**Two Pointers: Introduction**

Let’s go over the Two Pointers pattern, its real-world applications, and some problems we can solve with it.

**We'll cover the following**

* [Overview](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N75vOogKyl6#Overview)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N75vOogKyl6#Examples)
* [Does my problem match this pattern?](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N75vOogKyl6#Does-my-problem-match-this-pattern?)
* [Real-world problems](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N75vOogKyl6#Real-world-problems)
* [Strategy time!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N75vOogKyl6#Strategy-time!)

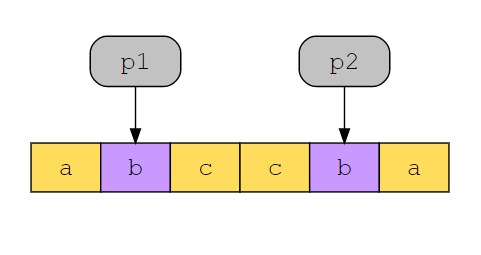
**Overview**

As the name suggests, the **two pointers** pattern uses two pointers to iterate over an array or list until the conditions of the problem are satisfied. This is useful because it allows us to keep track of the values of two different indexes in a single iteration. Whenever there’s a requirement to find two data elements in an array that satisfy a certain condition, the two pointers pattern should be the first strategy to come to mind.

The pointers can be used to iterate the data structure in one or both directions, depending on the problem statement. For example, to identify whether a string is a palindrome, we can use one pointer to iterate the string from the beginning and the other to iterate it from the end. At each step, we can compare the values of the two pointers and see if they meet the palindrome properties.

The naive approach to solving this problem would be using nested loops, with a time complexity of *O*(*n*2). However, by using two pointers moving towards the middle from either end, we exploit the symmetry property of palindromic strings. This allows us to compare the elements in a single loop, making the algorithm more efficient with a time complexity of *O*(*n*).

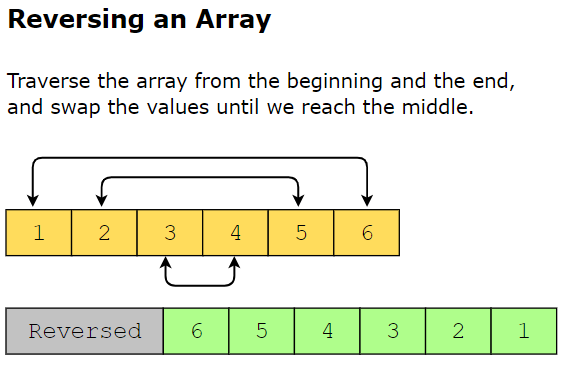
Here’s how the pointers will move along the string:

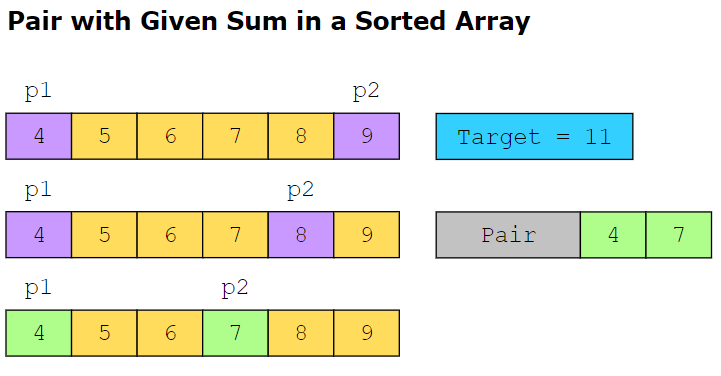


Essentially, the two pointers pattern is an application of the prune-and-search strategy, in which, at every step, we’re able to safely prune—that is, eliminate—a set of possible solutions.

## Examples

The following examples illustrate some problems that can be solved with this approach:





**Does my problem match this pattern?**

* Yes, if all of these conditions are fulfilled:
  + The input data can be traversed in a linear fashion, that is, it’s in an array, in a linked list, or in a string of characters.
  + The input data is sorted, or else, arranged in a way that is relevant to the problem, such as numeric data sorted in ascending or descending order, or characters arranged symmetrically.

Additionally, problems in this pattern usually involve comparing or swapping values pointed to by two indexes. In less common cases, each index may move along a separate array or string.

* No, if either of these conditions is fulfilled:
  + The input data cannot be traversed in a linear fashion, that is, it’s neither in an array, nor in a linked list, nor in a string of characters.
  + The problem requires an exhaustive search of the solution space, that is, eliminating one solution does not eliminate any others.

**Real-world problems**

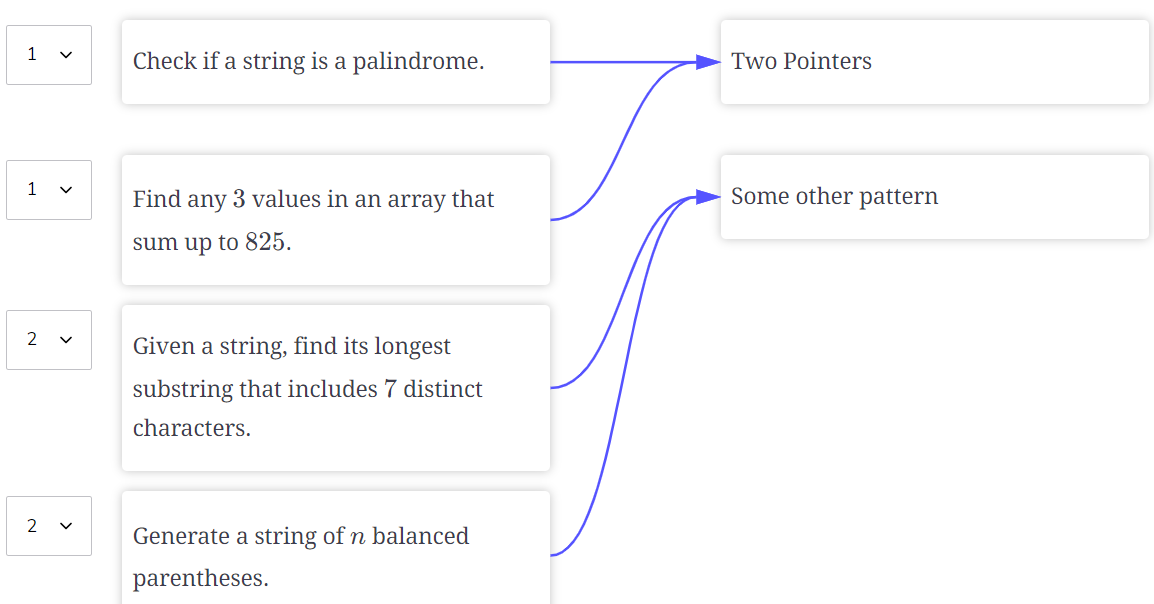
Many problems in the real world use the two pointers pattern. Let’s look at some examples.

* **Transmission errors:** In network protocol, response packets take the same route (in reverse) back to the source that the request packet took to the destination. Sometimes, the path may differ due to errors, and we can tolerate at most one diversion router. The path—in terms of the IDs of routers along the way—is recorded in an array present inside the packet. Two pointers can be used to determine whether the same path is followed from the source to the destination and from destination to source to identify the paths with more than one diversion router. So, our topology remains intact and transmission errors are detected.
* **Product suggestions:** While shopping online, when customers view their cart and the current total doesn’t qualify for free shipping, we want to show them pairs of products that can be bought together to make the total amount equal to the amount required to be eligible for free delivery. Two pointers can be used to suggest the pairs that add up to the required cost for free shipping.

**Strategy time!**

Match the problems that can be solved using the two pointers pattern.

**Note:** In the following exercise, option 1 corresponds to “Two Pointers” and option 2 corresponds to “Some other pattern”.



**Valid Palindrome**

Try to solve the Valid Palindrome problem.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7AY9D6zk3Jr#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7AY9D6zk3Jr#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7AY9D6zk3Jr#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7AY9D6zk3Jr#Figure-it-out!)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7AY9D6zk3Jr#Try-it-yourself)

**Statement**

Write a function that takes a string s as input and checks whether it’s a palindrome or not.

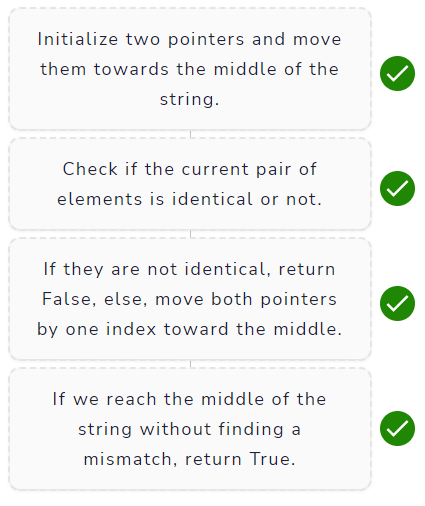
**Note**: A phrase, word or sequence is a palindrome that reads the same backwards as forwards.

**Constraints:**

* 1≤1≤ s.length ≤2×105≤2×105
* The string won’t have any spaces and will only consist of ASCII characters.

## Figure it out!

We have a game for you to play. Rearrange the logical building blocks to develop a clearer understanding on how to solve this problem.



# Solution: Valid Palindrome

Let's solve the Valid Palindrome problem using the Two Pointers pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Solution)
  + [Naive approach](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Naive-approach)
  + [Optimized approach using two pointers](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Optimized-approach-using-two-pointers)
    - [Step-by-step solution construction](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Step-by-step-solution-construction)
    - [Just the code](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Just-the-code)
    - [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Solution-summary)
    - [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/gk2JZw3nXm6#Space-complexity)

## Statement

Write a function that takes a string s as input and checks whether it’s a palindrome or not.

**Note**: A phrase, word or sequence is a palindrome that reads the same backwards as forwards.

**Constraints:**

* 1≤1≤ s.length ≤2×105≤2×105
* The string won’t have any spaces and will only consist of ASCII characters.

## Solution

So far, you’ve probably brainstormed some approaches and have an idea of how to solve this problem. Let’s explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

### Naive approach

The naive approach to solve this problem is to reverse the string and then compare the reversed string with the original string. If they match, the original string is a valid palindrome. This, despite being a solution of linear time complexity, will still occupy extra space, and we can use an optimized approach to save extra space.

### Optimized approach using two pointers

As we previously mentioned, a word or phrase that reads the same if it is reversed is defined as a palindrome. That means elements at both ends of the word or phrase should be precisely the same.

The two pointers approach would allow us to solve this problem in linear time and without any extra space complexity or built-in functions because we’ll be simply traverse the array from the start and the end at the same time to reach the middle of the string.

#### Step-by-step solution construction

We’ll have a pair of two pointers, where the first pointer is at the starting element of our string while the second pointer is at the end of the string. We move the two pointers towards the middle of the string and, at each step, we compare each element to its counterpart. The moment we encounter a non identical pair, we can return False because our string can’t be a palindrome.

We can actually dry run an example and see how this approach works with the string “RACECAR”. The first step is to set up two pointers and move them toward the middle of the string. We can do that with the following code snippet:

That's how we always traverse in opposite directions with the help of two pointers. The termination condition for our code is that the left pointer should always be less than the right pointer because the moment they cross each other, we reach the middle of the string and don't need to go any further. We can check how this works too with our example!

In the code sample above, we see that in the case of palindromic strings, at each step in the traversal toward the middle of the string, the characters at both the left and the right indexes are identical. However, in the third test case, “TART” (which isn't a palindromic string), in the second iteration of the loop, we see that our output identifies that the characters at the left and right indexes aren't the same. This observation allows us to add a simple check to our code.

If we encounter non identical pairs, we can simply return FALSE because the string isn't a palindrome, and we don't need to test any further. Otherwise, we're able to traverse to the middle of the string. In this case, we return TRUE because each element had a match at the expected position in the string.

class ValidPalindrome {

public static boolean isPalindrome(String s) {

int left = 0;

int right = s.length() - 1;

while (left < right) {

if (s.charAt(left) != s.charAt(right))

{

return false;

}

left = left + 1;

right = right - 1;

}

return true;

}

//Driver code

public static void main(String[] arg) {

String[] testCase = {

"RACEACAR",

"A",

"ABCDEFGFEDCBA",

"ABC",

"ABCBA",

"ABBA",

"RACEACAR"

};

for (int k = 0; k < testCase.length; k++) {

System.out.println("Test Case #" + (k + 1));

System.out.println(PrintHyphens.repeat("-", 100));

System.out.println("\tThe input string is " + testCase[k] + "' and the length of the string is " + testCase[k].length() + ".");

System.out.println("\n\tIs it a palindrome?..... " + isPalindrome(testCase[k]));

System.out.println(PrintHyphens.repeat("-", 100));

}

}

}

#### Solution summary

We initialize two pointers and move them from opposite ends. The first pointer starts moving toward the middle from the very first element of our string, while the second pointer starts moving toward the middle from the very last element of our string. This allows us to compare these elements at every single instance to find a non-matching pair, and if they reach the middle of the string without encountering any non-matching pair, that means we indeed traverse a palindrome.

#### Time complexity

The time complexity is *O*(*n*) where *n* is the number of characters present in the string.

#### Space complexity

The space complexity is *O*(1) because we use constant space to store two indices.

# Solution: Sum of Three Values

Let's solve the Sum of Three Values problem using the Two Pointers pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAW3LvvJrQy#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAW3LvvJrQy#Solution)
  + [Naive approach](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAW3LvvJrQy#Naive-approach)
  + [Optimal approach using two pointers](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAW3LvvJrQy#Optimal-approach-using-two-pointers)
    - [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAW3LvvJrQy#Solution-summary)
    - [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAW3LvvJrQy#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAW3LvvJrQy#Space-complexity)

## Statement

Given an array of integers, nums, and an integer value, target, determine if there are any three integers in nums whose sum equals the target. Return TRUE if three such integers are found in the array. Otherwise, return FALSE.

**Constraints:**

* 33 ≤≤ nums.length ≤≤ 10001000
* −103−103 ≤≤ nums[i] ≤≤ 103103
* −103−103 ≤≤ target ≤≤ 103103

## Solution

So far, you’ve probably brainstormed some approaches on how to solve this problem. Let’s explore some of these approaches and figure out which one to follow while considering time complexity and any implementation constraints.

### Naive approach

The naive approach to solving this problem is to use three nested loops. Each nested loop starts at the index greater than the parent loop. For example, if we use the variables i, j, and k in the loops, j will start from i + 1 and k will start from j+1. This approach will check all possible triples to see if they sum up to the required target value.

We have the solution but at what cost? Since we’re using three nested loops, the overall time complexity becomes �(�3)*O*(*n*3). Since we aren’t using any extra space and the entire process runs in place, the space complexity is �(1)*O*(1).

### Optimal approach using two pointers

We use a slightly modified version of the two pointers pattern. In the classic version of this pattern, two pointers are used to traverse a sorted list, one from each end. This is how we would find two numbers that summed up to a target value. Because we need to find three numbers, we run a nested version of this solution. We start from one end of the array, fix one value at a time, and run the two pointers algorithm on the remaining portion of the array, looking for the remaining target value (after subtracting the value we have fixed).

The first step is to sort the input array. We start by iterating over each element in the array, nums, using the index i. Then, in a nested loop, we use two pointers, low and high, to iterate over the array to find a triple so that nums[low] + nums[high] + nums[i] is equal to the required target.

* If the sum of the triple is equal to the target, it’s the solution.
* Else, if the sum of the three numbers is greater than the target, we move our high pointer backward.
* Else, if the sum of the three numbers is less than the target, we move our low pointer ahead.

We repeat this process by fixing the index i at each location in the array, one at a time.

Let’s see how this algorithm works for the above example:

class SumOfThree {

public static boolean findSumOfThree(int nums[], int target) {

// Sorting the input list

Arrays.sort(nums);

int low, high, triples;

// Fix one element at a time and find the other two

for (int i = 0; i < nums.length - 2; i++) {

// Set the indexes of the two pointers

// Index of the first of the remaining elements

low = i + 1;

// Last index

high = nums.length - 1;

while (low < high) {

// Check if the sum of the triple is equal to the sum

triples = nums[i] + nums[low] + nums[high];

// Found a triple whose sum equals the target

if (triples == target) {

return true;

}

// Move low pointer forward if the triple sum is less

// than the required sum

else if (triples < target) {

low++;

} // Move the high pointer backwards if the triple

// sum is greater than the required sum

else {

high--;

}

}

}

return false;

}

public static void main(String[] args) {

int[][] numsList = {{3, 7, 1, 2, 8, 4, 5},

{-1, 2, 1, -4, 5, -3},

{2, 3, 4, 1, 7, 9},

{1, -1, 0},

{2, 4, 2, 7, 6, 3, 1}};

int[][] testList = {{10, 20, 21},

{-8, 0, 7},

{8, 10, 20},

{1, -1, 0},

{8, 11, 15}};

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

System.out.print(i+1);

System.out.println(".\tInput array: " + Arrays.toString(numsList[i]));

for(int j=0 ;j<testList[i].length; j++){

if (findSumOfThree(numsList[i], testList[i][j])) {

System.out.println("\tSum for " + testList[i][j] + " exists ");

} else {

System.out.println("\tSum for " + testList[i][j] + " does not exist ");

}

}

System.out.println(PrintHyphens.repeat("-", 100));

}

}

}

#### Solution summary

1. Sort the array in ascending order.
2. Loop through the entire array and set up two pointers (low and high) on every iteration.
3. The low pointer is set to the current loop index + 1, and high is set to the last index of the array.
4. Calculate the sum of array elements pointed to by the current loop index, and the low and high pointers.
5. If the sum is equal to the target, return TRUE.
6. If the sum is greater than the target, move the high pointer backward.
7. If the sum is less than the target, move the low pointer forward.
8. Repeat until the loop has processed the entire array.
9. If after processing the entire array, we don’t find a triple that matches our requirement, we return FALSE.

#### Time complexity

* Sorting the array:  *O*(*nlog*(*n*))
* Nested loop to find the triplet:  *O*(*n*2)

The total time complexity of this solution is *O*(*nlog*(*n*)+*n*2).

#### Space complexity

The space complexity of this solution is *O*(1) because we only use the space needed to store two index values.

# Remove nth Node from End of List

Try to solve the Remove nth Node From End of List problem.

**We'll cover the following**

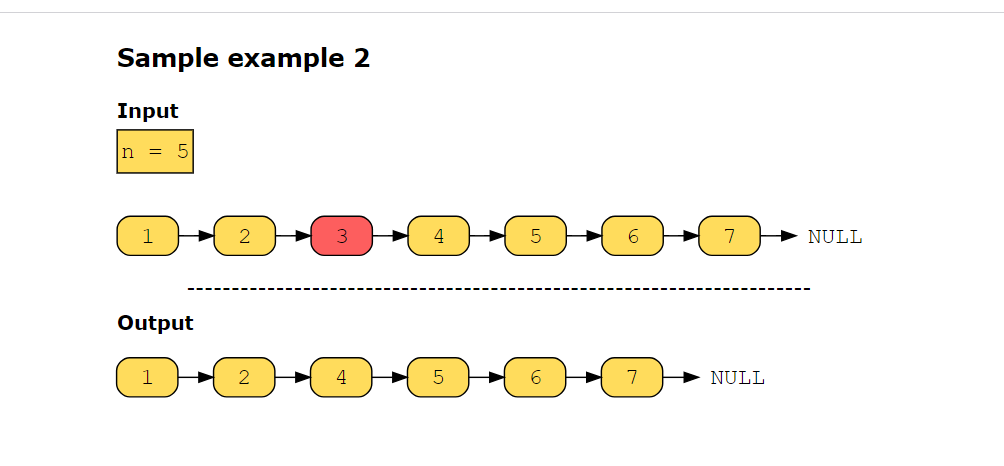
* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nGR0wXjPYw#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nGR0wXjPYw#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nGR0wXjPYw#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nGR0wXjPYw#Figure-it-out!)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nGR0wXjPYw#Try-it-yourself)

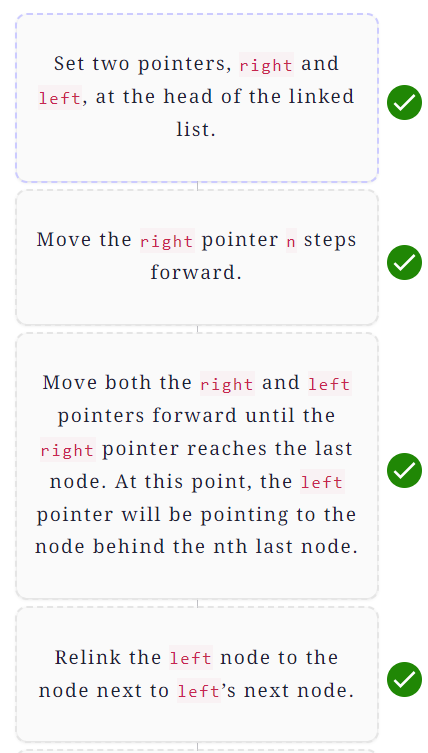
## Statement

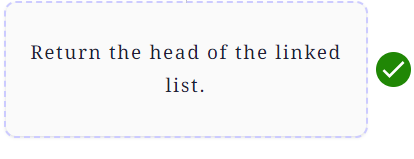
Given a singly linked list, remove the ��ℎ*nth* node from the end of the list and return its head.

**Constraints:**

* The number of nodes in the list is k.
* 1≤1≤ k ≤104≤104
* −103≤−103≤ Node.value ≤103≤103
* 1≤1≤ n ≤≤ k







# Solution: Remove nth Node from End of List

Let's solve the Remove nth Node from End of List problem using the Two Pointers pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/R8XOM40X23w#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/R8XOM40X23w#Solution)
  + [Naive approach](https://www.educative.io/courses/grokking-coding-interview-patterns-java/R8XOM40X23w#Naive-approach)
  + [Optimized approach using two pointers](https://www.educative.io/courses/grokking-coding-interview-patterns-java/R8XOM40X23w#Optimized-approach-using-two-pointers)
    - [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/R8XOM40X23w#Solution-summary)
    - [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/R8XOM40X23w#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/R8XOM40X23w#Space-complexity)

## Statement

Given a singly linked list, remove the *nth* node from the end of the list and return its head.

**Constraints:**

* The number of nodes in the list is k.
* 1≤1≤ k ≤104≤104
* −103≤−103≤ Node.value ≤103≤103
* 1≤1≤ n ≤≤ k

## Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let’s explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

### Naive approach

The naive approach calculates the length of the linked list by traversing the complete list. Then, we set a pointer, say ptr, at the start of the list and move it through the list till it reaches the (*k*−*n*−1)*th* node. The ptr pointer now points to the node before the target node, i.e., the *nth* last node. Save the next node of the ptr in a temporary pointer. Relink the ptr node to the node next to ptr ’s next node. Delete the node pointed by the temporary pointer. By doing so, the *nth* last node will be removed. However, this approach traverses the linked list twice. Let’s see if we can use the two pointers pattern to implement our solution in a single pass.

### Optimized approach using two pointers

Two pointers, left and right, are set at the head node. Move the right pointer n steps forward. After doing that, both pointers are exactly separated by n nodes apart. Start moving both pointers forward until the right pointer reaches the last node. At this point, the left pointer will be pointing to the node before the target node, i.e., the *nth* last node. We relink the left node to the node next to left pointer’s next node.

If the right pointer reaches NULL while moving it n steps forward, it means that the head node should be removed. We return the head's next node.

Let’s look at the following illustration to get a better understanding of the solution:

#### Solution summary

1. Two pointers, right and left, are set at the head node.
2. Move the right pointer n steps forward.
3. If right reaches NULL, return head's next node.
4. Move both right and left pointers forward till right reaches the last node.
5. Relink the left node to the node at left's next to the next node.
6. Return head.

#### Time complexity

The time complexity is *O*(*n*), where *n* is the number of nodes in the linked list.

#### Space complexity

The space complexity is *O*(1) because we use constant space to store two pointers.

# Sort Colors

Try to solve the Sort Colors problem.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMPWqj1KB0z#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMPWqj1KB0z#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMPWqj1KB0z#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMPWqj1KB0z#Figure-it-out!)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMPWqj1KB0z#Try-it-yourself)

## Statement

Given an array, colors, which contains a combination of the following three elements:

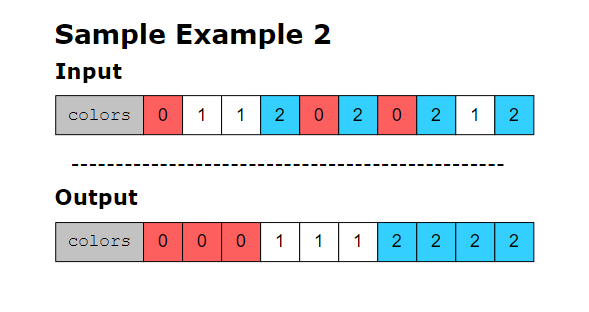
* 00 (representing red)
* 11 (representing white)
* 22 (representing blue)

Sort the array in place so that the elements of the same color are adjacent, with the colors in the order of red, white, and blue. The function should return the same array.

**Note:** The function should only return the modified colors array.

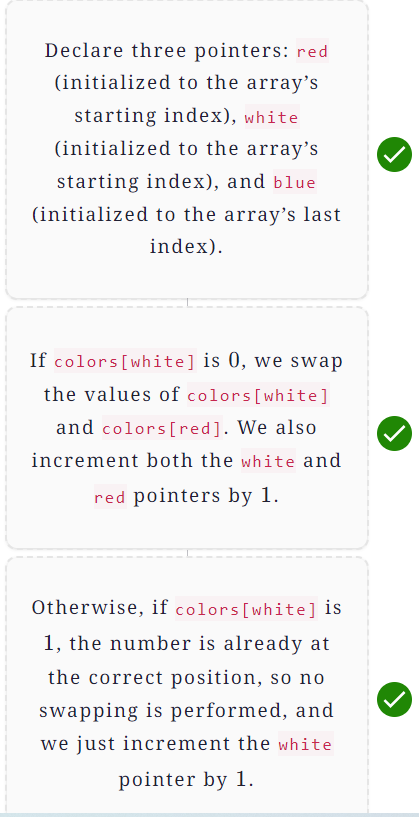
**Constraints:**

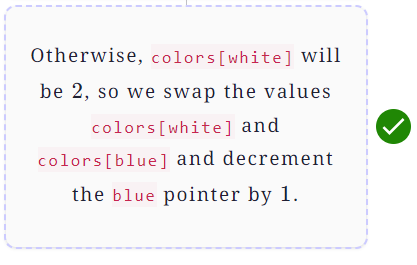
* 1≤1≤ colors.length ≤300≤300
* colors[i] can only contain 00s, 11s, or 22s.



## Figure it out!

We have a game for you to play. Rearrange the logical building blocks to develop a clearer understanding of how to solve this problem.





# Solution: Sort Colors

Let's solve the Sort Colors problem using the Two Pointers pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nEBPArRNwG#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nEBPArRNwG#Solution)
  + [Naive approach](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nEBPArRNwG#Naive-approach)
  + [Optimized approach using two pointers](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nEBPArRNwG#Optimized-approach-using-two-pointers)
    - [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nEBPArRNwG#Solution-summary)
    - [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nEBPArRNwG#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/7nEBPArRNwG#Space-complexity)

## Statement

Given an array, colors, which contains a combination of the following three elements:

* 00 (representing red)
* 11 (representing white)
* 22 (representing blue)

Sort the array in place so that the elements of the same color are adjacent, with the colors in the order of red, white, and blue. The function should return the same array.

**Constraints:**

* 1≤1≤ colors.length ≤300≤300
* colors[i] can only contain 00s, 11s, or 22s.

## Solution

So far, you’ve probably brainstormed some approaches and have an idea of how to solve this problem. Let’s explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

### Naive approach

The naive approach would be to sort the array. This would arrange the elements in the desired positions, i.e., 00s, then 11s, and last, 22s. The time complexity of this approach would be *O*(*nlog*(*n*)), which is the time required to sort the array. The space complexity of this approach would be *O*(1) since no extra space is being used.

### Optimized approach using two pointers

The idea is to use two pointers to traverse the array from either end. They keep track of the red and blue elements, respectively. In addition, we maintain another pointer to keep track of the white elements. These pointers are used to traverse the array in one pass. They are initialized as follows:

* red: This pointer will initially point to the 0�ℎ0*th* index of the array.
* white: This pointer will initially point to the 0�ℎ0*th* index of the array.
* blue: This pointer will initially point to the last index of the array.

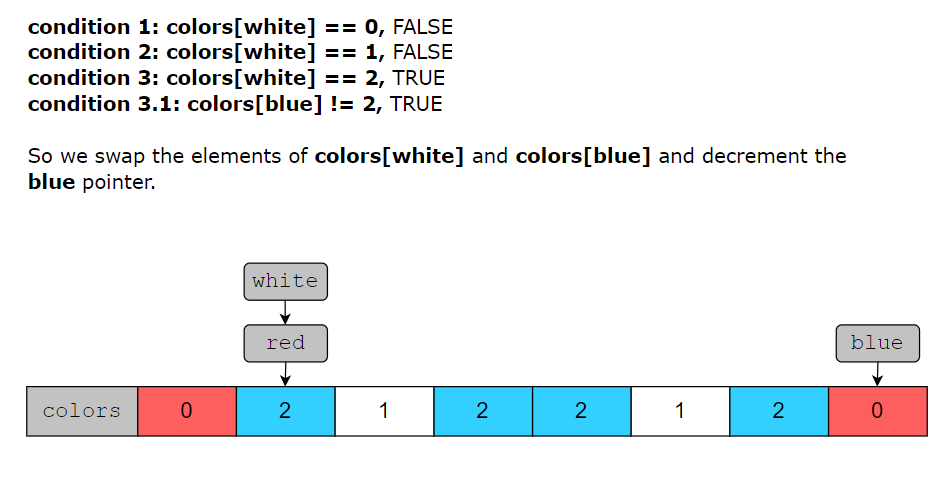
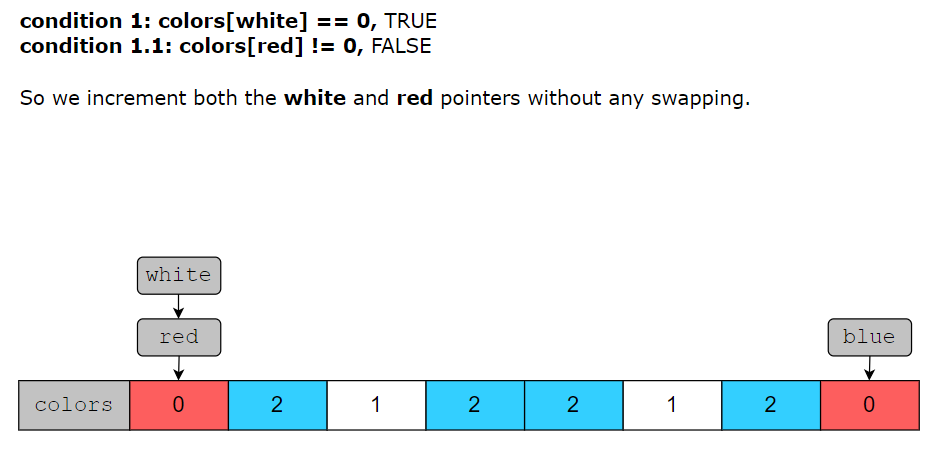
Here’s how the algorithm works:

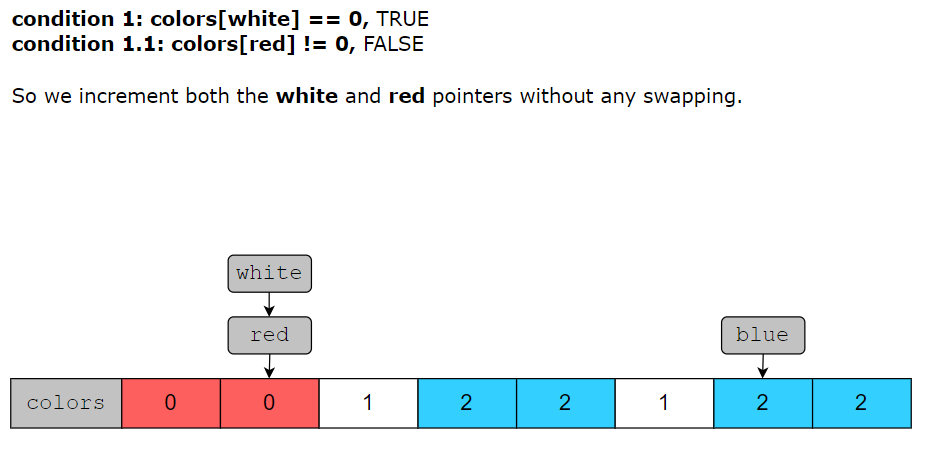
* **Condition 1:** If colors[white] is 00, the white pointer points to red. So we check a further condition:
  + **Condition 1.1:** If colors[red] is not 00, we swap the elements of colors[white] and colors[red]. Next, we move both the red and white pointers one position forward.
  + Otherwise, colors[red] will be 00, and there is no point in swapping. So, we move both the red and white pointers one position forward.
* **Condition 2:** Otherwise, if colors[white] is 11, the white pointer points to white. So we increment the white pointer by 11 to analyze the next element.
* **Condition 3:** Otherwise, the colors[white] will be 22, i.e., the white pointer points to blue. So we check two further conditions:
  + **Condition 3.1:** If colors[blue] is not 22, we swap the elements of colors[white] and colors[blue]. Next, we move the blue pointer one position backward.
  + **Condition 3.2**: Otherwise, both colors[white] and colors[blue] will be 22, so there is no point in swapping. So, we move the blue pointer one position backward.

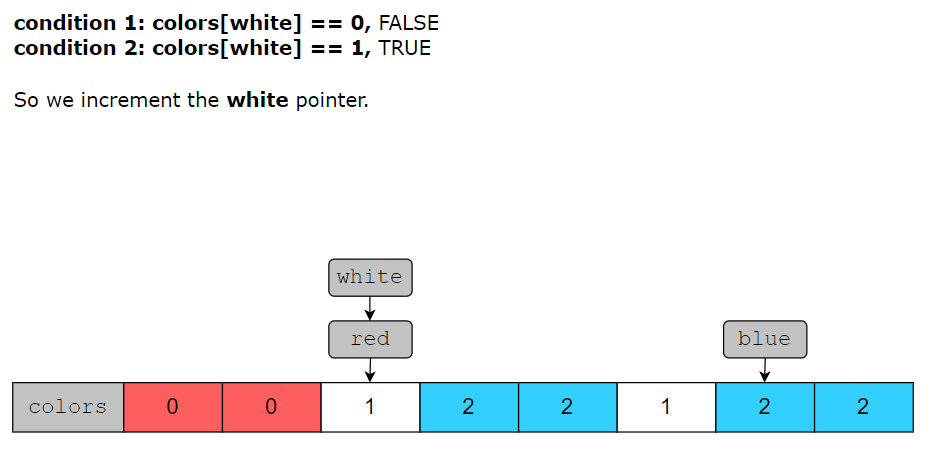
> **Note:** When we decrement the blue pointer, the white pointer remains unchanged since it has to analyze the swapped element to determine if further swapping is required with the red pointer.

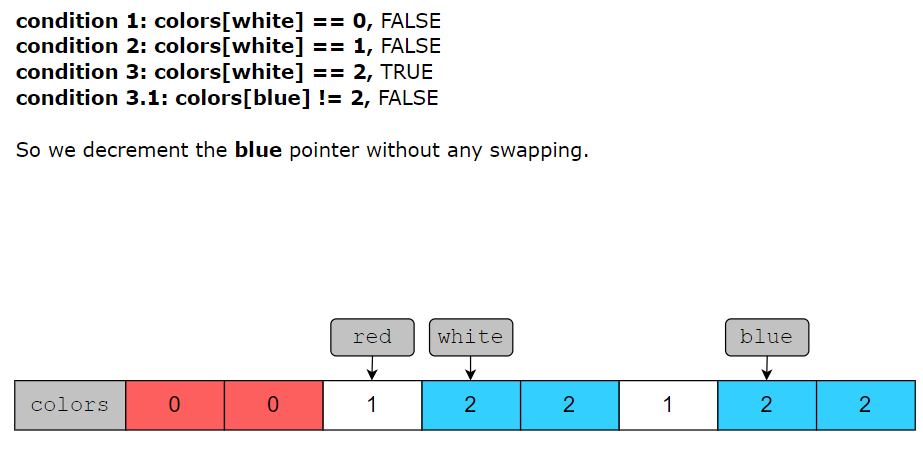
* The three steps above are repeated until the blue pointer becomes less than the white pointer, i.e., no elements are left to swap.

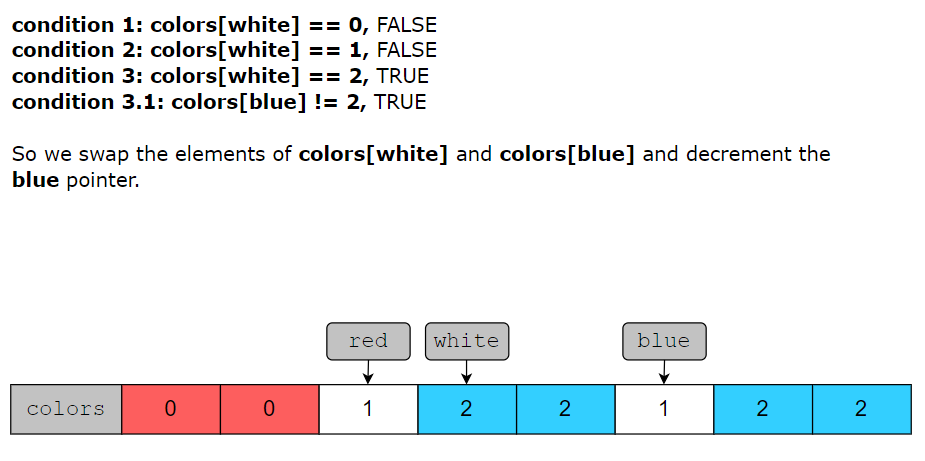
Let’s look at the following illustration to get a better understanding of the solution:

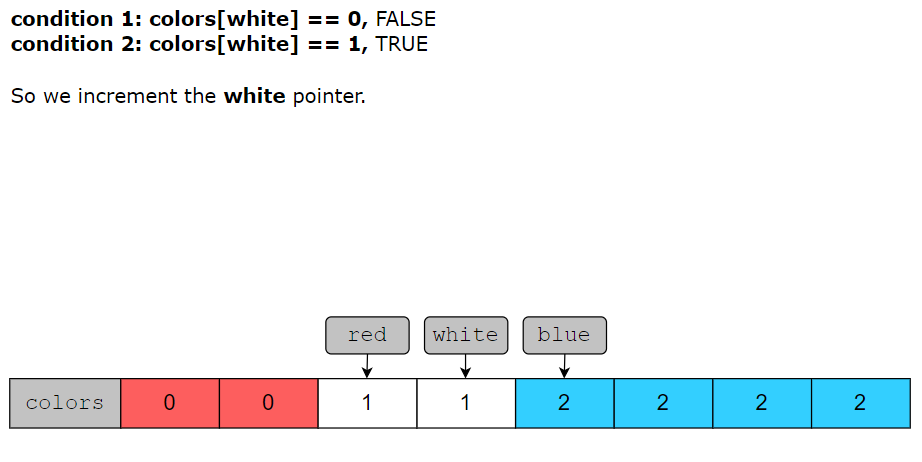


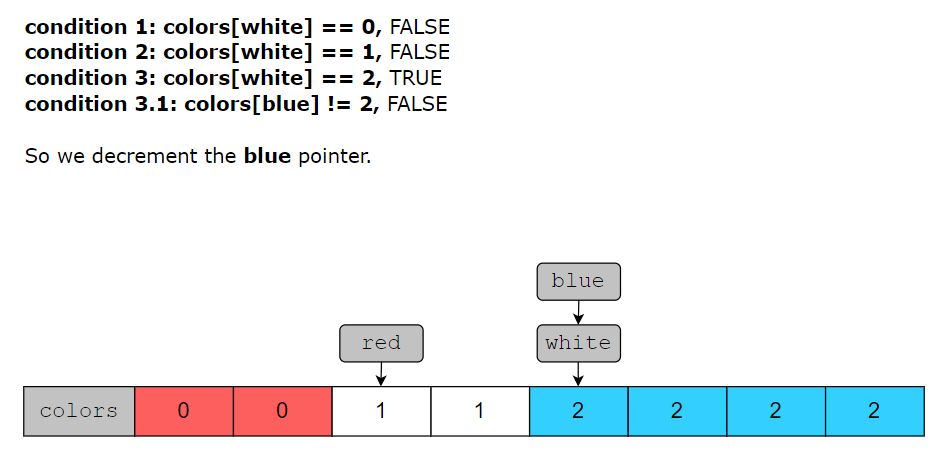


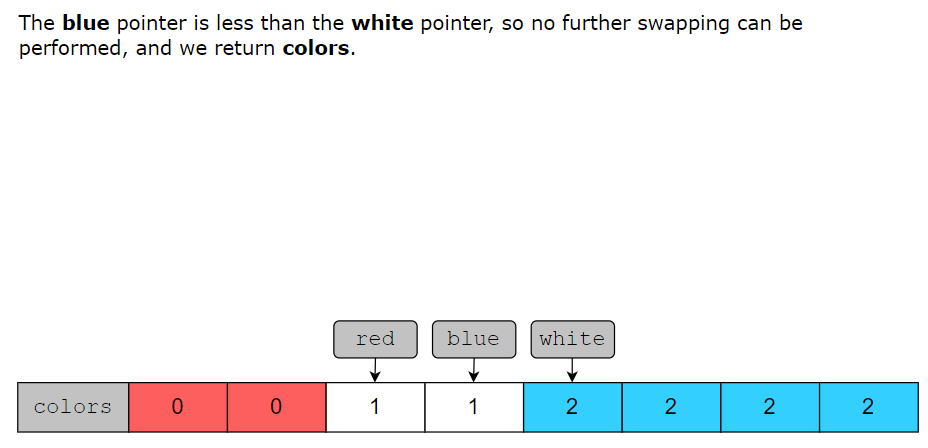












Let’s have a look at the code for the algorithm we just discussed:

import java.util.\*;

class Main {

public static int[] sortColors(int[] colors) {

int red = 0;

int white = 0;

int blue = colors.length - 1;

while (white <= blue) {

if (colors[white] == 0) {

if (colors[red] != 0) {

int temp = colors[red];

colors[red] = colors[white];

colors[white] = temp;

}

white++;

red++;

}

else if (colors[white] == 1) {

white++;

}

else {

if (colors[blue] != 2) {

int temp = colors[white];

colors[white] = colors[blue];

colors[blue] = temp;

}

blue--;

}

}

return colors;

}

// Driver code

public static void main(String[] args) {

int[][] inputs = {

{0, 1, 0},

{1, 1, 0, 2},

{2, 1, 1, 0, 0},

{2, 2, 2, 0, 1, 0},

{2, 1, 1, 0, 1, 0, 2}

};

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

System.out.println((i + 1) + ".\tcolors: " + Print.arrayToString(inputs[i]));

int[] sortedColors = sortColors(inputs[i].clone());

System.out.println("\n\tThe sorted array is: " + Print.arrayToString(sortedColors));

System.out.println(Print.repeat("-", 100));

}

}

}

#### Solution summary

To recap, the solution to this problem can be divided into five main parts:

* We traverse the array using three pointers, red, white, and blue.
* If the element pointed to by the white pointer is 00, we swap it with the element pointed to by the red pointer if it’s not pointing to 00, and increment both the red and white pointers.
* If the element pointed to by the white pointer is 11, we increment the white pointer.
* If the element pointed to by the white pointer is 22, we swap it with the element pointed to by the blue pointer if it’s not pointing to 22 and decrement the blue pointer.
* The array is sorted when the blue pointer becomes less than the white pointer.

#### Time complexity

The time complexity of this solution is *O*(*n*) since we’re only traversing the array once.

#### Space complexity

The space complexity of this solution is *O*(1) since no extra space is used.

# Reverse Words in a String

Try to solve the Reverse Words in a String problem.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N77BlmzjQ0m#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N77BlmzjQ0m#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N77BlmzjQ0m#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N77BlmzjQ0m#Figure-it-out!)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/N77BlmzjQ0m#Try-it-yourself)

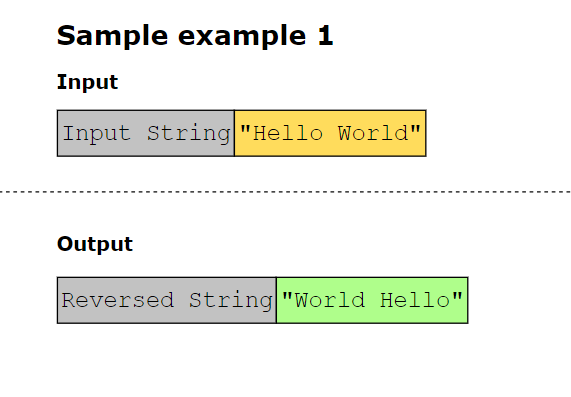
## Statement

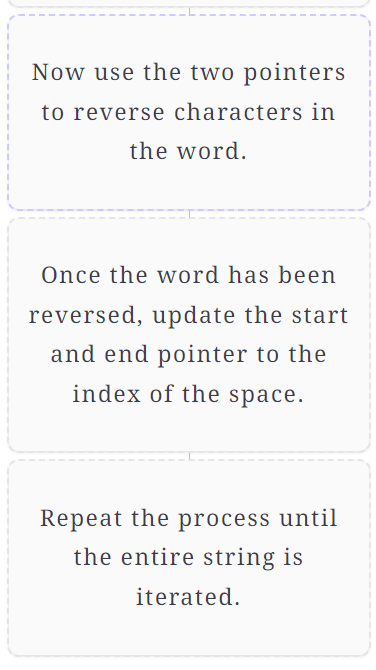
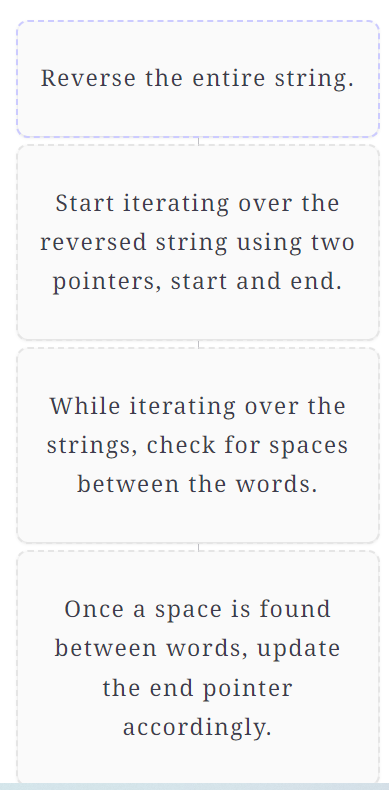
Given a sentence, reverse the order of its words without affecting the order of letters within a given word.

**Constraints:**

* The length of the sentence should be equal to or more than one character or word.
* Sentence contains English uppercase and lowercase letters, digits, and spaces.
* 11 ≤≤ sentence.length ≤≤ 104104
* The order of the letters within a word is not to be reversed.

**Note:** The input string may contain leading or trailing spaces or multiple spaces between words. The returned string, however, should only have a single space separating each word. Do not include any extra spaces.





# Solution: Reverse Words in a String

Let's solve the Reverse Words in a String problem using the Two Pointers pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMGyERPg7ME#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMGyERPg7ME#Solution)
  + [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMGyERPg7ME#Time-complexity)
  + [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RMGyERPg7ME#Space-complexity)

## Statement

Given a sentence, reverse the order of its words without affecting the order of letters within a given word.

**Constraints:**

* The length of the sentence should be equal to or more than one character or word.
* Sentence contains English uppercase and lowercase letters, digits, and spaces.
* 1 ≤≤ sentence.length ≤≤ 10^4
* The order of the letters within a word is not to be reversed.

**Note:** The input string may contain leading or trailing spaces or multiple spaces between words. The returned string, however, should only have a single space separating each word. Do not include any extra spaces.

## Solution

In this problem, we need to reverse the order of the words in a sentence without reversing the order of the characters within individual words. For this problem, we can use the two pointers solution to the string reversal problem. First, we reverse the whole sentence. Then, we use the same approach to reverse individual words within the sentence. So, we’re able to solve this problem in *O*(*n*) time and using *O*(*n*) space.

We follow these two steps:

1. Reverse the string.
2. Traverse the string, and reverse each word.

**Note**: In the second step, to identify each word, we keep traversing till a white space or end of text occurs.

class ReverseWords {

public static String reverseWords(String s) {

// We need to convert the input strings

// to arrays of characters as strings are immutable in Java

char[] s1 = s.toCharArray();

// trim spaces and convert string to string builder

String s2 = trimSpaces(s1, s1.length);

StringBuilder builder = new StringBuilder(s2);

// To reverse all words in the string, we will first reverse

// the entire string.

strRev(builder, 0, builder.length() - 1);

// Now all the words are in the desired location, but

// in reverse order: "Hello World" -> "dlroW olleH".

// Now, let's iterate the sentence and reverse each word in place.

// "dlroW olleH" -> "World Hello"

int n = builder.length();

int start = 0, end = 0;

// Find the start index of each word by detecting spaces.

while (start < n) {

// Find the end index of the word.

while (end < n && builder.charAt(end) != ' ')

++end;

// Let's call our helper function to reverse the word in-place.

strRev(builder, start, end - 1);

// moving to the next word

start = end + 1;

++end;

}

return builder.toString();

}

// A function that reverses a whole sentence character by character

public static void strRev(StringBuilder sb, int startRev, int endRev) {

// Starting from the two ends of the list, and moving

// in towards the centre of the string, swap the characters

while (startRev < endRev) {

char temp = sb.charAt(startRev); // temp store for swapping

sb.setCharAt(startRev++, sb.charAt(endRev));

sb.setCharAt(endRev--, temp);

}

}

// trim leading, trailing and multiple spaces

static String trimSpaces(char[] a, int n) {

int x = 0, y = 0;

while (y < n) {

while (y < n && a[y] == ' ') y++; // skip spaces

while (y < n && a[y] != ' ') a[x++] = a[y++]; // keep non spaces

while (y < n && a[y] == ' ') y++; // skip spaces

if (y < n) a[x++] = ' '; // keep only one space

}

return new String(a).substring(0, x);

}

public static void main(String[] args) {

String[] inputs = {"Hello World!", "We love Python.", "The quick brown fox jumped over the lazy dog.", "Hey!", "To be, or not to be", "AAAAA", "Hello World"};

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

System.out.print(i+1);

System.out.println(".\tActual string:\t\t"+ inputs[i]);

System.out.println("\tReversed String:\t"+ reverseWords(inputs[i]));

System.out.println(PrintHyphens.repeat("-", 100));

}

}

}

### Time complexity

Because the array is traversed twice, the time complexity of this solution is *O*(*n*+*n*)=*O*(*n*).

### Space complexity

The space complexity of this solution is *O*(*n*) as we use additional space to store the list of characters for reversal because strings are immutable in Java.

# Valid Palindrome II

Try to solve the Valid Palindrome II problem.

**We'll cover the following**

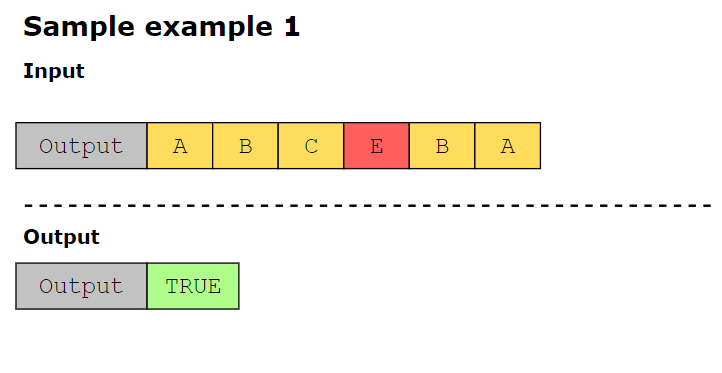
* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAWVrz2GkjG#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAWVrz2GkjG#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAWVrz2GkjG#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAWVrz2GkjG#Figure-it-out!)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/qAWVrz2GkjG#Try-it-yourself)

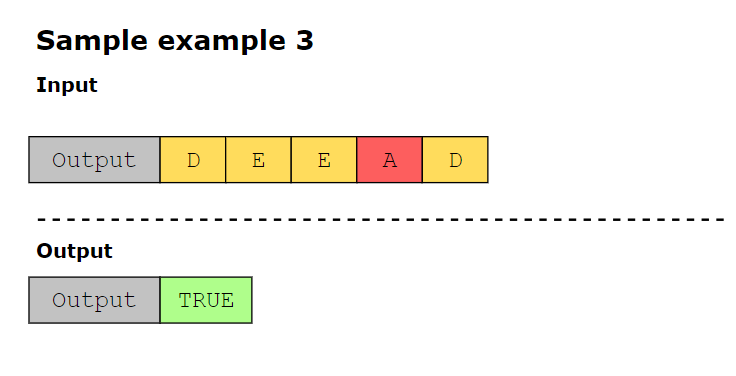
## Statement

Write a function that takes a string as input and checks whether it can be a valid palindrome by removing at most one character from it.

**Constraints:**

* 11 ≤≤ string.length ≤105≤105
* The string only consists of English letters

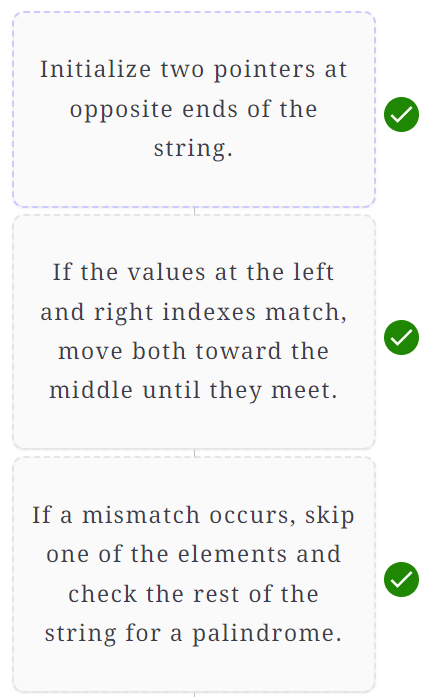


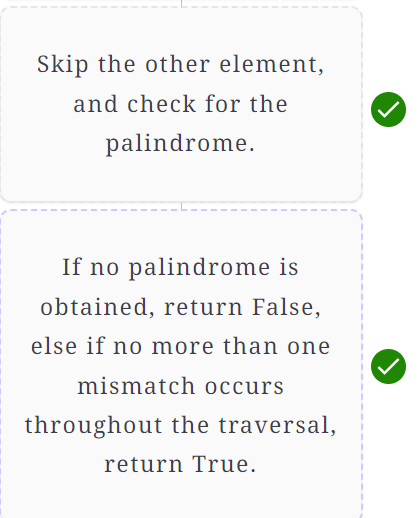


## Figure it out!

We have a game for you to play. Rearrange the logical building blocks to develop a clearer understanding on how to solve this problem.

**Note:**As an additional challenge, we have intentionally hidden the solution to this puzzle.





## Try it yourself

Implement your solution in main.java in the following coding playground. We have provided a useful code template in the other file that you may build on to solve this problem.

**Note:** We have left the solution to this challenge as an exercise for you. You may try to translate the logic of the solved puzzle into a coded solution.