Solution 1.

a) This algorithm traverses through the array twice to find a match between two numbers. For the outer loop, it is going to be from i=0 to i=n, so a total of n times. The time complexity will be O(n) For the inner loop, the worst case upperbound would be from j=1 to j=n, in total of (n-1) times, the time complexity would be O(n). so the upperbound running time of the algorithm is $O(n) * O(n) = O(n^2)$.

$$\sum_{i=0}^{n} \sum_{j=i+1}^{n} O(1) = \sum_{i=0}^{n} \sum_{j=1}^{n} O(1) = O\left(n^2\right)$$

b) To take the lowerbound of this algorithm, for the first iteration, we take 0 to $n/4$ and for the second iteration, we take $3n/4$ to n, so we prove

$$\sum_{i=0}^{n} \sum_{j=i+1}^{n} \Omega(1) = \sum_{i=0}^{\frac{n}{4}} \sum_{j=\frac{3n}{4}}^{n} \Omega(1) = \Omega\left(n^2\right)$$

Solution 2.

Sum(): I keep track of the Sum using a variable called sum1. When pushing something into the stack, modify the push function to increment sum1 by the number you push into the stack. When popping something out of the stack, modify the pop function so that sum1 decrements by the number you pop out.

The algorithm is correct as sum1 will be updated everytime when pop and push to keep track of the total number.

When calling Sum(), it will return sum1, this operation cost O(1) time.

Min(): To keep track of the smallest value in the stack, I used a variable called min1 and created an extra empty stack called stack2. Stack2 is to keep track of all min1 values I have. The initial value of min1 is the first element is the first element pushed in the stack. The initial value stored in stack2 is the initial value of min1. When pushing an element in the stack, modify the push function to compare the min1 with the value you pushed in, if the value is smaller than min1, replace min1 with the value you pushed in and push the new min1 into stack2. Otherwise, min1 remains the same and stack2 won't do anything as well.

When popping, modify the pop function so that if the value I pop is the same as min1, then pop stack2 as well, and update the min1 with the value that's on the top of the stack.

The algorithm is correct as when pushing, min1 will always be the smallest value, and all previous min1 value is stored in stack2 and it is sorted, the biggest value is on the top and the smallest is on the bottom.

so when popping, if I pop the smallest value, stack2 will be popped and update the min1 with the previous min1 value.

When calling Min(), the function returns the value of min1, the operation cost O(1) time.

Popsmaller(e): In order to achieve Popsmaller, modify the pop function so that when poping, if the value e the Popsmaller passed in is bigger or equal to the value stored on the top of the stack, then we pop it. If e is smaller than the top of the stack, then we push e into the stack.

This algorithm is correct as it will always pop the top element until it is bigger than the value e I passed in, then e will be pushed into the stack.

When building a stack from scratch, every push takes O(1) time, the idea is to let pushes pay for pops. Instead of paying O(1) time when pushing, we pay an extra O(1) time so in total of O(2) time. So when calling Popsmaller(e), the pop operation is already been paid for by pushes, so the amortized running time for Popsmaller is O(1).

Solution 3.

a) Have two pointers pointing at the beginning and end of the sorted array. Let's say the beginning pointer is pointer A and the end pointer is pointer B. Let's store the number of combinations it has in a variable called n. The initial value for n is 0.

We put the program in a while(true) loop.

If pointer A and pointer B clashes then we exit the loop and return the value stored in n.

If the *(value pointer A holds) + (the value pointer B holds)* is bigger or equal to m, then every value after pointer A + the value pointer B stores is bigger than m.

In this case $n = n+$ is the index at pointer B $-$ the index of pointer A.Then we move pointer B to pointer A by one index

else If the value pointer A holds + the value pointer B holds is smaller than m, then move pointer A to pointer B by index 1

In this case n remain the same.

b) The algorithm is correct because this array is sorted. Assuming that the array is sorted in ascending order, the following statements are true.

(1)When A[pointer_a] + A[pointer_b] is $\geq m$, since every value after *pointer_a* is larger than itself, the sum of them with A[pointer_b] will always be larger than m.In order to find next group of valid combinations, we need to move pointer B to pointer A by one index as we need to find a smaller value.

(2)When A[pointer_a] + A[pointer_b] is $< m$, we need to find bigger value.Because this is a sorted list, moving pointer A to pointer B by one index will create a bigger value.

We continuously repeat step one and step two until pointer A and B clash with each other, which means we have considered every possible combination of the array that the addition of value in two indexes is bigger than a given m.

c) The program is going to end when two pointer hits together. Pointer A and pointer B start from the beginning and end respectively. In every iteration, one of the pointers is going to move. When they hit together the total number of iterations is going to be n. So the running time of this algorithm is going to be O(n).