Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	02

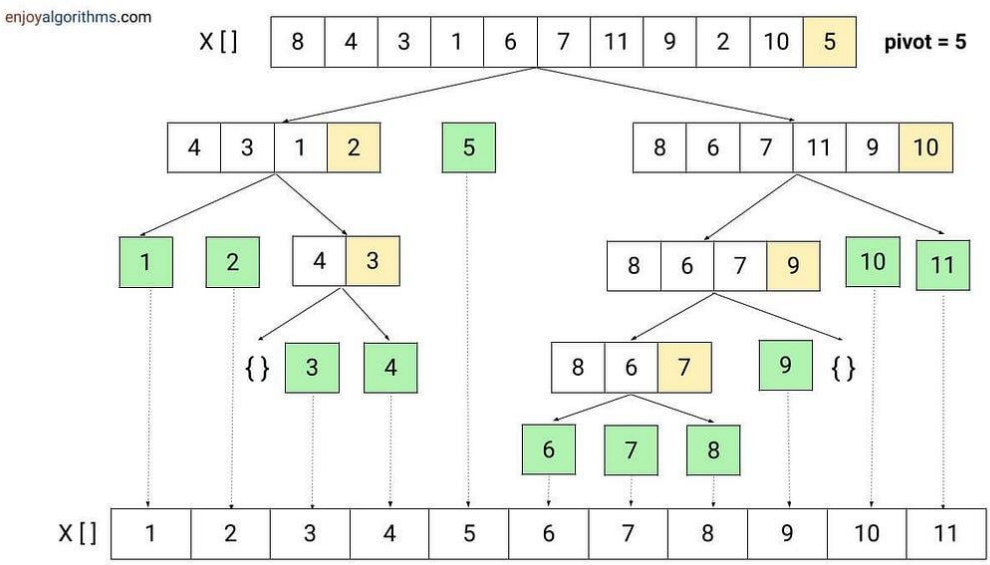
AIM: **IMPLEMENT AND ANALYSE MERGE AND QUICK SORT**

THEORY:

Quick sort

Quicksort is a sorting algorithm based on the divide and conquer approach where

- An array is divided into subarrays by selecting a pivot element (element selected from the array).
- While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
- The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.



- At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

- 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 Algorithm:
  The partition algorithm rearranges the sub-arrays in a place.

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

- Time Complexities
  - **Worst Case Complexity [Big-O]**: $O(n^2)$
    - ✓ It occurs when the pivot element picked is either the greatest or the smallest element.
    - ✓ This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty, and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.
    - ✓ However, the quicksort algorithm has better performance for scattered pivots.
  - **Best Case Complexity [Big-omega]**: O(n*log n)
    - ✓ It occurs when the pivot element is always the middle element or near to the middle element.
  - **Average Case Complexity [Big-theta]**: O(n*log n)
    - ✓ It occurs when the above conditions do not occur.
- **Space Complexity**
  - The space complexity for quicksort is O(log n).

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Function to swap two elements
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Partition the array using the last element as the pivot
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program
```

```
int main() {
   clock_t start, stop;
   clock_t start_w, stop_w;

    int n = 5000;
    int arr[n];
    for (int i = 0; i<n; i++){
        arr[i] = rand();
    }

   start = clock();

    quickSort(arr, 0, n - 1);
   stop = clock();
   float res = stop - start;
   printf("\nAverage/best case CPU time =%f units", res);

   start_w = clock();
    //worst case
    quickSort(arr, 0, n - 1);
   stop_w = clock();
   float x = stop_w - start_w;
   printf("\nworst case CPU time =%f units\n", x);

    return 0;
}
```

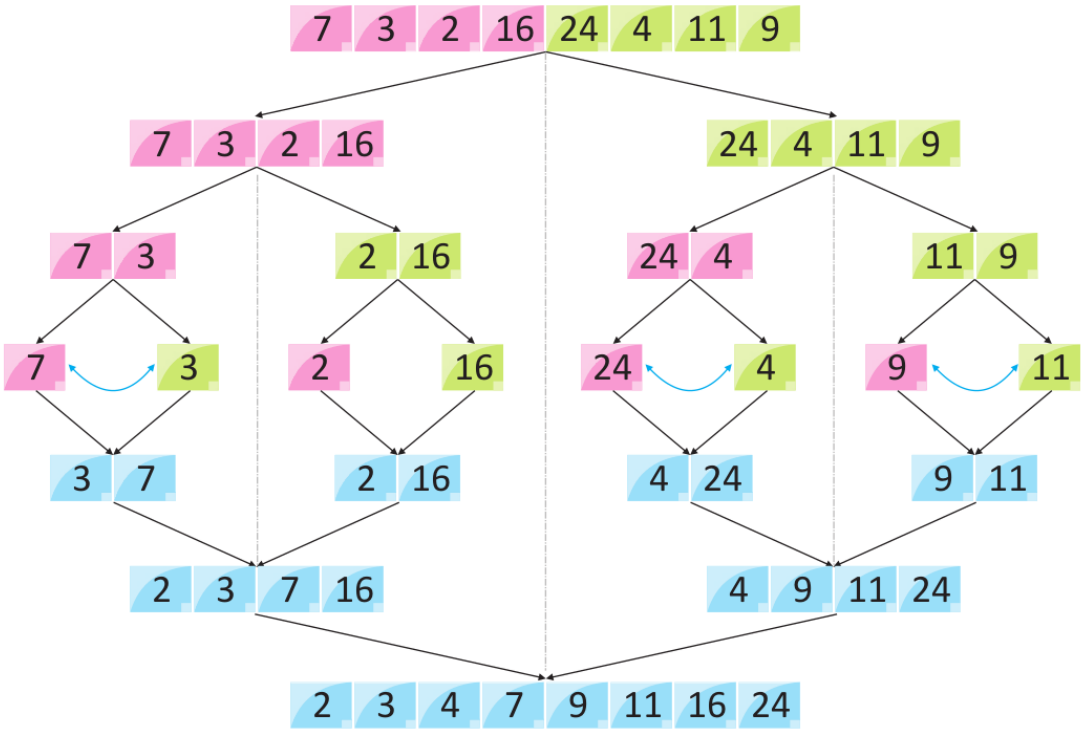
- Output:

```
jh' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'

Average/best case CPU time =2.000000 units
worst case CPU time =70.000000 units
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

Merge Sort

- Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
- Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided.



- This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion.
- If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves.
- Finally, when both halves are sorted, the merge operation is applied.
- Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.
- Algorithm:

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

```
//for merge function
Have we reached the end of any of the arrays?
    No:
        Compare current elements of both arrays
        Copy smaller element into sorted array
        Move pointer of element containing smaller element
    Yes:
        Copy all remaining elements of non-empty array
```

- Time Complexity:
  - The time complexity of Merge Sort is **θ(n log(n))** in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
- Space Complexity:
  - Space complexity is O(1) because an extra variable is used.

CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void mergeDesc(int arr[], int p, int q, int r)
{

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2)
```

```
  {
    if (L[i] >= M[j])
    {
      arr[k] = L[i];
      i++;
    }
    else
    {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1)
  {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2)
  {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Divide the array into two subarrays, sort them and merge them
void mergeSortDesc(int arr[], int l, int r)
{
  if (l < r)
  {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSortDesc(arr, l, m);
    mergeSortDesc(arr, m + 1, r);

    // Merge the sorted subarrays
```

```
    mergeDesc(arr, l, m, r);
  }
}
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r)
{

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2)
  {
    if (L[i] <= M[j])
    {
      arr[k] = L[i];
      i++;
    }
    else
    {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1)
```

```
  {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2)
  {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r)
{
  if (l < r)
  {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}

// Driver program
int main()
{
  clock_t start, stop;
  clock_t start_b, stop_b;
  clock_t start_w, stop_w;

  int size = 10000;

  int arr[size];

  for (int i = 0; i < size; i++)
  {
```

```
    arr[i] = rand();
  }

  start = clock();
  // sorting jumbled array
  mergeSort(arr, 0, size - 1);
  stop = clock();
  float res = stop - start;
  printf("\nAvg case CPU time =%f units", res);

  // sorting sorted array
  start_b = clock();
  mergeSort(arr, 0, size - 1);
  stop_b = clock();
  float x = stop_b - start_b;
  printf("\nBest case CPU time =%f units", x);

  // sorting sorted array in descending order
  start_w = clock();
  mergeSortDesc(arr, 0, size - 1);
  stop_w = clock();
  x = stop_w - start_w;
  printf("\nworst case CPU time =%f units", x);
}
```

OUTPUT:

```
ui1ar.1gp' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'

Avg case CPU time =0.000000 units
Best case CPU time =0.000000 units
worst case CPU time =1.000000 units
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

Basis for comparison	**Quick Sort**	**Merge Sort**
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	In the merge sort, the array is parted into just 2 halves (i.e. n/2).

Basis for comparison	Quick Sort	Merge Sort
Worst case complexity	O(n^2)	O(n log n)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Method	Quick sort is in- place sorting method.	Merge sort is not in – place sorting method.
Space	Quicksort does not require additional array space.	For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements.