**ASSIGNMENT:**

**Implementation:**

Both the randomized\_quicksort and quicksort functions implement the Quicksort algorithm with plain changes in the way the pivot is getting selected. The pivot is found randomly from the list and then used in randomized\_quicksort. By using randint, we do not have to go through consistently bad pivot choices that may result in the worst-case time complexity. Next, split the array into elements lesser than or equal to and greater than the arr[0] pivot. This is so that the algorithm can process duplicates properly. In contrast, quicksort uses a deterministic pivot, the last element of the array or sublist in this case. During each recursive call, the pivot is placed in the middle of elements which are smaller than and larger than it respectively.

**Performance analysis:**

Quicksort is a widely used sorting algorithm that works relatively well in most cases. Its average case time complexity is O(nlog n). This explanation gives us an idea of why it is O(nlog n) and its best the best case where pivot divides the array in half still provides for a very good sorting process as every operation runs (O(log n)) and the number of elements to operate on keeps decreasing with each iteration. This type of balanced partitioning further reduces the depth of the recursion tree which in turn opens the door for sorting n number of elements and faster sorting. This partitioning step requires (O(n)) time at each level of recursion, and the depth of this tree will be (O(log n)), thus giving us time complexity similar to merge sort, that is O(n log n) in the best case.

In the best case, Quicksort runs in O( n log n), which is optimal for comparison sort algorithms. The reason pivot is an ideal choice is that when pivot divides the array into two almost-equal sub-arrays, this makes the recursion tree less deeper Partitioning is an O(log n) operation and each individual partition action is a constant time (O(n)) operation. Consequently, the full-time complexity for sorting the whole array is (O(n log n)) therefore making Quicksort very efficient for most inputs.

In the worst-case scenario, the time complexity of Quicksort is degraded to O(n^2). This occurs when the pivot consistently partitions the array poorly, for example partitions those at either end of the array and that way more than one of them ends up almost sorted, but mostly randomly ordered. In this case, the recursion tree would be complex (of the order of O(n), since the partition around a median at every level) but each partitioning step will still take O(n) time so the entire effect is an O(n^2) time complexity. One way to mitigate this problem is to choose the pivot point randomly, which will decrease the odds of hitting a worst-case.

**Impact of randomization on Quicksort:**

Randomization in Quicksort is used to avoid the worst-case time complexity of Quicksort when the pivot divides the array recursively into highly unbalanced subarrays. In the deterministic version, the pivot is usually selected as the last element and that may lead to inefficiency in sorting already sorted or reverse-sorted arrays. Quicksort, with the randomized quicksort this tends to alter that because it chooses a random pivot from our subarray which makes a high chance of not hitting the worst-case scenario. This allows the algorithm to keep its O(n log n) average-case time complexity even with arrays that could be problematic for the deterministic version.

Using a random pivot to split the array prevents it from repeatedly making bad partition choices as on average, it will divide the array more evenly. The partitioning step in both implementations runs in time linearly proportional to the number of elements, and randomization helps balance the depth of the recursion tree. It should follow that the randomized version, when taken over the randomness of pivot selection and Quickselect results in the expected running time (as seen with the empirical data at each input size or distribution, which showed close run times for both versions). Two cases in point, when the system orders 1,000 random elements, this is deterministic at 0.00121 seconds and randomized to 0.00123 seconds from the first execution of the running time code implementation.

Although the effect of this randomization is relatively small in most normal cases, it can be essential for seeding against worst-case performance within specific input patterns. It tries to make sure no input sequence can cause the worst case. Therefore, it is going to be a little bit safer and better choice, especially for large datasets where worst case time complexity could cost too much.