# Question 1

**What is the total number of all possible subarrays of an array of size n?**

#### Select the correct choice:



O(n^2)



O(n^3)



O(n log n)



O(2^n)

# Answer 1

O(n^2)

# Question 2

**What is the total number of all possible subsets of an array of size n?**

#### Select the correct choice:



O(n^2)



O(n^3)



O(n log n)



O(2^n)

# Answer 2

O(2^n)

# Question 3

**For the array = [2, 6, -1, 7], which of the following statements are accurate?**

#### Select the correct choices:



There are 10 non-empty subarrays



There are 15 non-empty subarrays



There are 15 non-empty subsets



There are 10 non-empty subsets

# Answer 3

There are 10 non-empty subarray – n \* (n + 1) /2

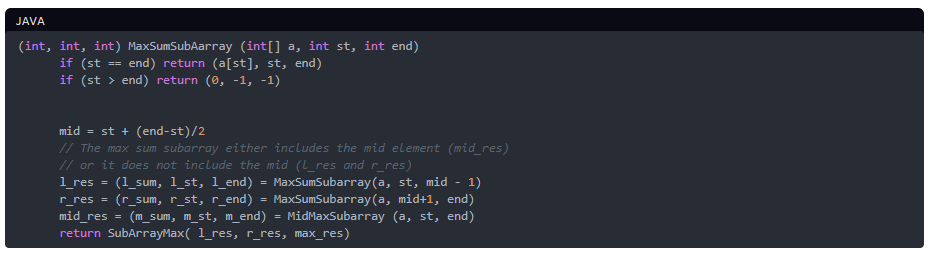
There are 15 non-empty subsets – 2 ^n

# Question 4

The n log n divide and conquer approach that leads to an O(n log n) solution to the MaxSumSubarray problem has the following overall structure.

Assume that MidMaxSubarray (a, st, end) returns the sum, start, and end of the maximum sum subarray that includes the element at index st+(end-st)/2.

Assume that SubArrayMax compares the three triples on the sum as key.



**Which of the following statements are true about this approach?**

#### Select the correct choices:



The base case of line 3 (st > end) is not needed because an empty subarray will never get created. Hint: Try and come up with a concrete example.



The base case of line 3 is needed because the recursive call on the left partition can create an empty subarray



The base case of line 3 is needed because the recursive call on the right partition can create an empty subarray



The main insight that led to this O(n log n) approach is the fact that the number of subarrays is way less than number of subsets



One main insight that led to this O(n log n) approach is that it is possible to find, in linear time, the maximum sum subarray guaranteed to include any given element (mid in our case)



The space complexity of this approach is O(n)



The space complexity of this approach is O(log n)



The height of the execution tree of this recursion is O(n)



The height of the execution tree of this recursion is O(log n)



The degree of the execution tree of this recursion is 2



The degree of the execution tree of this recursion is 3



The time complexity of this approach is O(n log n)



The time complexity of this approach is O(n^2)

# Answer 4:

The base case of line 3 is needed because the recursive call on the left partition can create an empty subarray

One main insight that led to this O(n log n) approach is that it is possible to find, in linear time, the maximum sum subarray guaranteed to include any given element (mid in our case)

The space complexity of this approach is O(log n)

The height of the execution tree of this recursion is O(log n)

The time complexity of this approach is O(n log n)

# Question 5

The overall pseudocode for Kadane’s linear time algorithm to find the maximum sum subarray is as follows:



**Which of the following are accurate statements regarding Kadane’s algorithm?**

#### Select the correct choices:



curr\_sum is the maximum possible sum for a subarray that ends at curr\_end, while including curr\_end.



curr\_sum is the maximum possible sum for a subarray of a[0...curr\_end], not necessarily ending at curr\_end



One big insight of this algorithm is that “the max sum subarray ending at index x can be computed in constant time using the max sum subarray ending at index x-1” PLUS the fact that “the global max sum subarray is ending at SOME index out of the n indices.”



One big insight of this algorithm is that Recursion is always a bad idea.



This algorithm runs in linear time



This algorithm runs in O(n log n) time but with no extra space.



One big insight of this algorithm is that the maximum sum subarray can not have any negative-sum prefix

# Answer 5

curr\_sum is the maximum possible sum for a subarray that ends at curr\_end, while including curr\_end.

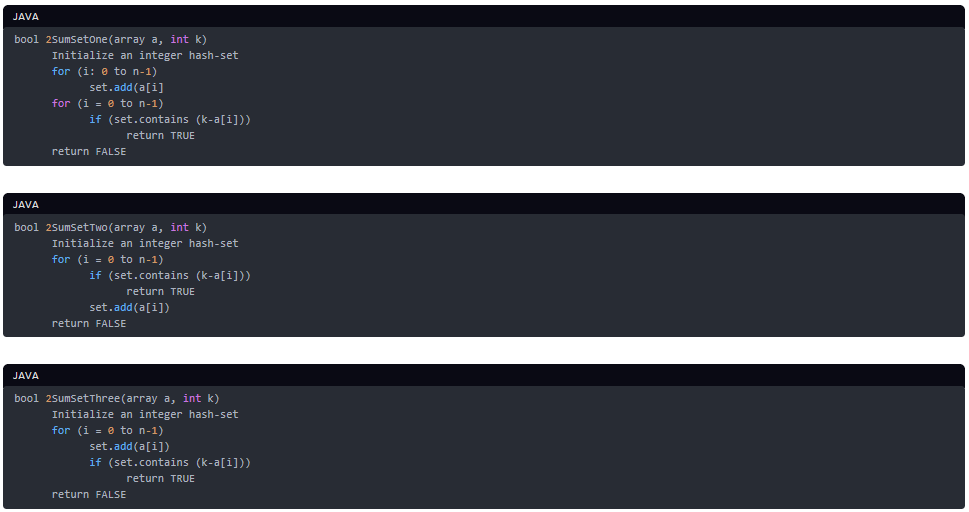
One big insight of this algorithm is that “the max sum subarray ending at index x can be computed in constant time using the max sum subarray ending at index x-1” PLUS the fact that “the global max sum subarray is ending at SOME index out of the n indices.”

This algorithm runs in linear time

One big insight of this algorithm is that the maximum sum subarray can not have any negative-sum prefix

# Question 6

Consider the following three versions of the hash\_set based approach to solving the 2 sum problem. In this problem, we want to find out if there exists two elements in an array a that add up to the integer k. Assume that a is an integer array that may contain negative integers as well as positive integers.



**Which of the following statements are accurate?**

#### Select the correct choices:



2SumSetOne works correctly when k is odd.



2SumSetTwo works correctly when k is odd.



2SumSetThree works correctly when k is odd.



2SumSetOne works correctly when k is even and the array a does not contain k/2



2SumSetTwo works correctly when k is even and the array a does not contain k/2.



2SumSetThree works correctly when k is even and the array a does not contain k/2.



2SumSetOne works correctly when k is even and the array a contains k/2 and no pair sums up to k.



2SumSetTwo works correctly when k is even and the array a contains k/2 and no pair sums up to k.



2SumSetThree works correctly when k is even and the array a contains k/2 and no pair sums up to k.



2SumSetOne works correctly regardless of k’s parity.



2SumSetTwo works correctly regardless of k’s parity.



2SumSetThree works correctly regardless of k’s parity.



If inserts in a set were free of cost and searches were mighty expensive, 2SumSetOne is preferable over 2SumSetTwo



If inserts in a set were free of cost and searches mighty expensive, 2SumSetTwo is preferable over 2SumSetOne



If searches in a set were free of cost and inserts mighty expensive, 2SumSetOne is preferable over 2SumSetTwo



If searches in a set were free of cost and if inserts were mighty expensive, 2SumSetTwo is preferable over 2SumSetOne

# Answer 6

2SumSetOne works correctly when k is odd.

2SumSetTwo works correctly when k is odd.

2SumSetThree works correctly when k is odd.

2SumSetOne works correctly when k is even and the array a does not contain k/2

2SumSetTwo works correctly when k is even and the array a does not contain k/2.

2SumSetThree works correctly when k is even and the array a does not contain k/2.

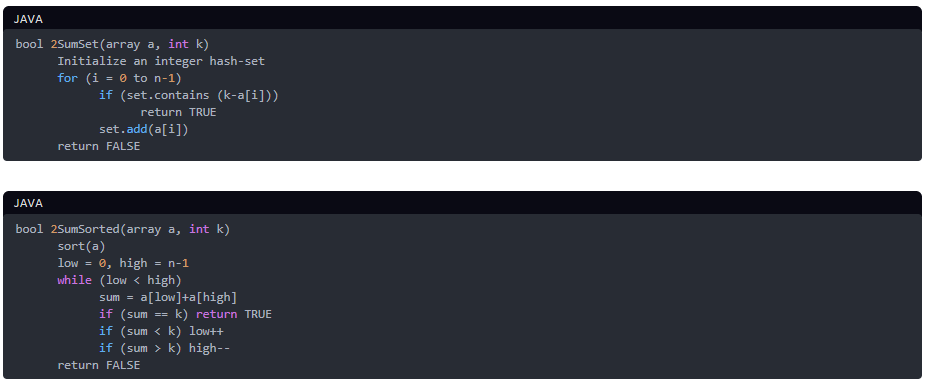
2SumSetTwo works correctly when k is even and the array a contains k/2 and no pair sums up to k.

2SumSetTwo works correctly regardless of k’s parity.

If inserts in a set were free of cost and searches were mighty expensive, 2SumSetOne is preferable over 2SumSetTwo

If searches in a set were free of cost and if inserts were mighty expensive, 2SumSetTwo is preferable over 2SumSetOne

# Question 7

Consider the following pseudocodes side by side, one for the set approach and the other for the sorting and scanning approach to solving 2 SUM.

**Mark all the accurate statements below.**

#### Select the correct choices:



Time complexity of 2SumSet is O(n)



Time complexity of 2SumSorted is O(n)



Time complexity of 2SumSet is O(n log n)



Time complexity of 2SumSorted is O(n log n), assuming that an O(n log n) sorting algorithm is used as the first step.



Time complexity of 2SumSorted can be O(n) if the input is already sorted.



Space complexity of 2SumSet is O(n)



Space complexity of 2SumSet is O(1)



Space complexity of 2SumSorted is O(n)



Space complexity of 2SumSorted is O(1), if an in place sorting algorithm is used.



Space complexity of 2SumSorted is the space complexity of the sorting algorithm used.



One main insight underlying the 2SumSorted approach is that sorting automatically means that there are fewer number of pairs that could now possibly sum up to k



One main insight underlying the 2SumSorted approach is that once you look at a pair (low, high) in a sorted array, and the sum is less than k, we know that pairing low with any other element to the right of it will be useless (since all elements left are less than a[high]). This allows the algorithm to not examine every possible pair.



The maximum number of pairs examined by 2SumSorted is O(n)



The maximum number of pairs examined by 2SumSorted is O(n^2)



The maximum number of pairs examined by 2SumSorted is O(n log n).



The main insight underlying the 2SumSet approach is that each element has one exact possible partner and we are better off just searching for that partner, which we can do in constant time using a hash set



The main insight underlying the 2SumSet approach is that hashing always leads to linear time solutions.

# Answer 7

Time complexity of 2SumSet is O(n)

Time complexity of 2SumSorted is O(n log n), assuming that an O(n log n) sorting algorithm is used as the first step.

Time complexity of 2SumSorted can be O(n) if the input is already sorted.

Space complexity of 2SumSet is O(n)

Space complexity of 2SumSorted is O(1), if an in place sorting algorithm is used.

Space complexity of 2SumSorted is the space complexity of the sorting algorithm used.

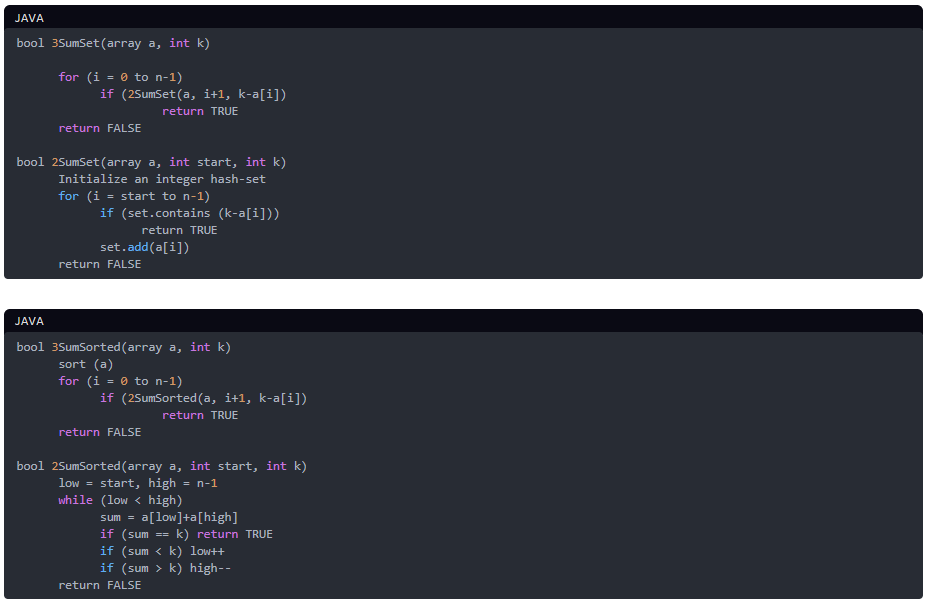
One main insight underlying the 2SumSorted approach is that once you look at a pair (low, high) in a sorted array, and the sum is less than k, we know that pairing low with any other element to the right of it will be useless (since all elements left are less than a[high]). This allows the algorithm to not examine every possible pair.

The maximum number of pairs examined by 2SumSorted is O(n)

The main insight underlying the 2SumSet approach is that each element has one exact possible partner and we are better off just searching for that partner, which we can do in constant time using a hash set

# Question 8

Consider the following two implementations of the 3 SUM problem and answer the following questions. Assume that the time complexity of sort is O(n log n) and space complexity of sort is O(1).



**What is the time complexity/space complexity of 3SumSet?**

#### Select the correct choice:



O(n)/O(n)



O(n^2)/O(1)



O(n^2)/O(n)



O(n log n)/ O(1)

# Answer 8

O(n^2)/O(n)

# Question 9