Experiment No: 2

Title: Implementation of Quick Sort algorithm using Divide & Conquer method.

Theory/Description:

The basic concept is to pick one of the elements in the array as a pivot value around which the other elements will be rearranged. Everything less than the pivot is moved left of the pivot (into the left partition). Similarly, everything greater than the pivot goes into the right partition. At this point, each partition is recursively quickly sorted.

The Quick sort algorithm is fastest when the median of the array is chosen as the pivot value. That is because the resulting partitions are of very similar size. Each partition splits itself in two and thus the base case is reached very quickly.

In practice, the Quick sort algorithm becomes very slow when the array passed to it is already close to being sorted. Because there is no efficient way for the computer to find the median element to use as the pivot, the first element of the array is used as the pivot. So when the array is almost sorted, quick sort doesn't partition it equally. Instead, the partitions are asymmetrical like in Figure. This means that one of the recursion branches is much deeper than the other, and causes execution time to go up. Thus, it is said that the more random the arrangement of the array, the faster the Quicksort Algorithm finishes.

Quick sort works by partitioning a given array A[p . . r] into two non-empty sub array A[p . . q] andA[q+1 . . r] such that every key in A[p . . q] is less than or equal to every key in A[q+1 . . r]. Then the two sub arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

```
Algorithm QuickSort(p,q)
// Sorts the elements a[p],……,a[q] which reside in the global array a[1 : n] into
ascending order.
{
    if(p <r)then          //        if there are more thanone element
    {
    //   divide P into two sub problems
         j=Partition (a,p,q+1);
         //        j is the position of the partitioningelement
    //   solve the sub problems
           QuickSort(p, j -1);
           QuickSort(j + 1, q);
     }
}
```

As a first step, Quick Sort chooses as pivot oneof the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal topivot will move toward the left and elements that are greater than or equal to pivot will move toward theright.

Partitioning the Array

Partitioning procedure rearranges the sub arrays in-place.

```
Algorithm Partition (a, m, p)
```

```
//Within a[m],a[m+1],….,a[p-1]the elements are rearranged in such a manner that if initially
t=a[m] then after completion a[q]=t for some q between m and p-1, a[k] |<=t for m<=k<q,
and a[k]>=t for q<k<p. q is returned. Set a[p]=∞.
{
    v = a[m]; i=m; j=p;
    repeat
    {
        repeati=i+1;
        until a[i]>=v;

        repeat j = j-1;
        until a[j]<=v;


       ifi< j then exchange A[i] ↔ A[j]
     } until( i>= j); a[m]=a[j];
    a[j]=v; returnj;
}
```

Partition selects the first key, a[p] as a pivot key about which the array will partitioned: Keys ≤ a[p] will be moved towards theleft.

Keys ≥ a[p] will be moved towards the right.

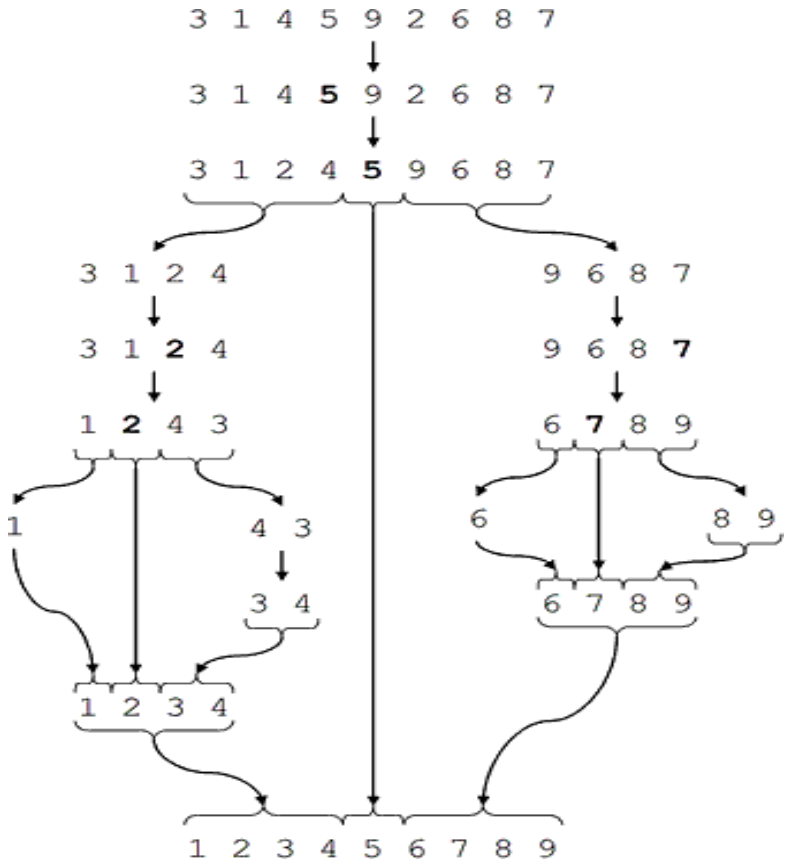

Figure The ideal Quicksort on a random array

Program Snippets:

```
voidquickSort(int numbers[], intarray_size)
{
  q_sort(numbers, 0, array_size - 1);
}


voidq_sort(int numbers[], int left, int right)
{
  int pivot, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right)
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }
    while((numbers[left] <= pivot) &&(left <right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }
  numbers[left] = pivot;
  pivot = left;
  left= l_hold;
  right = r_hold;
  if (left <pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, pivot+1, right);
}
```

Input: 13 -5 -8 15 60 17 31 47

Output:

Sorted numbersare:-8 -5 13 15 17 31 47 60

Conclusion:

Performance Analysis of Quick Sort

The running time of quick sort depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning.

A very good partition splits an array up into twoequal sized arrays. A badpartition, on other hand, splits an array up intotwoarrays of very different sizes. The worstpartition puts only one element in one array and all other elements in the other array. If thepartitioning is balanced, the Quick sort runs asymptotically as fast as merge sort. On theother hand, if partitioning is unbalanced, the Quick sort runs asymptotically as slow as insertionsort.

Best Case

The best thing that could happen in Quicksort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of thekeysin A[p . . r] every time procedure 'Partition' is called. The procedure'Partition' always split the array to be sorted into two equal sizedarrays.

If the algorithm'Partition' produces two regions of size n/2. The recurrencerelation is then $T(n) = T(n/2) + T(n/2)+\Theta(n) = 2T(n/2) + \Theta(n)$

OR

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

VIVA-VOCE QUESTIONS:

1. What is the worst case time complexity of the Quick sort?