Noah Hayek (27080409)

ASSIGNMENT 2 - COMP 352

# **QA.1**

## **a**

count 🡨 0

Algorithm findSum(A[], x, left, right)

**Input:** Array *A* of size *n*, *x* is the value of the sum of two values in *A*, *left* and *right* represent the left and right indices of *A* respectively. Initially, *left* is explicitly assigned to zero.

**Output:** The two indices and each of their respective values that sum up to *x*.

**IF** (count = 0)

sort array A

left 🡨 0

right 🡨 (length of A) - 1

**END IF**

count 🡨 count +1

**IF** (left > right)

exit

**ELSE**

**IF** (A[left] + A[right] = x)

**THEN** Print “Indices ” left + “ & “ + right + “ with values ” + A[left] + “ & ” + A[right]

**return** findSum(A, x, left+1, right-1)

**END IF**

**IF** (A[left] + A[right] > x)

**return** findSum(A, x, left, right - 1)

**ELSE**

**return** findSum(A, x, left +1, right)

**END IF**

**END**

## **b**

Despite being O(n), ordering the array in an increasing order makes it easier to find the values since all elements increase in size as we move from left to right. The arranged elements in the array are then recursively added to each other in order to find the sum which two elements (if any) is equal to x.

## **c**

The algorithm is O(*n* log *n*). Sorting the algorithm will result in O(n) and searching through the array will cost O(log *n*).

## **d**

Since the Big-O of the algorithm is O(*n* log *n*), its Big-Ω is Ω(*n* log *n*).

# **QA.2**

## **a**

Algorithm findSum(A[], x)

**Input:** Array *A* (implemented as a circular array) of size *n*, *x* is the value of the sum of two values in *A*.

**Output:** The two values that sum up to *x*.

**WHILE**(queue is not empty)

**DO**

hold 🡨 dequeue()

count 🡨 (length – front + rear) % length

temp 🡨 dequeue()

**WHILE**(hold + temp != x and count > 0)

**DO**

enqueue(temp)

temp 🡨 dequeue()

count - -

**END WHILE**

**Print** hold + “ & “ + temp

**END WHILE**

**END.**

## **b**

The algorithm uses a circular array. It eliminates having to relocate empty indices in the array in order for them to be used during queue operations.

## **c**

O(n2). Since we have a two nested while loops the worst case for n\*n 🡺 n2 is O(n2).

## **d**

Since the Big-O of the algorithm is O(n2), its Big-Ω is Ω(n2).

# **QA.3**

## **a**

Algorithm rearrangeElements(A[], left, right)

**Input:** Array *A* of size *n*, *left* is the

**Output:** rearranged array A such that all even elements are stored before odd ones.

**IF** (left < right)

**IF** (A[left] % 2 = 1 && A[right] % 2 =1)

**return** rearrangeElement(A, left + 1, right )

**ELSE IF** (A[left] % 2 = 0 && A[right] % 2 = 1)

temp = A[right]

A[right] = A[left]

A[left] = temp

**return** rearrangeElement(A, left + 1, right - 1)

**END ELSE IF**

**ELSE IF** (A[left] % 2 = 0 && A[right] % 2 = 0)

**return** rearrangeElement(A, left, right - 1)

**END ELSE IF**

**ELSE**

**return return** rearrangeElement(A, left + 1, right - 1)

**END IF**

**END IF**

**END.**

The algorithm uses tail recursion to go through elements in the array. Each recursive call is O(1) and this means that the time complexity in the worst case is O(n).

## **b**

The time complexity will be the same and cannot be reduced from O(n) even if the problem allowed extra data structures to be used for storage because in the worst case of the best data structure algorithm, we still have to go through all of the array to find where the even and odd ones are and place them accordingly.

# **QA.4**

## **a**

## **b**

If the binary tree is stored in an array-list, the array-list will contain for each node the element, the parent node, the left child and the right child node.

# **QA.5**

Algorithm computeDepth(node)

**Input:** Node of a tree

**Output:** the depth of that tree.

**IF** (node = null)

return -1;

**…………………………………**

**--------------------------------------**

**………………………………..**

**END.**