Sardar Patel Institute of Technology

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

Name	Pratiksha Pravin Patil
UID No.	2022301016
Class & Division	Comp(A)
Subject.	Design Analysis of Algorithm
Experiment No.	2
Aim	Experiment based on divide and conquers approach
Objective.	To understand the running time of algorithms by implementing two sortingalgorithms based on divide and conquers approach namely Merge and Quick sort.

Theory: Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into sub problems, then solving the sub problems, andcombining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typicalto 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.

Algorithm:

```
1.  QUICKSORT (array A, start, end)
2.  {
3.   1 if (start < end)
4.   2 {
5.  3 p = partition(A, start, end)
6.  4 QUICKSORT (A, start, p - 1)
7.  5 QUICKSORT (A, p + 1, end)
8.  6 }
9.  }
```

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 $\mathbf{O(n^2)}$

Merge Sort:

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Now, let's see the algorithm of merge sort.

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
1.    MERGE_SORT(arr, beg, end)
3.    if beg < end
4.    set mid = (beg + end)/2
5.    MERGE_SORT(arr, beg, mid)
6.    MERGE_SORT(arr, mid + 1, end)
7.    MERGE (arr, beg, mid, end)
8.    end of if
9.    END MERGE_SORT
```

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

Program:

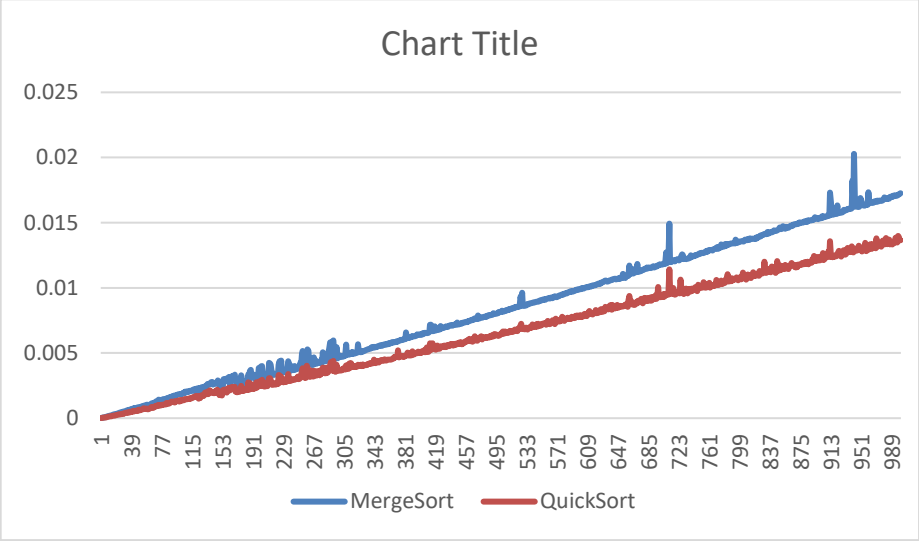
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
long SWAP = 0;
void merge(int arr[], int p, int q, int r)
{
int i, j, k;
int n1 = q - p + 1;
int n2 = r - q;
int L[n1], R[n2];
for (i = 0; i < n1; i++)
L[i] = arr[p + i];
for (j = 0; j < n2; j++)
R[j] = arr[q + 1 + j];
i = 0;
j = 0;
k = p;
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];
```

```
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);
}
}
int quicksort(int a[], int start, int end)
{
int pivot = a[end];
//int pivot = a[start];
//int random = start + rand() % (end - start);
//int pivot = a[random];
//int mid = start + (end - start)/2;
//int pivot = a[mid];
int i = (start - 1);
for (int j = start; j <= end - 1; j++)
{
if (a[j] < pivot)
{
```

```
if (a[j] < pivot)
{
i++;
int t = a[i];
a[i] = a[j];
a[j] = t;
 SWAP++;
}
}
int t = a[i + 1];
a[i + 1] = a[end];
a[end] = t;
 SWAP++;
return (i + 1);
}
double quick(int a[], int start, int end)
{
if (start < end)
{
int p = quicksort(a, start, end);
quick(a, start, p - 1);
quick(a, p + 1, end);
}
}
int main()
{
double qust,mest;
srand(time(0));
FILE *fp,*file;
fp = fopen("random.txt", "w");
for (int i = 0; i < 100000; i++)
{
fprintf(fp, "%d\n", rand() % 900001 + 100000);
}
int upper_limit = 100;
```

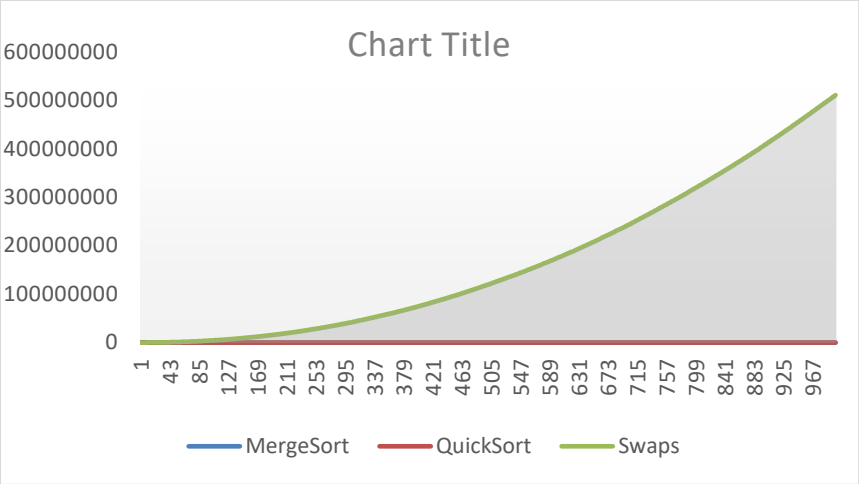
```
104  for (int i = 0; i < 100000; i++)
105  {
106  fprintf(fp, "%d\n", rand() % 900001 + 100000);
107  }
108  int upper_limit = 100;
109  fclose(fp);
110  file = fopen("output.txt","w");
111  fprintf(file,"Block\tMerSort\tQuickSort\tSwaps\n");
112  for (int i = 0; i < 1000; i++)
113  {
114  fp = fopen("random.txt", "r");
115  int arr1[upper_limit], arr2[upper_limit], temp_num;
116  for (int j = 0; j < upper_limit; j++)
117  {
118  fscanf(fp, "%d", &temp_num);
119  arr1[j] = temp_num;
120  arr2[j] = temp_num;
121  }
122  fclose(fp);
123  clock_t t;
124  t = clock();
125  mergeSort(arr2,0,upper_limit-1);
126  t = clock() - t;
127  mest = ((double)t) / CLOCKS_PER_SEC;
128  clock_t t1;
129  t1 = clock();
130  qust=quick(arr1,0,upper_limit-1);
131  t1 = clock() - t1;
132  qust = ((double)t1) / CLOCKS_PER_SEC;
133  fprintf(file,"%d\t%lf\t%lf\t%ld\n",i+1, mest, qust,SWAP);
134  fflush(stdout);
135  upper_limit += 100;
136  }
137  return 0;
138  }
```

RESULT:



Merge sort: As we can see from the graph, merge sort takes more time at eachiteration than quick sort as its always about quick sort in the graph. There are some spikes in at certain movements but overall we can observe a linear increase in the graph.

Quick sort: On the other hand, time taken to perform quick sort is just slightly less than merge sort but on the other hand there are a lot of spikes that can be observed where one rises to as high as 0.05 seconds. It is expected that with some proper worksome of these spikes can be overcome. Also, compare with merge sort, which achieves similar results but has much lower round-trip-time.



As we observe as we add more numbers at each iteration to the block the number of swapsrequired increase significantly reaching hundreds of millions. Linear increase is observed

Conclusion: In this experiment we have studied that quick sort performs better than merge sort but it also unstable whereas merge sort takes a little more time but it has a linear increase and doesn't have a lotof ups and downs.

Moreover, when performing quick sort on large data the position of the pivot matter a lot as justchanging the pivot yields drastic results. Also, no. of swaps required to sort data is directly proportional to the size of the data.