Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

AY: 2024-25

Class:	**SE**	**Semester:**	**IV**
Course Code:	**CSL401**	**Course Name:**	**Analysis of Algorithm Lab**

Name of Student:	**Shravani Sandeep Raut**
Roll No. :	**48**
Experiment No.:	**3**
Title of the Experiment:	**Quick Sort and Merge sort**
Date of Performance:	**23/01/2025**
Date of Submission:	**30/01/2025**

Evaluation

Performance Indicator	Max. Marks	Marks Obtained
Performance	5	
Understanding	5	
Journal work and timely submission	10	
Total	20	

Performance Indicator	Exceed Expectations (EE)	Meet Expectations (ME)	Below Expectations (BE)
Performance	4-5	2-3	1
Understanding	4-5	2-3	1
Journal work and timely submission	8-10	5-8	1-4

Checked by

Name of Faculty : Mrs. Sneha Yadav

Signature :

Date:

Experiment No. 3

Title: Quick Sort and Merge Sort

Aim: To implement Quick Sort and Merge Sort and Comparative analysis for large values of 'n'.

Objective: To introduce the methods of designing and analysing algorithms.

Theory:

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows:

1. Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.
2. Conquer: Sort the two subsequence recursively using merge sort.
3. Combine: Merge the two sorted subsequence to produce the sorted answer.

Partition-exchange sort or quicksort algorithm was developed in 1960 by Tony Hoare. He developed the algorithm to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.

Quick sort algorithm on average, makes O(n log n) comparisons to sort n items. In the worst case, it makes O(n2) comparisons, though this behavior is rare. Quicksort is often faster in practice than other O(n log n) algorithms. Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space used by the stack during the recursion.

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sublists.

1. Elements less than pivot element.

2. Pivot element.

3. Elements greater than pivot element.

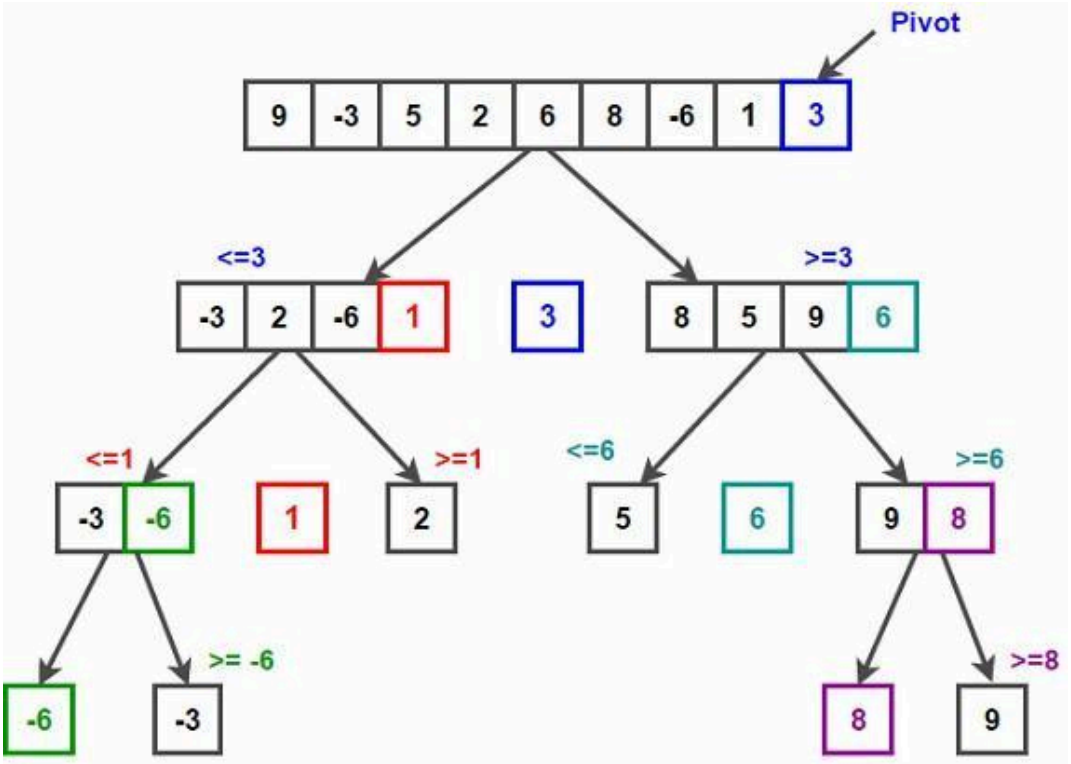
Where pivot as middle element of large list. Let's understand through example:

List : 3 7 8 5 2 1 9 5 4

In above list assume 4 is pivot element so rewrite list as: 3 1 2 4 5 8 9 5 7

Here, I want to say that we set the pivot element (4) which has in left side elements are less than and right hand side elements are greater than. Now you think, how's arrange the less than and greater than elements? Be patient, you get answer soon.

Example:

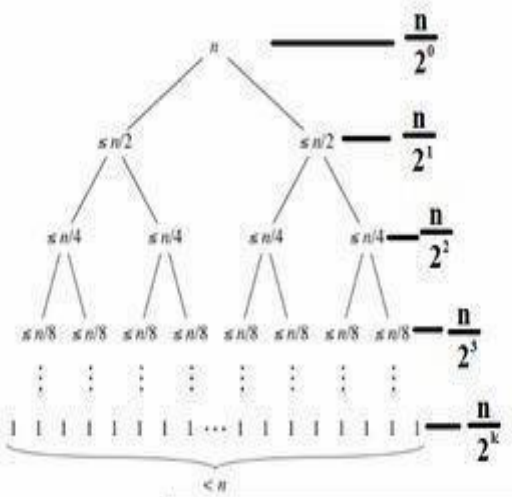



```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now at right place */
```

```
        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)
}
```

Quick Sort: Best case scenario



After k steps, the value becomes 1

$\Rightarrow \frac{n}{2^k} = 1$

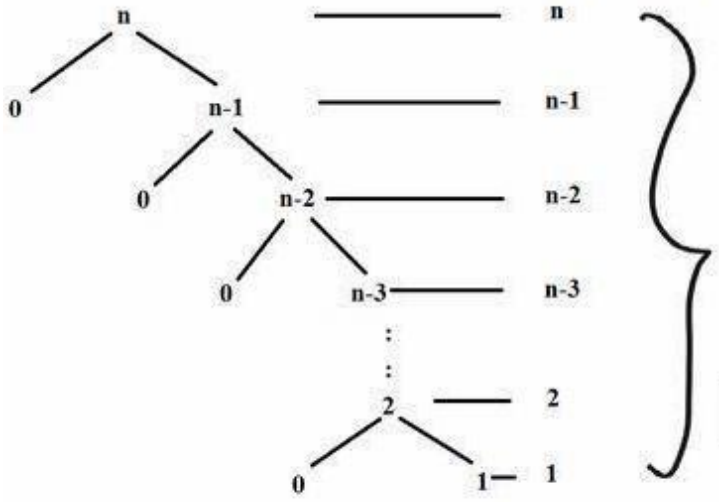
$\Rightarrow n = 2^k$

Applying log on both sides,

$k = \log n$

Time complexity = height of tree = k = O(log n)

Quick Sort- Worst Case Scenario



Time complexity =

$n + (n-1) + (n-2) + (n-3) + ... + 2 + 1$

$= \sum n$

$= \frac{n * (n-1)}{2}$

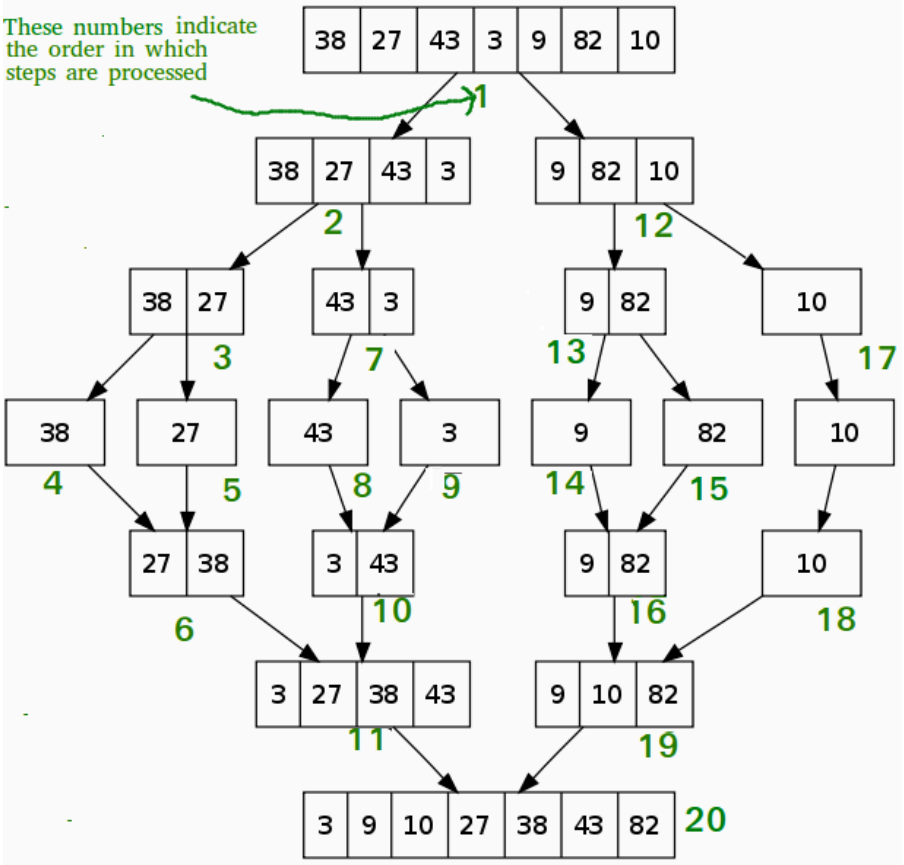
$\Rightarrow O(n^2)$

During the Mergesort process the object in the collection are divided into two collections. To split a collection, Mergesort will take the middle of the collection and split the collection into its left and its right part. The resulting collections are again recursively sorted via the Mergesort algorithm.

Once the sorting process of the two collections is finished, the result of the two collections is combined. To combine both collections Mergesort start at each collection at the beginning. It pick the object which is smaller and inserts this object into the new collection. For this collection it now selects the next elements and selects the smaller element from both collection.

Once all elements from both collections have been inserted in the new collection, Mergesort has successfully sorted the collection. To avoid the creation of too many collections, typically one new collection is created and the left and right side are treated as different collections.

Example:



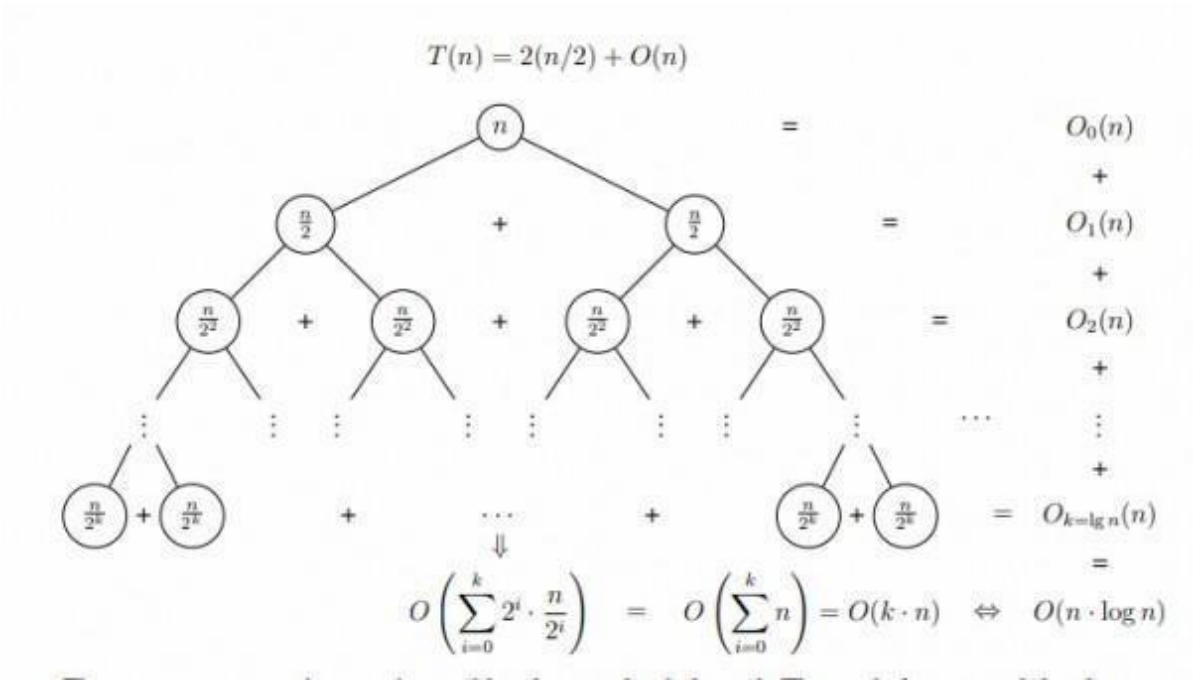
Algorithm and Complexity:

```
1   Algorithm MergeSort1(low, high)
2   // The global array a[low : high] is sorted in nondecreasing order
3   // using the auxiliary array link[low : high]. The values in link
4   // represent a list of the indices low through high giving a[ ] in
5   // sorted order. A pointer to the beginning of the list is returned.
6   {
7       if ((high − low) < 15) then
8           return InsertionSort1(a, link, low, high);
9       else
10      {
11          mid := ⌊(low + high)/2⌋;
12          q := MergeSort1(low, mid);
13          r := MergeSort1(mid + 1, high);
14          return Merge1(q, r);
15      }
16  }
```

Algorithm 3.10 Merge sort using links

```
1   Algorithm Merge1(q, r)
2   // q and r are pointers to lists contained in the global array
3   // link[0 : n]. link[0] is introduced only for convenience and need
4   // not be initialized. The lists pointed at by q and r are merged
5   // and a pointer to the beginning of the merged list is returned.
6   {
7       i := q; j := r; k := 0;
8       // The new list starts at link[0].
9       while ((i ≠ 0) and (j ≠ 0)) do
10      { // While both lists are nonempty do
11          if (a[i] ≤ a[j]) then
12          { // Find the smaller key.
13              link[k] := i; k := i; i := link[i];
14              // Add a new key to the list.
15          }
16          else
17          {
18              link[k] := j; k := j; j := link[j];
19          }
20      }
21      if (i = 0) then link[k] := j;
22      else link[k] := i;
23      return link[0];
24  }
```

Algorithm 3.11 Merging linked lists of sorted elements



Implementation:

MERGE SORT

```
#include <stdio.h>

int n, A[20];

void Merge_Sort(int A[], int low, int high);
void Combine(int A[], int low, int mid, int high);

int main() {
   int i;
   printf("Enter the size of array: ");
   scanf("%d", &n);

   printf("Enter the elements of array:\n");
   for (i = 0; i < n; i++) {
      printf("Enter value: ");
      scanf("%d", &A[i]);
   }

   printf("\nThe unsorted array is: \n");
   for (i = 0; i < n; i++) {
      printf("%d\t", A[i]);
   }

   Merge_Sort(A, 0, n - 1);

   printf("\nAfter sorting, the array is: \n");
   for (i = 0; i < n; i++) {
      printf("%d\t", A[i]);
   }
```

```
    return 0;
}

// Recursive Merge Sort
void Merge_Sort(int A[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        Merge_Sort(A, low, mid);
        Merge_Sort(A, mid + 1, high);
        Combine(A, low, mid, high);
    }
}

void Combine(int A[], int low, int mid, int high) {
    int n1 = mid - low + 1, n2 = high - mid;
    int Left[n1], Right[n2];


    for (int i = 0; i < n1; i++)
        Left[i] = A[low + i];
    for (int j = 0; j < n2; j++)
        Right[j] = A[mid + 1 + j];


    int i = 0, j = 0, k = low;
    while (i < n1 && j < n2) {
        if (Left[i] <= Right[j]) {
            A[k++] = Left[i++];
        } else {
            A[k++] = Right[j++];
        }
    }

    while (i < n1) {
        A[k++] = Left[i++];
    }

    while (j < n2) {
        A[k++] = Right[j++];
    }
}
```

QUICK SORT

```
#include <stdio.h>

int n, A[20];

void Quick_Sort(int A[], int low, int high);
int Partition(int A[], int low, int high);

int main() {
    int i;
    printf("Enter the size of array: ");
    scanf("%d", &n);

    printf("Enter the elements of array:\n");
    for (i = 0; i < n; i++) {
        printf("Enter value: ");
```

```
        scanf("%d", &A[i]);
    }

    printf("\nThe unsorted array is: \n");
    for (i = 0; i < n; i++) {
        printf("%d\t", A[i]);
    }

    Quick_Sort(A, 0, n - 1);

    printf("\nAfter sorting, the array is: \n");
    for (i = 0; i < n; i++) {
        printf("%d\t", A[i]);
    }

    return 0;
}

// Quick Sort function (Recursive)
void Quick_Sort(int A[], int low, int high) {
    if (low < high) {
        int pivotIndex = Partition(A, low, high);
        Quick_Sort(A, low, pivotIndex - 1);
        Quick_Sort(A, pivotIndex + 1, high);
    }
}

int Partition(int A[], int low, int high) {
    int pivot = A[high]; // Choose last element as pivot
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (A[j] < pivot) {
            i++;
            // Swap A[i] and A[j]
            int temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    int temp = A[i + 1];
    A[i + 1] = A[high];
    A[high] = temp;

    return i + 1;
}
```

Output -

```
Enter the size of array: 5
Enter the elements of array:
Enter value: 90
Enter value: 34
Enter value: 12
Enter value: 52
Enter value: 3

The unsorted array is:
90      34      12      52      3
After sorting, the array is:
3       12      34      52      90
```

```
Enter the size of array: 5
Enter the elements of array:
Enter value: 30
Enter value: 12
Enter value: 78
Enter value: 34
Enter value: 5

The unsorted array is:
30      12      78      34      5
After sorting, the array is:
5       12      30      34      78
```

Conclusion:

Quick Sort and Merge Sort are efficient, divide-and-conquer sorting algorithms. Quick Sort is faster in practice due to in-place sorting and better cache performance, though its worst-case time complexity is O(n^2). Merge Sort guarantees O(n log n) performance even in the worst case, making it more predictable and stable. However, it requires additional space, unlike Quick Sort. Quick Sort is preferred for average cases and smaller datasets, while Merge Sort is ideal for linked lists and large datasets requiring stable sorting. Both algorithms are foundational in computer science and widely implemented in various applications and standard libraries.