

Assignment 1

Mingyuan Ba
SID: 530157791

March 13, 2024

Problem 1

a)

The number of iterations is

$$(n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2}$$

which is bounded by n^2 , For each iteration corresponding to indices i, j , we only need to perform a single comparison operation, so the time complexity per iteration is $O(1)$. Thus, the overall time complexity is $O(n^2)$.

b)

Assume for simplicity that n is even. To lowerbound the running time, consider only comparisons made during the first half of its execution. Since this is part of the full execution, analyzing only this part gives a lower bound on the total running time. The main observation we need is that for each of the considered iterations, we make at least $\frac{n}{2}$ comparisons, allow us to lower bound the total number of comparisons made:

$$\sum_{i=0}^{n-1} (n-i-1) \geq \sum_{i=0}^{\frac{n}{2}-1} \frac{n}{2} = \frac{n^2}{4} = \Omega(n^2)$$

Problem 2

Sum()

When a new element get pushed or popped, the sum need to be updated. The **Sum** operation return the value of variable **sum**, which takes $O(1)$ time.

We modified the push and pop operations. Adding the new element to the sum takes $O(1)$ time, so push still runs in $O(1)$ time. Similarly, subtracting the removed element from the sum takes $O(1)$ time, so pop still runs in $O(1)$ time.

```
1: function NEWPUSH(e)
2:   sum  $\leftarrow$  sum + e
3:   PUSH(e)
4: end function
```

```
1: function NEWPOP
2:   e  $\leftarrow$  POP
3:   sum  $\leftarrow$  sum - e
4: end function
```

```
1: function SUM
2:   return sum
3: end function
```

Min()

To fulfill the requirement, I used a new stack (*minStack*) to keep track of the minimum value within the main stack (*mainStack*). Here's an explanation and analysis of each operation along with their time complexity:

- *NewPush*:

For each *Newpush* operation, an element x is pushed onto the *mainStack*. Simultaneously, x is compared with the top element of *minStack* (the current minimum value). If x is smaller or equal, or if *minStack* is empty, x will be pushed onto *minStack*. This ensures that the top of *minStack* always holds the minimum value of all elements in *mainStack*.

Since the *NewPush* operation itself and the conditional update of *minStack* are constant time operations, the time complexity is $O(1)$.

- *NewPop*:

When performing a pop operation, the top element is popped from *mainStack*. Then, if this popped element is equal to the top element of *minStack* (the current minimum value), it is also popped from *minStack*. This maintains the correctness of *mainStack* to reflect the new minimum value of *mainStack* after the pop operation.

As both popping the element and updating *minStack* are constant time operations, the time complexity is $O(1)$.

- *Min*:

The top element of *minStack* always represents the minimum value among all elements in *mainStack*, returning the top element of *minStack* retrieves the minimum value in $O(1)$ time.

```
1: function NEWPUSH( $x$ )
2:   mainStack.push( $x$ )
3:   if minStack.isEmpty() or  $x \leq \text{minStack.top}()$  then
4:     minStack.push( $x$ )
5:   end if
6: end function
```

```
1: function NEWPOP(void)
2:   if not mainStack.isEmpty() then
3:      $x \leftarrow \text{mainStack.pop}()$ 
4:     if  $x = \text{minStack.top}()$  then
5:       minStack.pop()
6:     end if
7:     return  $x$ 
8:   else
9:     return Error
10:  end if
11: end function
```

▷ Stack is empty

```

1: function MIN(void)
2:   if not minStack.isEmpty() then
3:     return minStack.top()
4:   else
5:     return Error
6:   end if
7: end function

```

▷ Stack is empty

PopSmaller

Algorithm 1 PopSmaller(e) Function

```

1: function POPSMALLER( $e$ )
2:   while ! isEmpty(stack) and peek(stack)  $\leq e$  do
3:     pop(stack)
4:   end while
5:   push(stack,  $e$ )
6: end function

```

Assume a sequence of n operations consists of combination of **push**, **pop**, and **PopSmaller(e)** operations. For every element, the maximum number of times it can be involved in a pop operation (whether it's a regular pop or part of **PopSmaller(e)**) is once, because once it's popped, it's not pushed back unless explicitly done by another operation. Therefore, each element is pushed once and popped at most once, leading to a total cost of $2n$ for n elements (considering the worst-case scenario where all elements are eventually popped). Thus, the total work done for n operations is $O(n)$, and the average work done per operation, or the amortized cost, is $O(n)/n = O(1)$.

Following the definition of amortized time complexity, where a sequence of n operations has $O(f(n))$ amortized time if the total work is $O(nf(n))$, we have shown that for a sequence of n operations involving **PopSmaller(e)**, the total work is $O(n)$, making the amortized time complexity $O(1)$ per operation. This satisfies the condition without increasing the complexity of other stack operations.

To elucidate further, assuming that when an element is pushed onto the stack, we charge 3 dollars, with 1 dollar allocated for the actual time consumption of the *push* operation and the remaining 2 dollars reserved for potential future operations within *PopSmaller*, such as comparisons and pops($O(1)$).

In this approach, each *push* operation incurs an amortized cost of 3 dollars, with one unit covering the actual *push* operation and the other two saved for potential future comparisons and pops within *PopSmaller*. Conversely, the pop and comparison actions within *PopSmaller* do not incur additional cost, as they are covered by the extra dollars from preceding push operations. In conclusion, as each push operation incurs an amortized cost of 3 dollars while pop and comparison operations entail no additional cost, it can be concluded that **PopSmaller** is effectively amortized $O(1)$.

Problem 3

a)

```
1: function A( $B, m$ )
2:    $n \leftarrow \text{sizeof}(B)$ 
3:    $j \leftarrow n - 1$ 
4:    $i \leftarrow 0$ 
5:    $\text{counts} \leftarrow 0$ 
6:   while  $i < j$  do
7:     if  $B[i] + B[j] \geq m$  then
8:        $\text{counts} \leftarrow \text{counts} + (j - i)$ 
9:        $j \leftarrow j - 1$ 
10:    else
11:       $i \leftarrow i + 1$ 
12:    end if
13:  end while
14:  return  $\text{counts}$ 
15: end function
```

b)

Theorem: For B, m given above, the algorithm A should correctly calculate the number of index pairs i, j such that $B[i] + B[j] \geq m$ for all $i < j$.

Base Case:

$n = 2$, B has only 2 element $B[0], B[1]$. The algorithm A compares $B[0] + B[1]$ with m . If $B[0] + B[1] \geq m$, A sets counts to 1 which is correct. Else, there is no valid index pair.

Inductive Step:

Assume that for any array B of length k algorithm A works correctly. We need to prove that for an array B of length $k + 1$, the algorithm A can also correctly calculate the number of satisfying index pairs.

Proof:

The algorithm A starts checking from $i = 0$ and $j = k$.

- If $B[0] + B[k] \geq m$, given the sorted property, all elements between $B[0]$ and $B[k]$ also sum up to be greater than or equal to m . $k - i$ correctly counts the number of pairs with $B[k]$. Then, algorithm decrease j by 1 (which is $k - 1$ now). Based on the assumption, the algorithm has already calculated all valid index pairs within the indexes from 0 to $k - 1$. Thus, algorithm finds all valid index pairs for an array B of length $k + 1$.
- If $B[0] + B[k] < m$, the algorithm increments i by 1 in a loop until $B[i] + B[k] \geq m$ is satisfied. At this point, $k - i$ is added to the counts, or the loop continues until $i = j$ (indicating there are no valid index pairs). In both cases, we have found all valid pairs involving $B[k]$. Based on the assumption, we have found all valid pair involving $B[k-1]$. Thus the algorithm finds all valid index pairs for an array B of length $k + 1$.

Conclusion

Therefore, by induction, we conclude that for any sorted array B of length n and any positive integer m , the algorithm $A(B, m)$ correctly calculates the number of index pairs (i, j) such that $B[i] + B[j] \geq m$.

c)

Line 2-5, 7, 11, 14 all consist of assignments, simple comparisons, simple math operations and the return statements return booleans, all of which takes $O(1)$ time. Each iteration of the while loop on *line 6-13* perform $O(1)$ time operations. And either i is incremented by 1 (*line 11*), or j is decremented by 1 (*line 9*). Hence, each iteration decreases the distance between i and j by 1. Given that the initial distance between i and j is $n - 1$ (where n is the length of the array B), the loop can run at most $n - 1$ times. Thus, the loop takes at most $O(1) \cdot O(n)$ or $O(n)$ time. Since we saw that the operations outside the loop all take constant time, the loop dominates the running time, which comes down to $O(n)$ time in total, as required.