Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058, India
(Autonomous College Affiliated to University of Mumbai)

Experiment No.	**2**
Name	**Varada Khadake**
UID No.	**2021300059**
Class & Division	**COMPS A BATCH D**
Date of performance	**11/02/2023**
Date of submission	**14/02/2023**

Aim: To analyze the run times of merge sort and quick sort algorithms.

Algorithm:

1.Start
2.initialize array a with size=100000
3for i=0 to i<1000
4.call function getData(i+1,a)
5.initialize start and end values of clock() function
6.call function insertionSort(a,(i+1)*100)
7.print array
8. call function getData(i+1,a)
9.initialize start and end values of clock() function
10.call function selectionSort(a,(i+1)*100)
11.print array
12.end for

mergeSort():

```
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)
```

The merge function works as follows:

1. Create copies of the subarrays L <- A[p..q] and M <- A[q+1..r].
2. Create three pointers i, j and k
a. i maintains current index of L, starting at 1
b. j maintains current index of M, starting at 1
c. k maintains the current index of A[p..q], starting at p.
3. Until we reach the end of either L or M, pick the larger among the elements from L and M and place them in the correct position at A[p..q]
4. When we run out of elements in either L or M, pick up the remaining elements and put in A[p..q]

quickSort():

```
QUICKSORT (array A, start, end)
{
  if (start < end)
  {
 p = partition(A, start, end)
 QUICKSORT (A, start, p - 1)
 QUICKSORT (A, p + 1, end)
 }
}

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

Observation/Theory:

Merge Sort:

Using the **Divide and Conquer** technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.
Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

Quick sort:

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **O(n^2)**.

As observed from the graphs, the run time of merge sort is seen to be almost constant. While that of quick sort increases as the size of array increase. Also, the run time of merge sort is seen to be always lower than that of quick sort. The maximum of merge sort is 0.04 while that of merge sort is 0.20. As the time complexity of merge sort is same in all cases O(n*log n) so it remains almost constant.

Code:

Merge sort:

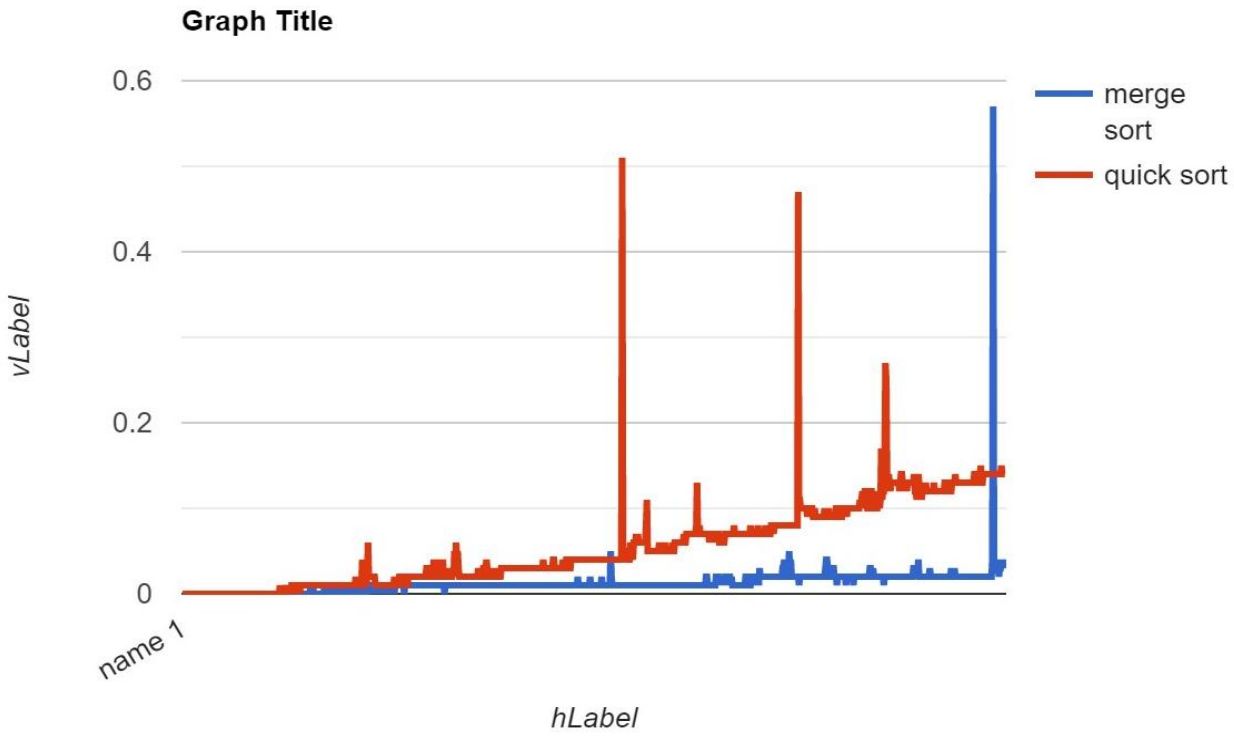
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    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++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

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

void mergeSort(int arr[],
               int l, int r)
{
    if (l < r)
    {

        int m = l + (r - l) / 2;

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

        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

void getData(int x, int arr[])
{
    for (int i = 0; i < x * 100; i++)
    {
        arr[i] = rand() % 100 + 1;
    }
}

int main()
{

    clock_t start, end;
    double time_req;
    int a[100000];
    for (int i = 0; i < 1000; i++)
    {

        getData(i + 1, a);
        start = clock();

        mergeSort(a, 0, ((i + 1) * 100) - 1);
        end = clock();

        time_req = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%.21f ", time_req);
    }

    return 0;
}
```

Quick sort:

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

int partition(int a[], int start, int end)
{
    int pivot = a[end];
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {

        if (a[j] < pivot)
        {
            i++;
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i + 1];
    a[i + 1] = a[end];
    a[end] = t;
    return (i + 1);
}

void quick(int a[], int start, int end)
{
    if (start < end)
    {
        int p = partition(a, start, end);
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}

void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}

void getData(int x, int arr[])
{
    for (int i = 0; i < x * 100; i++)
    {
        arr[i] = rand() % 100 + 1;
    }
}

int main()
{

    clock_t start, end;
    double time_req;
    int a[100000];
    for (int i = 0; i < 1000; i++)
    {

        getData(i + 1, a);
        start = clock();

        quick(a, 0, ((i + 1) * 100) - 1);
        end = clock();

        time_req = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%.21f ", time_req);
    }

    return 0;
}
```

Output:



Conclusion: I observed the graphs of run time of both the algorithms and inferred that merge sort is better than quick sort for large size of array. This experiment helped me to understand the working and implementation of both the algorithms.