CSCB36 A1: Nena Harsch and Natalia Tabja

Question 1 (written by Natalia, verified by Nena)

We have the following conditions from the pseudocode:

Line 4: if A[n - j + 1] $\neq j$ then return

For this condition to avoid an early return, we must have $A[n-j+1] = j$. This is equivalent to having the array in reverse order, where:

$$A = [n, n-1, n-2, \dots, 2, 1]$$

Line 5: if (A[i] $\neq n - i + 1$) or $(A[1] + A[2] = 2n - 1)$ then return

For the first part of this condition to avoid an early return, $A[i]$ must satisfy $A[i] = n - i + 1$ for all i, which is the same condition as in **Line 4**. This also requires that the array remains in reverse order.

However, for the second part of this condition, we need to ensure that $A[1] + A[2] \neq 2n - 1$ to avoid an early return. In a reverse-ordered array, $A[1] = n$ and $A[2] = n - 1$, which leads to:

$$A[1] + A[2] = n + (n - 1) = 2n - 1$$

This always triggers an early return due to the second condition in **Line 5**. Hence, there is no way to construct an array that will avoid all returns. This early return is inevitable for any reverse-ordered array.

During the first iteration of the outer loop, the inner loop will run n times, checking each element of the array. However, once the inner loop completes, the condition in **Line 5** will evaluate to true because $A[1] + A[2] = 2n - 1$, and the function will return. Therefore, in this case, the function performs n iterations of the inner loop before returning, which represents the worst-case scenario.

For any other array that is *not* reverse-ordered, the function will return even earlier, as the condition in **Line 4** will likely be violated during the first few iterations of the inner loop. This makes the reverse-ordered array the worst possible case, where the function proceeds furthest before returning.

Time Complexity:

In the worst-case scenario, where the array is in reverse order, the inner loop runs n times for the first iteration of the outer loop, and then the function returns due to the condition in **Line 5**. Each iteration of the loop involves constant-time operations (comparisons and assignments), so the time complexity for the worst-case scenario is:

$$T(n) = O(n)$$

Since the function performs at most n iterations before returning, the worst-case time complexity is linear. Additionally, because the function must perform at least n operations in the worst case, we can also state that:

$$T(n) = \Omega(n)$$

Thus, we conclude that the time complexity for the worst-case scenario is:

$$T(n) = \Theta(n)$$

Question 2 (written by Nena, verified by Natalia)

a. For this question, the pseudocode is defined in Algorithm 1 and 2:

Algorithm 2 explanation:

General note: The functions *insert_into_max* and *insert_into_min* are both insert functions for heap with the same idea as insert(A, x), just the names help the reader see what kind of heap we are inserting into" max or min heap.

Lines 4-5: Here we are creating two heaps: one to store the higher half of the numbers and one to store the lower half of the numbers. We will make low a max heap so that the root is the highest of the low numbers. We will make high a min heap so that the root is the lowest of the high numbers. We store the first number in array A in the low heap. If i is even, then each heap should have an equal amount of nodes. If i is odd, the the low heap should have one more node than the high heap (Note: could be interchanged, but must stick with it the entire code).

Line 7: This loop will iterate through the rest of the numbers in the prefix and put them in the right heap. If we only have one value in the array A, then this for loop is skipped and goes right to line 23.

Lines 8-12: For each value in array A, we will first assign it to low or high heap. We will check if the value is greater than the root of the low heap (which is stored at index 0), and if it is we will insert it into the high heap. Else, we will insert it into the low heap.

Lines 14-20: Here we will balance the heaps. This will ensure that low's root and high's root will be close or contain the median since low's root is the max of the lower half of the numbers and high's root is the min of the upper half of the numbers. If the low heap has too many numbers, then we will remove the root (which is the max of the heap) and move it to the high heap. On the contrary, if the high heap has too many numbers, then we will remove the root (which is the min of the heap) and move it to the low heap.

Lines 23-27: If the heap only contains one item (high.heapsize == 0) or the prefix is an odd length, then the median will be contained at the root of the low heap (low[0]). Else, if the heap is even, we need to take the mean of low heap's and high heap's roots.

Argument: This algorithm is correct because it creates two heaps that contain the lower half of the numbers and the upper half of the numbers in the given prefix. Heaps allow us to dynamically add values and maintain that they are balanced. The max heap (low) allows quick access to the largest element in the lower half. The min heap (high) allows quick access to the smallest element in the upper half. Therefore, we can calculate the median. In even cases, we take the average of the roots. In the odd case, we just take the root of the heap with one node greater (in this case the low heap).

b. First we find the runtime for the algorithm Median. In the worst case, we know that inserting into a heap, extracting the min or max node of the heap are bounded by $\Theta(\log_2 n)$. These are found in lines 9, 11, 15, 16, 18, 19. Finding the heapsize and extracting the max and min are bounded by constant time $\Theta(1)$. Therefore, if we put it all together, the algorithm Median(A, i) in the worst case can be defined as $f(n) = (i-2)(\log_2 n + \log_2 n + \log_2 n + constants)$. If we find Theta, we can simplify this to $\Theta(\log_2)$. We will run the algorithm Median n times because there are n prefixes for an array A of size n. Assigning each predixes median to the ith index of array M will be constant time. Therefore, the whole algorithm that computes the median of every prefix of an input array A will run in time $\Theta(n \log_2)$

Algorithm 1 COMPUTE-PREFIX-MEDIANS(A, M)

```
Input: Input array $A$, Output array $M$
Output: All prefix's medians stored in output array $M$

for i = 1 to n do
    M[i] = MEDIAN(A, i)
end for
```

Algorithm 2 MEDIAN(A, i)

```
Input: Array $A$, index of end of prefix $i$
Output: Median of the prefix of length $i$ in array $A$

low = [A[1]]
high = [ ]

for j = 2 to i do
    if A[j] > low[0] then
        insert_into_min(high, A[j])
    else
        insert_into_max(low, A[j])
    end if

    if low.heapsize > high.heapsize + 1 then
        max = extract_max(low)
        insert_into_min(high, max)
    else if high.heapsize > low.heapsize then
        min = extract_min(high)
        insert_into_max(low, min)
    end if
end for

if high.heapsize == 0 or i % 2 == 1 then
    return low[0]
else
    return $\frac{low[0]+high[0]}{2}$
end if
```