Q4.2) Let A = [n,n−1,n−2,…,3,2,1] be an array where the first n positive integers are listed in decreasing order.
Determine whether Heapsort or Quicksort sorts this array faster.

Given array A = [n, n-1, n-2,…..3,2,1].

Algorithm and Explanation - Quick Sort

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The pivot is the right-most element for this question. We start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with A[i]. Otherwise, we ignore current element.

```
/* low  –> Starting index,  high  –> Ending index */
quickSort(arr[], low, high) {
   if (low < high) {
      /* pi is partitioning index, arr[pi] is now at right place */
      pi = partition(arr, low, high);
      quickSort(arr, low, pi – 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```

```
/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
pivot = arr[high];
 i = (low – 1)  // Index of smaller element and indicates the
// right position of pivot found so far
for (j = low; j <= high- 1; j++){
```

```
 // If current element is smaller than the pivot
if (arr[j] < pivot){
i++;    // increment index of smaller element
 swap arr[i] and arr[j]
    }
 }
   swap arr[i + 1] and arr[high])
return (i + 1)
}
```

In the given question, all the array elements are in decreasing order and the pivot is right-most element (1) which is the smallest element in the array. So, in this case, the top-level iteration of quickSort will require (n-1) comparisons and will split the array into two subarrays: one of size 1 and one of size (n-1). The first one is already sorted, and we apply the quickSort recursively to the second one, splitting the second one will require (n-2) comparisons. The process is repeated until all the elements are sorted in the array.

Therefore, in total, we have (n-1) + (n-2) + ... + 1 = n*(n-1) / 2 = $O(n^2)$ comparisons.

As a result, the splitting into sub-collections occurs at index n (end). This divides an n-item collection into 1 item less than the previous level and 0 items.

This method of selecting a "bad pivot" and then partitioning n-1 and 0 is performed recursively, resulting in an extremely skewed tree. This is the worst case for Quick Sort algorithm when the picked pivot is always an extreme (smallest or largest) element and the input array is sorted in ascending or descending order. The recurrence relation will be T(n) = T(n-1) + $\Theta(n)$ will result in time complexity $\Theta(n^2)$.

So, in this case of bad pivot, the running time of Quick Sort is $\Theta(n^2)$.

Algorithm and Explanation - Heap Sort

Heap Sort is a comparison-based sorting method based on the binary heap data structure. The binary heap structure allows the HeapSort algorithm to take advantage of the heap's properties:

- Every node's value must be greater (or smaller) than all values stored in its children
- It's a complete tree, which means it has the smallest possible height.

HeapSort algorithms follows the following steps,

- Build a min (or max) heap from the input array
- At this point, the smallest item is stored at the root of the heap. We'll remove the element from the root node, and store the rightmost leaf in the root node.
- Heapify the root of the tree which means arranging the nodes in the correct order so that they follow the heap property.
- Repeat steps 2 and 3 while the size of the heap is greater than 1

```
function heapify(A, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1    # left = 2*i + 1
    r = 2 * i + 2    # right = 2*i + 2
    if l < n and A[largest] < A[l]:
        largest = l
    if r < n and A[largest] < A[r]:
        largest = r
    if largest != i:
        A[i], A[largest] = A[largest], A[i]  # swap
        heapify(A, n, largest)

# procedure to sort an array of given size
function heapSort(A):
    n = len(A)
    # Building a max-heap
    for i in range(n//2 - 1, -1, -1):
        heapify(A, n, i)
    # Extract elements one by one
    for i in range(n-1, 0, -1):
        A[i], A[0] = A[0], A[i]  # swap
        heapify(A, i, 0)
```

In the given question, all the array elements are in decreasing order. In the heapify() function, we walk through the tree from top to bottom. The height of a binary tree (the root not being counted) of size n is $\log_2 n$ at most, that is, if the number of elements doubles, the tree becomes only one level deeper. When we remove the root element which is the largest element and replace it with the bottom most node which is a left, the set of elements has now N-1 elements and the tree is not a max heap. To make it a Max heap again, we need to check each node and swap the parent and child nodes accordingly. As a result, the running time of procedure heapify will take $O(\log_2 n)$ time and running time to create and build max-heap from the given array of n elements will take O(n) time. Hence, the overall time complexity of the given algorithm will take O(nlogn) in the best, worst and average case as it uses are fundamental property of binary heaps to sort an array.

Therefore, we can say that Heap Sort runs in $\Theta(nlogn)$ time in all cases, no matter what the input array is.

Hence, for the given question, where A = [n, n-1, n-2,…..3,2,1], Heap Sort works better and sorts this array faster in $\Theta(nlogn)$ time as compared to Quick Sort which takes $\Theta(n^2)$ time.