Name: Rishi Tiku

Class: SE DS

UID: 202170067

Subject: Design & Analysis of Algorithms

Experiment: 2

AIM: To find the running time of Merge & Quick Sort Algorithms.

Introduction

Merge Sort

Merge sort is 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.

In simple terms, the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

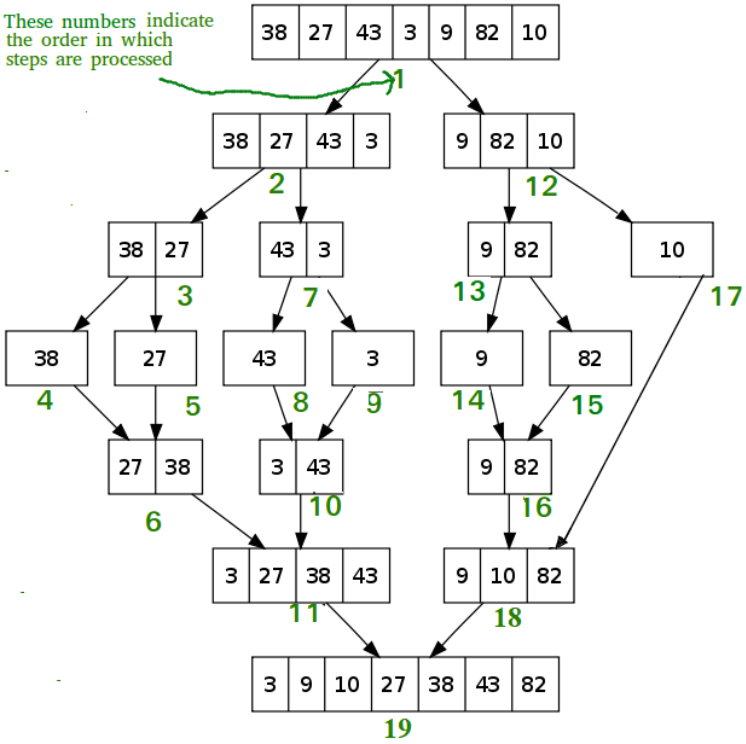
One of the main advantages of merge sort is that it has a time complexity of

$$O(n.\log n)$$

which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

$$\text{Recurrence Relation: } T(n) = 2T(n/2) + cn$$



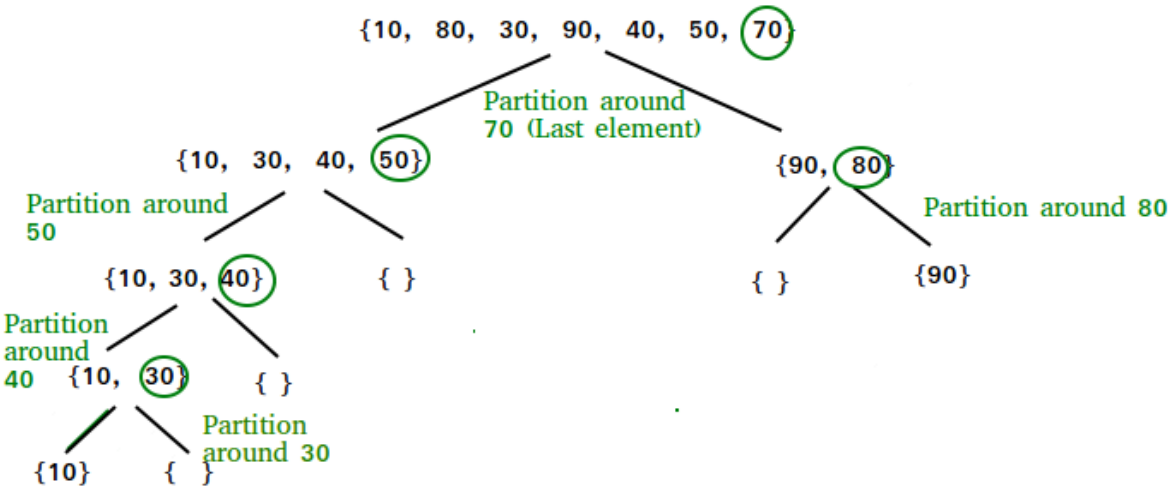
Quick Sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Recurrence Relation: $T(n) = T(k) + T(n-k-1) + \theta(n)$


{10, 80, 30, 90, 40, 50, 70}
Partition around
70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around
50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition
around
40
{10, 30}
{ }
Partition
around 30
{10}
{ }

Algorithm

1) Merge Sort

Step 1: START

Step 2: declare array and left, right, mid variable

Step 3: perform merge function.

```
if left > right
        return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

Step 4: STOP

2) Quick Sort Pivot Algorithm

Step 1 – START

Step 2 − Choose the highest index value has pivot

Step 3 − Take two variables to point left and right of the list excluding pivot

Step 4 − left points to the low index

Step 5 − right points to the high

Step 6 − while value at left is less than pivot move right

Step 7 − while value at right is greater than pivot move left

Step 8 − if both step 5 and step 6 does not match swap left and right

Step 9 − if left ≥ right, the point where they met is new pivot

Step 10 – STOP

3) Quick sort Algorithm

Step 1 – START

Step 2 − Make the right-most index value pivot

Step 3 − partition the array using pivot value

Step 4 − quicksort left partition recursively

Step 5 − quicksort right partition recursively

Step 6 – STOP

4) Program

Step – 1 START

Step – 2 Create a file containing 100000 random numbers (unsorted).

Step – 3 Load the first 100, first 200, first 300 (… and so on) elements of the file into an array A of the same size as the number of elements.

Step – 4 Duplicate A to get B.

Step – 5 Calculate the time taken to run Merge Sort on A and Quick Sort on B for x number of elements in array.

Step – 6 Output time taken for both algorithms along with number of elements present to a file, preferably .csv for easy import into Excel.

Step – 7 Run Step 5 till x = 100000, incrementing x in steps of 100

Step – 8 STOP

CODE

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

void printCur()
{
    time_t s, val = 1;
    struct tm* current_time;

    // time in seconds
    s = time(NULL);

    // to get current time
    current_time = localtime(&s);

    // print time in minutes,
    // hours and seconds
    printf("%02d:%02d:%02d\n",
           current_time->tm_hour,
           current_time->tm_min,
           current_time->tm_sec);

}

void quickSort(int number[],int first,int last){
    int i, j, pivot, temp;
    if(first<last){
        pivot=first;
        i=first;
        j=last;
        while(i<j){
            while(number[i]<=number[pivot]&&i<last)
            i++;
            while(number[j]>number[pivot])
            j--;
            if(i<j){
                temp=number[i];
                number[i]=number[j];
                number[j]=temp;
             }
```

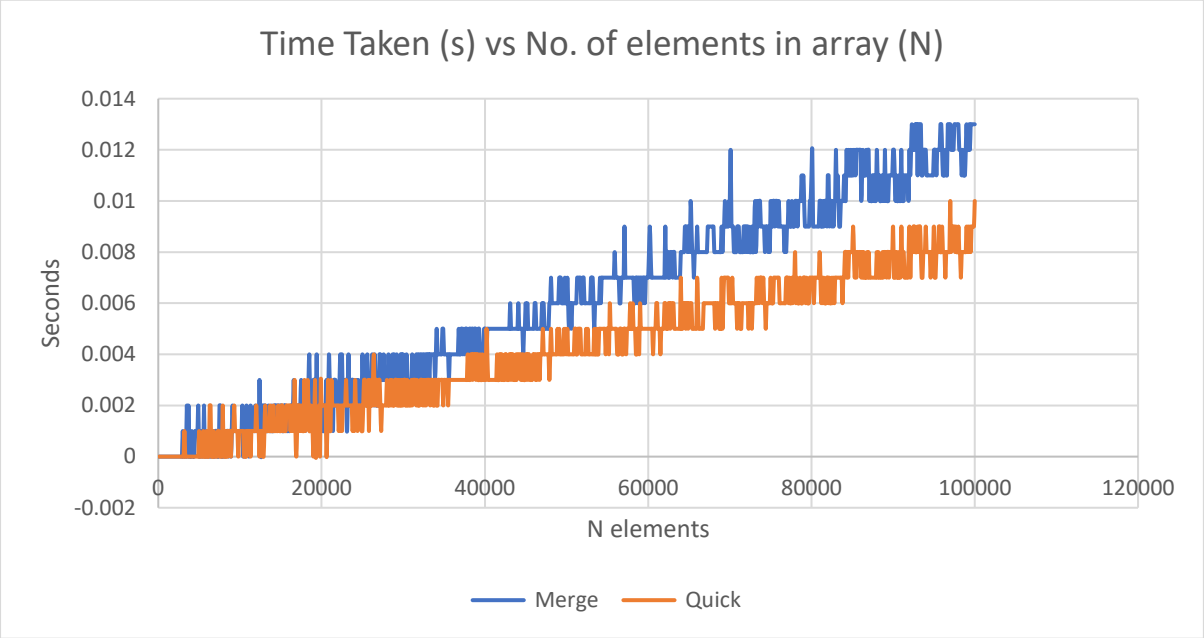
```
        }
        temp=number[pivot];
        number[pivot]=number[j];
        number[j]=temp;
        quickSort(number,first,j-1);
        quickSort(number,j+1,last);
    }
}

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

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp 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 temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    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 there
    are any */
```

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

    /* Copy the remaining elements of R[], if there
    are any */
    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 h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

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


int genFile()
{
    FILE * fptr;
    if(!(fptr = fopen("RandomNumbers.txt", "w")))
    {
        return 1;
    }

    for(int i = 0; i<100000; i++)
```

```
    {
        fprintf(fptr, "%d\n", rand());
    }

    fclose(fptr);
    return 0;
}


int main()
{
    genFile();
    FILE * fptrs, * fptrd;
    fptrd = fopen("timesdemo.csv", "w");
    if(!fptrd)
        return 0;
    fprintf(fptrd, "Cases, Merge, Quick\n");

    for(long int x = 100; x<=100000; x+=100)
    {
        fptrs = fopen("RandomNumbers.txt", "r");
        int A[x], B[x];
        clock_t start1, end1, start2, end2;
        if(x % 1000 == 0)
        {
                printf("x = %ld. Time = ", x);
        printCur();
        }
        for(long int i = 0; i<x; i++)
        {
            fscanf(fptrs, "%d\n", &A[i]);
            B[i] = A[i];
        }

        start1 = clock();
        mergeSort(A, 0, x-1);
        end1 = clock();

        start2 = clock();
        quickSort(B, 0, x-1);
        end2 = clock();
```

```
        double t1 = (double) (end1 - start1) / CLOCKS_PER_SEC;
        double t2 = (double) (end2 - start2) / CLOCKS_PER_SEC;

        fprintf(fptrd, "%ld, %f, %f\n", x, t1, t2);

        fclose(fptrs);
    }
    fclose(fptrd);
    return 0;
}
```

Graph


Time Taken (s) vs No. of elements in array (N)
Seconds
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
-0.002
0
20000
40000
60000
80000
100000
120000
N elements
Merge
Quick

Observation

Complexity	Merge	Quick
Time	Worst: O(N.log N) Best: O(N.log N)	O(N.log N) $O(N^2)$
Space	O(n)	O(log N)

1) Quick Sort is observed to be faster than Merge Sort in this example.
2) In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.
3) In quick sort, each recursive call uses O(1) words in local variables, hence the total space complexity is proportional to the height of the recursion tree.
4) 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.
5) Though the worst-case complexity of quicksort is more than other sorting algorithms such as Merge sort and Heap sort, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.
6) In merge sort, worst case 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.

Conclusion

Merge and Quick Sort Algorithms' Time & Space Complexity is investigated and Quick Sort is found to be a faster sorting algorithm for the data sample.