

Problem Statement:

Given an integer array `nums`, find the contiguous subarray (containing at least one number) that has the largest sum and return its sum.

Example:

Input: `nums =`

You said:



no start with first. Write for all 10 below

ChatGPT said:



Let's work on all 10 problems in sequence. For each problem, I'll provide both a simple and an efficient solution, along with explanations and examples.

1. Two Sum

Problem Statement:

Given an array of integers `nums` and an integer `target`, return the indices of the two numbers such that they add up to the target. You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example:

Input: `nums = [2, 7, 11, 15]`, `target = 9`

Output: `[0, 1]`

Explanation: The numbers at indices `0` and `1` are `2` and `7`, and they add up to 9 (2 + 7 = 9).

Solution 1: Simple (Brute Force)

Approach:

Using two nested loops, we check every pair of numbers in the array to see if their sum equals the target.

Code:

java

```
public class TwoSum {
    public int[] twoSum(int[] nums, int target) {
        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[i] + nums[j] == target) {
                    return new int[]{i, j};
                }
            }
        }
        return new int[0];
    }
}
```

Time Complexity:

$O(n^2)$ due to two loops.

Solution 2: Efficient (HashMap)

Approach:

Use a HashMap to store the complement (target - current number) and check if the complement exists in the map as you iterate through the array.

Code:

java

```
import java.util.HashMap;

public class TwoSum {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[]{map.get(complement), i};
            }
            map.put(nums[i], i);
        }
        return new int[0];
    }
}
```

Time Complexity:

$O(n)$ because we only traverse the array once.

2. Maximum Subarray Sum

Problem Statement:

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

Input: `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`

Output: `6`

Explanation: The subarray `[4, -1, 2, 1]` has the largest sum of 6.

Solution 1: Simple (Brute Force)

Approach:

Check every possible subarray and calculate their sum, keeping track of the maximum sum.

Code:

java

```
public class MaxSubArray {
    public int maxSubArray(int[] nums) {
        int maxSum = Integer.MIN_VALUE;
        for (int i = 0; i < nums.length; i++) {
            int currentSum = 0;
```

```

        for (int j = i; j < nums.length; j++) {
            currentSum += nums[j];
            maxSum = Math.max(maxSum, currentSum);
        }
    }
    return maxSum;
}

```

Time Complexity:

$O(n^2)$ due to the nested loops.

Solution 2: Efficient (Kadane's Algorithm)

Approach:

Kadane's algorithm tracks the maximum subarray sum using a single loop by keeping track of the current subarray sum.

Code:

java

```

public class MaxSubArray {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];
        for (int i = 1; i < nums.length; i++) {
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }
        return maxSum;
    }
}

```

Time Complexity:

$O(n)$ since we only traverse the array once.

3. Sort 0, 1, 2

Problem Statement:

Given an array `nums` containing only `0`, `1`, and `2`, sort the array in-place without using extra space.

Example:

Input: `nums = [2, 0, 2, 1, 1, 0]`

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

Solution 1: Simple (Counting Sort)

Approach:

Count the number of `0`s, `1`s, and `2`s, then overwrite the array with the correct number of each value.

Code:

```
java
```

```
public class SortColors {
    public void sortColors(int[] nums) {
        int count0 = 0, count1 = 0, count2 = 0;
        for (int num : nums) {
            if (num == 0) count0++;
            else if (num == 1) count1++;
            else count2++;
        }

        for (int i = 0; i < nums.length; i++) {
            if (i < count0) nums[i] = 0;
            else if (i < count0 + count1) nums[i] = 1;
            else nums[i] = 2;
        }
    }
}
```

Time Complexity:

$O(n)$ as we are counting and then filling the array.

Solution 2: Efficient (Dutch National Flag Algorithm)

Approach:

Use three pointers (low, mid, high) to partition the array into sections for `0`, `1`, and `2`.

Code:

```
java
```

```
public class SortColors {
    public void sortColors(int[] nums) {
        int low = 0, mid = 0, high = nums.length - 1;
        while (mid <= high) {
            if (nums[mid] == 0) {
                swap(nums, low++, mid++);
            } else if (nums[mid] == 1) {
                mid++;
            } else {
                swap(nums, mid, high--);
            }
        }

        private void swap(int[] nums, int i, int j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
}
```

Time Complexity:

$O(n)$ with a single traversal.

4. Binary Search

Problem Statement:

Given a sorted array `nums` and a target value, return the index if the target is found. If not, return the

index where it would be if it were inserted in order.

Example:

Input: `nums = [1, 3, 5, 6]`, `target = 5`

Output: `2`

Solution 1: Simple (Iterative Binary Search)

Approach:

Using the binary search algorithm, keep dividing the array in halves to find the target.

Code:

```
java

public class BinarySearch {
    public int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return left;
    }
}
```

Time Complexity:

$O(\log n)$ due to binary search.

5. Nth Fibonacci Number

Problem Statement:

Given `n`, find the nth Fibonacci number. The sequence starts with `0, 1, 1, 2, 3, 5, 8, ...`

Example:

Input: `n = 5`

Output: `5`

Explanation: The 5th Fibonacci number is 5.

Solution 1: Simple (Recursive)

Approach:

Use recursion to compute the nth Fibonacci number.

Code:

```
java

public class Fibonacci {
    public int fib(int n) {
        if (n <= 1) return n;
        return fib(n - 1) + fib(n - 2);
    }
}
```

```
}
}
```

Time Complexity:

$O(2^n)$ due to exponential recursion.

Solution 2: Efficient (Iterative)

Approach:

Use iteration to compute the Fibonacci number with two variables tracking the last two numbers.

Code:

```
java
```

```
public class Fibonacci {
    public int fib(int n) {
        if (n <= 1) return n;
        int a = 0, b = 1;
        for (int i = 2; i <= n; i++) {
            int temp = b;
            b = a + b;
            a = temp;
        }
        return b;
    }
}
```

Time Complexity:

$O(n)$ due to single loop.

Let me know if you'd like to proceed with the remaining problems!

You said:



yes proceed

ChatGPT said:



Let's continue with the next set of problems:

6. Count Inversions

Problem Statement:

Given an array `arr[]`, count the number of inversions in the array. An inversion is a pair `(arr[i], arr[j])` such that `arr[i] > arr[j]` and `i < j`.

Example:

Input: `arr = [8, 4, 2, 1]`

Output: `6`

Explanation: The six inversions are `(8,4)`, `(8,2)`, `(8,1)`, `(4,2)`, `(4,1)`, and `(2,1)`.

Solution 1: Simple (Brute Force)

Approach:

Use two nested loops to count all pairs `(i, j)`

You said:



?

ChatGPT said:



Let's continue with the next problems.

6. Count Inversions

Problem Statement:

Given an array `arr`, count how many inversions are there in the array. Two elements `arr[i]` and `arr[j]` form an inversion if `arr[i] > arr[j]` and `i < j`.

Example:

Input: `arr = [8, 4, 2, 1]`

Output: `6`

Explanation: The six inversions are (8,4), (8,2), (8,1), (4,2), (4,1), and (2,1).

Solution 1: Simple (Brute Force)

Approach:

Compare every pair of elements to count inversions.

Code:

java

```
public class CountInversions {
    public int countInversions(int[] arr) {
        int count = 0;
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] > arr[j]) {
                    count++;
                }
            }
        }
        return count;
    }
}
```

Time Complexity:

$O(n^2)$ due to nested loops.

Solution 2: Efficient (Merge Sort-based Approach)

Approach:

This uses a modified merge sort to count inversions during the merge step.

Code:

java

```
public class CountInversions {
    public int countInversions(int[] arr) {
        return mergeSort(arr, 0, arr.length - 1);
    }

    private int mergeSort(int[] arr, int left, int right) {
        int invCount = 0;
        if (left < right) {
            int mid = (left + right) / 2;
            invCount += mergeSort(arr, left, mid);
            invCount += mergeSort(arr, mid + 1, right);
            invCount += merge(arr, left, mid, right);
        }
        return invCount;
    }

    private int merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0, invCount = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                temp[k++] = arr[j++];
                invCount += (mid + 1 - i);
            }
        }

        while (i <= mid) temp[k++] = arr[i++];
        while (j <= right) temp[k++] = arr[j++];

        System.arraycopy(temp, 0, arr, left, temp.length);
        return invCount;
    }
}
```

Time Complexity:

$O(n \log n)$ due to merge sort.

7. Reverse Linked List

Problem Statement:

Given the head of a singly linked list, reverse the list and return the reversed list.

Example:

Input: `head = [1, 2, 3, 4, 5]`

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

Solution 1: Simple (Iterative)

Approach:

Iterate through the list and reverse the pointers one by one.

Code:

```
java

class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

public class ReverseLinkedList {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;
        while (current != null) {
            ListNode nextNode = current.next;
            current.next = prev;
            prev = current;
            current = nextNode;
        }
        return prev;
    }
}
```

Time Complexity:

$O(n)$ because we traverse the list once.

Solution 2: Efficient (Recursive)

Approach:

Reverse the linked list using recursion by reversing smaller sublists.

Code:

```
java

public class ReverseLinkedList {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode newHead = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }
}
```

Time Complexity:

$O(n)$ due to recursion stack.

8. 3 Sum

Problem Statement:

Given an array `nums` of n integers, find all unique triplets in the array that give the sum of zero.

Example:

Input: `nums = [-1, 0, 1, 2, -1, -4]`

Output: `[[-1, -1, 2], [-1, 0, 1]]`

Solution 1: Simple (Brute Force)

Approach:

Check every possible triplet.

Code:

```
java
import java.util.*;

public class ThreeSum {
    public List<List<Integer>> threeSum(int[] nums) {
        Set<List<Integer>> result = new HashSet<>();
        for (int i = 0; i < nums.length - 2; i++) {
            for (int j = i + 1; j < nums.length - 1; j++) {
                for (int k = j + 1; k < nums.length; k++) {
                    if (nums[i] + nums[j] + nums[k] == 0) {
                        List<Integer> triplet = Arrays.asList(nums[i], nums[j], nums[k]);
                        Collections.sort(triplet);
                        result.add(triplet);
                    }
                }
            }
        }
        return new ArrayList<>(result);
    }
}
```

Time Complexity:

$O(n^3)$ due to triple loops.

Solution 2: Efficient (Two-pointer)

Approach:

Sort the array, and for each element, use two pointers to find the other two elements that sum to zero.

Code:

```
java
import java.util.*;

public class ThreeSum {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> result = new ArrayList<>();
        for (int i = 0; i < nums.length - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue; // Skip duplicates
            int left = i + 1, right = nums.length - 1;
            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];
                if (sum == 0) {
                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));
                    while (left < right && nums[left] == nums[left + 1]) left++;
                    while (left < right && nums[right] == nums[right - 1]) right--;
                }
            }
        }
    }
}
```

```

        left++;
        right--;
    } else if (sum < 0) {
        left++;
    } else {
        right--;
    }
}
}
return result;
}
}

```

Time Complexity:

$O(n^2)$ due to sorting and two-pointer traversal.

9. Find Missing Positive

Problem Statement:

Given an unsorted integer array, find the smallest missing positive integer.

Example:

Input: `nums = [3, 4, -1, 1]`

Output: `2`

Solution 1: Simple (Sort and Scan)

Approach:

Sort the array and scan for the first missing positive.

Code:

```

java

import java.util.*;

public class MissingPositive {
    public int firstMissingPositive(int[] nums) {
        Arrays.sort(nums);
        int missing = 1;
        for (int num : nums) {
            if (num == missing) {
                missing++;
            }
        }
        return missing;
    }
}

```

Time Complexity:

$O(n \log n)$ due to sorting.

Solution 2: Efficient (Constant Space)

Approach:

Use the array itself to mark the presence of elements by rearranging them.

Code:

```
java
```

```
public class MissingPositive {
    public int firstMissingPositive(int[] nums) {
        int n = nums.length;
        for (int i = 0; i
```

Let's finish the last two problems:

9. Find Missing Positive (continued)

Solution 2: Efficient (Constant Space)

Approach:

Use the array itself to mark the presence of elements by rearranging them. Each number should be in its correct index (e.g., `nums[i] == i + 1`), and if it's not, we swap it. After the first pass, the first index where the number does not match its expected value (`nums[i] != i + 1`) will indicate the smallest missing positive.

Code:

```
java
```

```
public class MissingPositive {
    public int firstMissingPositive(int[] nums) {
        int n = nums.length;

        for (int i = 0; i < n; i++) {
            while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] != nums[i]) {
                swap(nums, i, nums[i] - 1);
            }
        }

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

        return n + 1;
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

Time Complexity:

$O(n)$ due to the single pass and constant space.

10. Second Largest Element in Array

Problem Statement:

Given an array, find the second largest element in the array.

Example:

Input: `nums = [3, 2, 1]`

Output: `2`

Solution 1: Simple (Sort and Pick)

Approach:

Sort the array and pick the second last element. We first sort the array and return the second largest element by accessing the second last index.

Code:

```
java

import java.util.Arrays;

public class SecondLargest {
    public int findSecondLargest(int[] nums) {
        Arrays.sort(nums);
        return nums[nums.length - 2];
    }
}
```

Time Complexity:

$O(n \log n)$ due to sorting.

Solution 2: Efficient (Single Pass)

Approach:

Track the largest and second-largest elements in a single pass. If the current element is greater than the largest, update the second largest and largest. If the current element is only greater than the second largest but not the largest, update the second largest only.

Code:

```
java

public class SecondLargest {
    public int findSecondLargest(int[] nums) {
        if (nums.length < 2) {
            throw new IllegalArgumentException("Array must have at least two elements");
        }

        int first = Integer.MIN_VALUE;
        int second = Integer.MIN_VALUE;

        for (int num : nums) {
            if (num > first) {
                second = first;
                first = num;
            } else if (num > second && num != first) {
                second = num;
            }
        }

        if (second == Integer.MIN_VALUE) {
            throw new IllegalArgumentException("No second largest element found");
        }

        return second;
    }
}
```

```
}
}
```

Time Complexity:

$O(n)$ since we are traversing the array once.

Recap

- **Two Sum:** Brute force ($O(n^2)$) and efficient solution using a HashMap ($O(n)$).
- **Maximum Subarray:** Brute force ($O(n^2)$) and efficient Kadane's Algorithm ($O(n)$).
- **Sort 0, 1, 2:** Counting sort ($O(n)$) and Dutch National Flag Algorithm ($O(n)$).
- **Binary Search:** Simple iterative approach ($O(\log n)$).
- **Nth Fibonacci Number:** Recursive ($O(2^n)$) and iterative approach ($O(n)$).
- **Count Inversions:** Brute force ($O(n^2)$) and efficient merge sort-based approach ($O(n \log n)$).
- **Reverse Linked List:** Iterative ($O(n)$) and recursive approach ($O(n)$).
- **3 Sum:** Brute force ($O(n^3)$) and two-pointer approach ($O(n^2)$).
- **Find Missing Positive:** Sorting (



A network error occurred. Please check your connection and try again. If this issue persists please contact us through our help center at help.openai.com.

There was an error generating a response

 Regenerate

ChatGPT can make mistakes. Check important info.