AOA Lab 2

Name: Prakash P. Biswas
Roll No: 9947
Branch: SE Comps B
Batch: A

Merge Sort(Stable**):**

CODE:

```
#include <stdio.h>

// Merge the two lists
void mergelist(int arr[], int l, int m, int r) {
  int i, j, k;
  int n1 = m - l + 1; // Size of left subarray
  int n2 = r - m;     // Size of right subarray

  // Create temporary arrays
  int L[n1], R[n2];

  // Copy data to temporary arrays L[] and R[]
  for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
  for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];

  // Merge the temporary arrays back into arr[l..r]
  i = 0;
  j = 0;
  k = l;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = R[j];
      j++;
    }
    k++;
  }
  // Copy the remaining elements of L[], if any
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }
  // Copy the remaining elements of R[], if any
```

```c
  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
  }
}

// l is for left index and r is right index of the sub-array of arr to be
sorted
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    // Same as (l+r)/2, but avoids overflow for large l and r
    int m = l + (r - l) / 2;
    // Sort first and second halves
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    // Merge the sorted halves
    mergelist(arr, l, m, r);
  }
}

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

int main() {
  // Predefined array
  int arr[] = {12, 11, 13, 5, 6, 6, 7};
  int n = 7;
  // Print the given array
  printf("Original array: \n");
  printArray(arr, n);
  // Apply the merge sort on the array
  mergeSort(arr, 0, n - 1);
  // Print the sorted array
  printf("Sorted array: \n");
  printArray(arr, n);
  return 0;
}
```

<u>OUTPUT:</u>

```
Original array:
12 11 13 5 6 6 7
Sorted array:
5 6 6 7 11 12 13
```

COMPLEXITY:

In merge sort array is recursively sorted ~~into~~ by dividing ~~a~~subarray in two subarrays.

$\therefore T(n) = 2T(n/2) + O(n)$ — time taken to merge two subarrays

$\therefore$ By master method,

$f(n) = n, \; a = 2, \; b = 2$

$n^{\log_b a} = n^{\log_2 2} = n$

$\therefore f(n) = n^{\log_b a}$.

$\therefore T(n) = \cancel{f(n) \times \log n}\ \Theta(f(n) \times \log n)$

$\cancel{\Theta(n) = n\log n}$ $\boxed{T(n) = \Theta(n\log n)}$

Quick Sort(Fast)**:**

CODE:

```
#include <stdio.h>

void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high) {
    // Making the last element as the pivot element
    int pivot = arr[high];
    // Get the index where the pivot needs to be placed
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            // If the current element is greater than the pivot then increment
the
            // counter and swap it with the previous element
            i++;
            swap(&arr[i - 1], &arr[j]);
        }
    }
    // Once the position to swap is found, then swap the pivot to the position
we get
    swap(&arr[i], &arr[high]);
    return i;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Getting the location of the pivot element
        int pi = partition(arr, low, high);
        // Dividing the unsorted subarrays as before and after the pivot
element
        // and again applying the quicksort on them
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    // Predefined array
    int arr[] = {12, 11, 13, 5, 6, 6, 7};
    int arr_size = 7;
    // Print the given array
    printf("Given array is \n");
```

```
    for (int i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");
    // Apply quicksort on the given array
    quickSort(arr, 0, arr_size - 1);
    // Print the sorted array
    printf("Sorted array is \n");
    for (int i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT:

```
ksort.cpp
Given array is
12 11 13 5 6 6
Sorted array is
5 6 6 11 12 13
```

COMPLEXITY:

1. Worst case:

Array is already sorted or nearly sorted.

$\therefore T(n) = T(n-1) + O(n)$ $\boxed{T(n) = 1 \text{ for } n=1}$

$= T(n-2) + (n-1) + n$

$= T(n-3) + (n-2) + (n-1) + n$

$\vdots$

$= T(n-k) + (n-(k-1)) + (n-(k-2)) \cdots n$

$\therefore n-k = 1$

$\therefore k = n-1$

$\therefore T(n) = T(1) + T(n-(n-1-1)) + T(n-(n-1-2)) \cdots T(n)$

$= T(1) + T(2) \cdots T(n)$

$= 1 + 2 + 3 \cdots n$

$= \frac{n(n+1)}{2}$

$\boxed{\therefore T(n) = O(n^2)}$

2. Best case:

When pivot keeps on dividing the subarray into 2 equal halves.

$\therefore T(n) = 2T(n/2) + n$

$\therefore$ By master method

$f(n) = n, \quad a=2, \quad b=2$

$n^{\log_b a} = n^{\log_2 2} = \boxed{n}$

$\therefore f(n) = n^{\log_b a}$

$\therefore T(n) = \Omega(n \times \log n)$

$\boxed{T(n) = \Omega(n\log n)}$

The time complexity for average case is $\Theta(n\log n)$ because array is divided into two balanced partitions using up and down counter

What changes will you do to make quick sort a randomized quick sort?

- The main difference between a randomised quicksort and a general quicksort lies in how they choose the pivot element during partitioning.
- In general quicksort, either the first or last element is taken as pivot element.
- In randomised quicksort a random element is taken as the pivot element from the subarray.
- This helps lower the possibility of encountering the worst case scenario.
- To switch from general quicksort to a randomised quicksort we can take a random element from the subarray as pivot element instead of hardcoding it as the first or last element.