Danyl Fernandes
2020012004 (72)
11-04-2021

AOA Experiment 2

Aim:

To implement & analyze Merge Sort Algorithm and to compare its complexity with Quick Sort:

Implementation:

```
#include <bits/stdc++.h>
using namespace std;

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

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0;
    int j = 0;
    int 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++;
```

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

void mergeSort(int arr[],int l,int r) {
    if (l >= r) {
        return;
    }

    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) {
    for (int i = 0; i < size; i++)
        cout << A[i] << " ";
}

int main() {
    int arr[] = {4, 2, 1, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}

```

Output:

```
Given array is
4 2 1 5 6 7
Sorted array is
1 2 4 5 6 7
```

Danyl Fernandes
2020012004 (72)

Exp 02

Theory

- Merge sort is one of those classic sortin algorithms that follows divide & conquer stragegy in which the original problem is divided into smaller sub problems.

- The nature of the sub problems remains the same as the original problem, except the number of instances.

- These problems are solved iteratively and results of all the sub problems are combined & we then get the final solution

- The performance of Merge sort does not depend on any specification of input other than the size of the input, thus it has the best, average & worst case efficiency of $O(\log_2 n)$

- The disadvantage of the merge sort is the requirement of an auxillary array of size n during merging, thus 2n memory locations are used by merge sort to merge the solutions

- It uses stack space due to recursive calls proportional to $\log_2 n$

Daryl Fernandes
2020012004 (72)

- The need for an auxillary array can be eleminated by applying in-place merging of results or sub-arrays.

<u>Algorithm</u>

```
void merge (int a[] , int lb , int m , int ub){
        int i, j , temp [100], k = 0;
        i = lb ;
        j = m+1 ;
        while (i<=m && j<= ub) {
            if (a[i] < a[j]) {
                temp [k] = a [i] ;
                i++;
                k++;
            }
            else {
                temp [k] = a[j];
                j++;
                k++;
            }
            while (i<= m) {
                temp [k++] = a[j++] ;
            }
            while (j <= m) {
                temp [k++] = a [j++];
            }
            k = 0.
        for (i = lb ; i <= ub ; i++) {
            a[j] = temp [k++] ;
        }
void mergesort (in a[] , int lb, int ub){
```

Daryl Fernandes
2020012004 (72)

```
    if (lb < ub) {
        m = (lb + ub) / 2 ;
        mergesort (a, lb, m);
        mergesort (a, m+1, ub);
        merge (a, lb, m, ub);
    }
}
```

Complexity Analysis:

The recurrence relation can be written a

$$T(n) = \begin{cases} 1 \\ T(n/2) + T(n/2) + C_n & \text{if } n > 1 \end{cases}$$

In general, for sufficiently large n

$$T(n) = \begin{cases} 2T(n/2) + C_n & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

$$T(n) = 2T(n/2) + C_n$$
$$= 2\left[2T\left(\frac{n}{2^2}\right) + \frac{C_n}{n}\right] C_n$$
$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{C_n}{4}\right] + 2C_n$$
$$= 2^3 T\left(\frac{n}{2^3}\right) + 3C_n$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kC_n \quad \cdots \circledast$$

Let $\frac{n}{2^k} = 1$

$\therefore n = 2^k$

$k = \log_2 n$

Daryl Fernandes
2020D|2004 (72)

Substituting this value in *

$T(n) = n(T(1)) + \log_2 n \cdot C \cdot n$

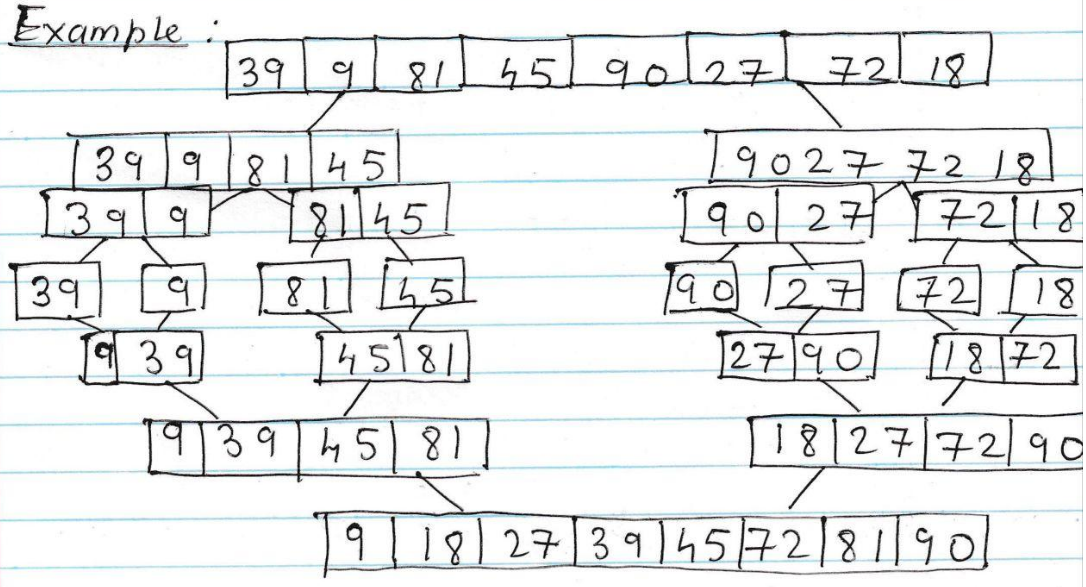
$T(n) = n + Cn \cdot \log_2 n$

$T(n) = O(n \log_2 n)$

Comparison with Quick sort:

- The best time complexity for both merge sort & quick sort is $O(n \cdot \log n)$.
- The worst case time complexity for quick sort is $O(n^2)$ where merge sort is $O(\log_2 n n)$, hence quick sort takes a lot more comparisons in this case
- Quick sort is faster in some cases such as for small data set but merge sort can operate well on any type of data sets whether large or small quick sort cannot
- Merge Sort is more efficient than quick sort.

Example:



Darryl Fernandes
2020012004 (72)

Conclusion:
Implementation of merge sort with analysis and complexity comparison with quick sort was successful.