TITLE: Quick Sort

Student Name: ARITTRA BAG

Semester: 3rd

Section: B

Class Roll: 103

University Roll: 10900122105

Subject Code: PCC-CS301

Subject Name: Data Structure & Algorithms

QUICK SORT

Explanation

Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes O (n log n) comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as O(n2). Basically, the quick sort algorithm is faster than other O (n log n) algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the ***partition*** operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Like merge sort, the ***base case*** of the recursion occurs when the array has zero or one element

because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

Technique

Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable. That is, loc = 0, left = 0, and right = n–1 (where n in the number of elements in the array)

2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc.

That is, a[loc] should be less than a[right].

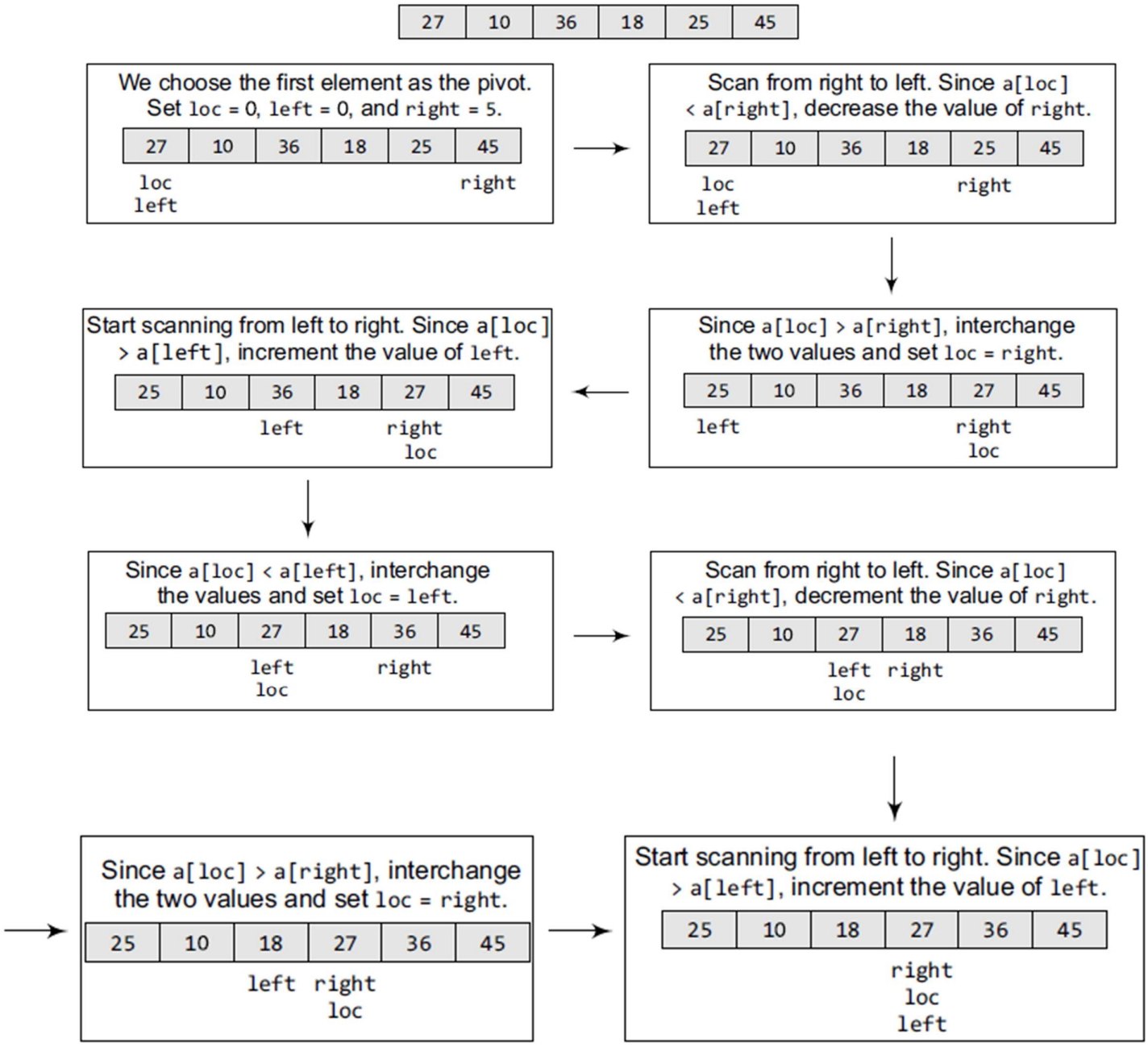
a) If that is the case, then simply continue comparing until right becomes equal to loc. Once right = loc, it means the pivot has been placed in its correct position.
b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.
c) Set loc = right

3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc.

That is, a[loc] should be greater than a[left].

a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position.
b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.
c) Set loc = left.

Example 14.6 Sort the elements given in the following array using quick sort algorithm



Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it. The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

Algorithm

PARTITION (ARR, BEG, END, LOC)

```
Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC =
BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG =
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC !=
RIGHT
     SET RIGHT = RIGHT - 1
     [END OF LOOP]
Step 4: IF LOC = RIGHT
          SET FLAG = 1
     ELSE IF ARR[LOC] > ARR[RIGHT]
          SWAP ARR[LOC] with ARR[RIGHT]
          SET LOC = RIGHT
     [END OF IF]
Step 5: IF FLAG = 0
          Repeat while ARR[LOC] >= ARR[LEFT] AND LOC !=
     LEFT
          SET LEFT = LEFT + 1
          [END OF LOOP]
Step 6:        IF LOC = LEFT
               SET FLAG = 1
          ELSE IF ARR[LOC] < ARR[LEFT]
               SWAP ARR[LOC] with ARR[LEFT]
               SET LOC = LEFT
          [END OF IF]
     [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

QUICK_SORT (ARR, BEG, END)

```
Step 1: IF (BEG < END)
          CALL PARTITION (ARR, BEG, END, LOC)
          CALL QUICKSORT(ARR, BEG, LOC - 1)
          CALL QUICKSORT(ARR, LOC + 1, END)
     [END OF IF]
Step 2: END
```

Code

```
#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
void main(){
      int arr[size], i, n;
      printf("\n Enter the number of elements in the array: ");
      scanf("%d", &n);
      printf("\n Enter the elements of the array: ");
      for(i=0;i<n;i++){
            scanf("%d", &arr[i]);
      }
      quick_sort(arr, 0, n-1);
      printf("\n The sorted array is: \n");
      for(i=0;i<n;i++)
      printf(" %d\t", arr[i]);
      getch();
}
int partition(int a[], int beg, int end){
      int left, right, temp, loc, flag;
      loc = left = beg;
      right = end;
      flag = 0;
      while(flag != 1){
            while((a[loc] <= a[right]) && (loc!=right))
            right--;
            if(loc==right)
            flag =1;
            else if(a[loc]>a[right]){
                  temp = a[loc];
                  a[loc] = a[right];
                  a[right] = temp;
                  loc = right;
            }
            if(flag!=1){
                  while((a[loc] >= a[left]) && (loc!=left))
                  left++;
                  if(loc==left)
                  flag =1;
                  else if(a[loc] <a[left]){
                        temp = a[loc];
                        a[loc] = a[left];
                        a[left] = temp;
                        loc = left;
```

```
                }
            }
        }
        return loc;
}
void quick_sort(int a[], int beg, int end){
        int loc;
        if(beg<end){
            loc = partition(a, beg, end);
            quick_sort(a, beg, loc-1);
            quick_sort(a, loc+1, end);
        }
}
```

Complexity

In the average case, the running time of quick sort can be given as O(n log n). The partitioning of
the array which simply loops over the elements of the array once uses O(n) time. In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only log n nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n). And because at each level, there can only be O(n), the resultant time is given as O(n log n) time. Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as O(n2). The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot. However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of O(n log n).

Pros and Cons of Quick Sort

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.