies

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:

```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.
```

Example 2:

```
Input: height = [4,2,0,3,2,5]
Output: 9
```

```cpp
int trap(vector<int>& height) {
    int n = height.size();
    if (n == 0) return 0; // handle empty vector

    int left = 0, right = n - 1;    // These are pointers
    int leftMax = 0, rightMax = 0;  // These are max heights found so far, going leftwards and rightwards

    int trappedWater = 0;

    while (left < right) {
        if (height[left] < height[right]) {           // Left height is smaller than right height.
                                                      // Then, there is the potential for the left height
                                                      // to contain the water

            if (height[left] >= leftMax)              // Check if left height is now a maximum height found so far from the left.
                leftMax = height[left];
            else                // heightx[left] < leftMax  ->  Then we are 100% sure there is water above this height.
                                                        // Why? We know that to the right there is a height higher than the maxLeft.
                                                        // And we know that the current height is lower than the maxLeft

                trappedWater += leftMax - height[left];
            left++;
        } else {        // height[left] >= height[right]

            if (height[right] >= rightMax)
                rightMax = height[right];
            else
                trappedWater += rightMax - height[right];
            right--;
        }
    }

    return trappedWater;
}
```