LEENA K CH.SC.U4CSE24122

Design Analysis and Algorithm – Lab Work

Week 4

Question 1: Write a program to Illustrate Merge Sort to sort the array

{157,110,147,122,111,149,151,141,123,112,117,133}.

Code:

```
//CH.SC.U4CSE24122
#include <stdio.h>
#define MAX 50
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[MAX], R[MAX];
    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++];
        else
            arr[k++] = R[j++];
    }
    while(i < n1)
        arr[k++] = L[i++];
    while(j < n2)
        arr[k++] = R[j++];
}
```

```c
void mergeSort(int arr[], int l, int r){
    int m;
    if(l < r){
        m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
int main(){
    printf("CH.SC.U4CSE24122\n");
    int i;
    int arr[] = {157,110,147,122,111,149,151,141,123,112,117,133};
    int n = sizeof(arr) / sizeof(arr[0]);
    mergeSort(arr, 0, n - 1);
    printf("Sorted array: ");
    for(i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Output:

```
C:\Users\kpriy\Downloads\uh

CH.SC.U4CSE24122
Sorted array: 110 111 112 117 122 123 133 141 147 149 151 157
--------------------------------
Process exited after 0.07928 seconds with return value 0
Press any key to continue . . .
```

Time Complexity:
The program recursively divides the array into two halves until each subarray has one element. At each level of recursion, the merge step requires *n* comparisons and assignments. Since the array is divided log n times, the total number of operations is proportional to nlog n. Hence, the **Time Complexity of Merge Sort in the best case, average case, and worst case is O(n log n)**.

Space Complexity:
The space occupied will be proportional to the size of the input array. For an array of size *n*, additional temporary arrays L[] and R[] are created during each merge step, together holding *n* elements. Apart from these, a few integer variables (i, j, k, l, m, r) are used for indexing and recursion. The recursion stack depth is log n. Hence, the total extra memory used grows linearly with the input size.
Therefore, the **Space Complexity is O(n)**.

Question 2: Write a program to Illustrate Quick Sort the array {157,110,147,122,111,149,151,141,123,112,117,133}.

Code:

```
//CH.SC.U4CSE24122
#include<stdio.h>
int partition(int arr[],int l,int h){
    int pivot=arr[h];
    int i=l-1,j;
    for(j=l;j<h;j++){
        if(arr[j]<pivot){
            i++;
            int temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
        }
    }
    int temp=arr[i+1];
    arr[i+1]=arr[h];
    arr[h]=temp;
    return (i+1);
}
void quicksort(int arr[],int l,int h){
    if(l<h){
        int pi=partition(arr,l,h);
        quicksort(arr,l,pi-1);
        quicksort(arr,pi+1,h);
    }
}
int main(){
```

```c
    printf("CH.SC.U4CSE24122\n");
    int arr[]={157,110,147,122,111,149,151,141,123,112,117,133};
    int h=sizeof(arr)/sizeof(arr[0]);
    quicksort(arr,0,h-1);
    printf("Sorted Array: ");
    int i;
    for(i=0;i<h;i++){
        printf("%d ",arr[i]);
    }
    return 0;
}
```

Output:

```
CH.SC.U4CSE24122
Sorted Array: 110 111 112 117 122 123 133 141 147 149 151 157
--------------------------------
Process exited after 0.06539 seconds with return value 0
Press any key to continue . . .
```

Space Complexity:
The space occupied will be proportional to the recursion depth. For an array of size *n*, the program uses a few integer variables (pivot, i, j, l, h, pi, temp) for indexing and swapping. The recursion stack depth depends on how the pivot divides the array: in the best and average case, the depth is log n, while in the worst case it can go as deep as *n*. Hence, the total extra memory used depends on recursion depth.
Therefore, the **Space Complexity is O(log n) on average and O(n) in the worst case**.

Time Complexity:
The program partitions the array around a pivot and recursively sorts the two halves. At each level of recursion, the partition step requires *n* comparisons and swaps. If the pivot divides the array evenly, the array is split log n times, giving a total of nlog n operations. If the pivot divides poorly (always smallest or largest element), one side has size n-1 and the other has size 0, leading to n^2 operations.
Hence, the **Time Complexity of Quick Sort is O(n log n) in the best case and average case, and O(n^2) in the worst case**.