Solution 1.

a) This algroithem traverse 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 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 algorithem is $O(n) * O(n) = O(n^2)$.

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

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

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

Solution 2.

Sum(): I keep track 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 poping something out of the stack, modify the pop function so that sum1 decrement by the number you pop out.

The algorithem is correct as sum1 will be updated eveyrtime when pop and push to keep track 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 create an extra empty stack called stack2. Stack2 is to keep track of all min1 value I have. The initial value of min1 is the first element is the first element pushed in the stack. The the initial value store in stack2 is the initial value of min1. When pushing 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 remain the same and stack2 won't do anything as well.

When poping, modify 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 algorithem is correct as when pushing, min1 will always the smallest value, and all previous min1 value is stored in stack2 and it is sorted, the biggest value is on the top and smallest is on the bottom.

so when poping, if I poped the smallest value, stack2 will be poped and updated 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 bigghe or euqal 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 algorithem 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 to 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 amortised runing time for Popsmaller is O(1).

Solution 3.

a) Have two pointers pointing at the begining and end of the sorted array. Let's say the begining pointer is pointer A and the end pointer is pointer B. Let's store the number of combination it has in a variable called n.

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 and the initital value is 0.

If the value pointer A + the value pointer B stores is bigger or equal than m, then everything after pointer A + the value pointer B stores is bigger than m.

In this case $n = n+$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 + the value pointer B stores is smaller than m, then move pointer A to pointer B by index 1

In this case n remian the same.

b) The algroithem is correct because this array is sorted, (1)When the value pointer A + the value pointer B stores is bigger or equal than m, everything after pointer A is going to be bigger or equal to the value pointer A pointing. So everything after pointer A + the value pointer B stores is bigger than m. In order to find every valid combination, we need to move pointer B to pinter A by one index as we already find every value that is bigger than the value pointer A + the value pointer B stores, we need to find a smaller value. Because the list is sorted, moving pointer B to pointer A by one index can find the smaller value systematiclly.

(2)When the value pointer A + the value pointer B stores is smaller than m, in order to find a value to be bigger or equal to m, we need a bigger value. Because this is a sorted list, moving pointer A to pointer B by one index will create bigger value.

We continously repeating step one and step two until pointer A and B clashes with each other, whcih means we have considered every possible combination the array that the addition of value two indexes has is bigger than a given m.

(3)