REPORT ON QUICK SORT VS RANDOMISED QUICK SORT:

QUICKSORT

Quicksort is a widely used sorting algorithm that follows the divide-and-conquer paradigm. It works by partitioning an array into two sub-arrays, one containing elements smaller than a pivot element, and the other containing elements greater than or equal to the pivot. It then recursively applies the same process to each sub-array until the entire array is sorted.

The basic steps of the Quicksort algorithm are:

1. Choose a pivot element from the array. This can be done in different ways, such as selecting the first or last element, or choosing a random element.
2. Partition the array into two sub-arrays based on the pivot element. All elements smaller than the pivot are placed in the left sub-array, and all elements greater than or equal to the pivot are placed in the right sub- array.
3. Recursively apply the same process to each sub-array until the entire array is sorted.

Quicksort is often used in practice because it has good average-case performance and requires only a small amount of additional memory. However, it is not stable, meaning that the relative order of equal elements may not be preserved, and it may be vulnerable to certain types of attacks if not implemented carefully.

ALGORITHM:

```
QUICKSORT (array A, start, end)
{
  If (start < end)
    {
   p = partition(A, start, end)
   QUICKSORT (A, start, p - 1)
   QUICKSORT (A, p + 1, end)
    }
}
PARTITION (array A, start, end)
```

```
{
pivot ? A[end]
i ? start-1 for j ? start
to end -1 { do if (A[j]
< pivot) { then i ? i +
1 swap A[i] with A[j]
 }}
swap A[i+1] with A[end] return
i+1
}
```

EXAMPLE:

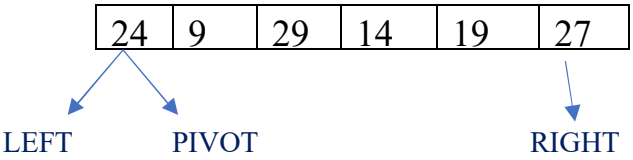
Let the elements of array are :

STEP-1:

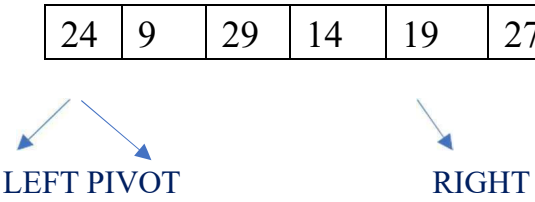
24	9	29	14	19	27

- In the given array, we consider the leftmost element as pivot.
- a[left] = 24, a[right] = 27 and a[pivot] = 24.
- Since, pivot is at left, so algorithm starts from right and move towards left.

STEP-2:



- Now, a[pivot] < a[right], so algorithm moves forward one position towards left



STEP-3:

- Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.
- Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right

19	9	29	14	24	27

STEP-4:

- Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.
- As a[pivot] > a[left], so algorithm moves one position to right

19	9	29	14	24	27

STEP-5:

19	9	29	14	24	27

- Now a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right

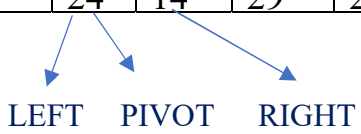
STEP-6:

- Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left

19	9	24	14	29	27

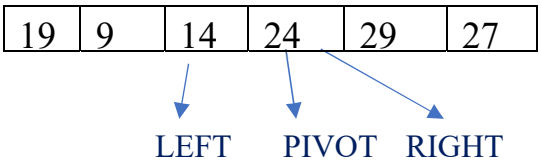
- Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24.
- As a[pivot] < a[right], so algorithm moves one position to left

19	9	24	14	29	27



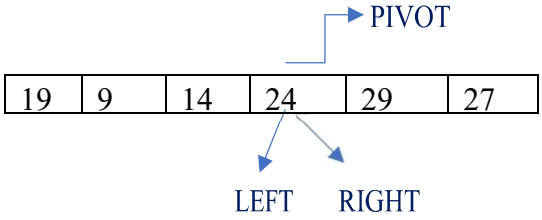
STEP-7:

- Now, a[pivot] = 24, a[left] = 24, and a[right] = 14.
- As a[pivot] > a[right] so, swap a[pivot] and a[right], now pivot is at right.



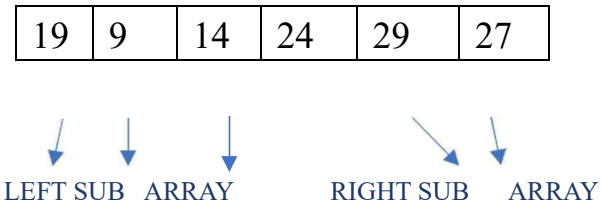
STEP-8:

- Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



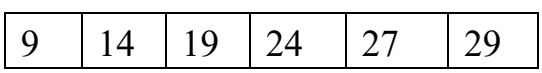
STEP-9:

- Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.
  □ Element 24, which is the pivot element is placed at its exact position.
- Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



STEP-10:

- Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be.



TIME COMPLEXITY:

Case	Time Complexity
Best Case	O(n*log n)
Average Case	O(n*log n)
Worst Case	$O(n^2)$

Randomized Quick Sort

Randomized Quick Sort is an efficient sorting algorithm that uses a divide-and-conquer approach to sort an array. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The process is then repeated recursively on each of the sub-arrays until the entire array is sorted.

The randomized version of Quick Sort improves the worst-case scenario of Quick Sort by randomly selecting a pivot element from the array, rather than always selecting the first or last element as the pivot.

This reduces the likelihood of selecting a bad pivot, such as an element that is already sorted or nearly sorted, which can result in poor performance.

The randomness of the pivot selection ensures that the worst-case scenario of Quicksort (when the pivot is either the smallest or largest element in the array) occurs with very low probability. This makes Randomized Quicksort more robust and efficient than traditional Quicksort.

The average-case time complexity of Randomized Quicksort is O(n log n), which is the same as the time complexity of Quicksort. However, the worst-case time complexity of Randomized Quicksort is still O(n^2) in the rare case when the pivot is always selected as the smallest or largest element in the array.

In order to further reduce the probability of encountering the worst-case scenario, some implementations of Randomized Quicksort use the "median-of- three" pivot

selection technique, which chooses the pivot as the median of the first, middle, and last elements of the array.

Randomized Quicksort is an in-place sorting algorithm, which means that it sorts the elements of the array in place without requiring any additional memory. This makes it very memory-efficient.

In practice, Randomized Quicksort is one of the fastest sorting algorithms, and is commonly used in many programming languages and libraries. It is often used as the default sorting algorithm for many systems and applications.

Here's how the algorithm works:

1. Choose a random element from the array to be the pivot.
2. Partition the array into two sub-arrays, one containing elements smaller than the pivot and one containing elements larger than the pivot.
3. Recursively apply the algorithm to the sub-arrays.
4. Base case: If a sub-array has length 1 or 0, it is already sorted and can be returned.
5. Combine the sorted sub-arrays by concatenating them in order.

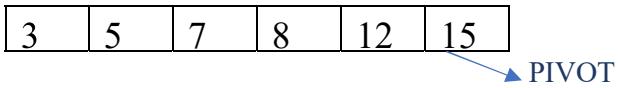
ALGORITHM:

```
partition-left(arr[], low, high)
  pivot = arr[high] i = low //
  place for swapping for j :=
  low to high – 1 do if arr[j] <=
  pivot then
       swap arr[i] with arr[j] i =
       i + 1
  swap arr[i] with arr[high] return i

partition-right(arr[], low, high)
  r = Random Number from low to high
  Swap arr[r] and arr[high] return
  partition-left(arr, low, high)

quicksort(arr[], low, high)
  if low < high
    p = partition-right(arr, low, high)
    quicksort(arr, low , p-1)
    quicksort(arr, p+1, high)
```

EXAMPLE:

3	5	7	8	12	15

STEP-1:



- Considering the worst case possible, if the random pivot chosen is also the highest index number, it compares all the other numbers and another pivot is selected.
- Since 15 is greater than all the other numbers in the list, it won't be swapped, and another pivot is chosen.

STEP-2:

- This time, if the random pivot function chooses 7 as the pivot number
- Now the pivot divides the list into half so standard quick sort is carried out usually. However, the time complexity is decreased than the worst case.
- It is to be noted that the worst case time complexity of the quick sort will always remain $O(n^2)$ but with randomizations we are decreasing the occurrences of that worst case.

TIME COMPLEXITY:

The equations for the time complexity of randomized quicksort are:

- Average-case time complexity: T(n) = O(n log n) - Worst-case time complexity: T(n) = O(n^2) where T(n) denotes the time complexity of the algorithm for an input of size n. The average-case time complexity is derived from the recurrence relation:

T(n) = T(k) + T(n-k-1) + O(n)

Where k is the index of the pivot element after partitioning, and O(n) denotes the time complexity of partitioning. The expected value of k is approximately n/2, which yields the O(n log n) time complexity.

The worst-case time complexity:

Occurs when the partitioning process always picks the largest or smallest element as the pivot. In this case, the recurrence relation becomes: T(n) = T(n-1) + O(n) which yields the O(n^2) time complexity.

DIFFERENCE:

Quick Sort	Randomized Quick Sort
Always selects first or last element as pivot	Chooses a random element as pivot
Worst-case scenario occurs when input is sorted or nearly sorted	More resilient to worst-case scenarios
Average case time complexity is O(n log n)	Average case time complexity is also O(n log n)
Worst-case time complexity is O(n^2)	Worst-case time complexity is O(n^2) but is less likely to occur
Inefficient for large input sizes or highly unbalanced data	More efficient for large input sizes or highly unbalanced data
Does not require additional space for randomization	Requires additional space for randomization
Simple to implement	Slightly more complex to implement

Conclusion:

In terms of efficiency, randomized quicksort is generally considered to be the best option between quicksort and randomized quicksort, as it has a lower probability of hitting the worst-case time complexity and is thus more efficient in practice. Overall, while both Quick Sort and Randomized Quick Sort share many similarities, the randomization step in Randomized Quick Sort makes it a more robust and efficient algorithm in practice. So Randomised Quick sort is good and efficient.