Design and Analysis of Algorithms Sorting

Name : Halitha Begam S

Roll-No: CH.SC.U4CSE24157

1. Write a program to implement merge sort

CODE:

```c
#include <stdio.h>

void merge(int arr[],int left,int mid,int right){
    int i,j,k;
    int n1=mid-left+1;
    int n2=right-mid;

    int L[n1],R[n2];

    for (i=0;i<n1;i++)
        L[i]=arr[left+i];
    for (j=0;j<n2;j++)
        R[j]=arr[mid+1+j];

    i=0;j=0;k=left;

    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++];
}

void mergeSort(int arr[],int left,int right) {
    if(left < right){
        int mid=(left+right)/2;

        mergeSort(arr,left,mid);
        mergeSort(arr,mid+1,right);
        merge(arr,left,mid,right);
    }
}
```

```c
int main() {
    int n,i;
    int arr[50];

    printf("Enter number of elements:");
    scanf("%d",&n);

    printf("Enter elements:\n");
    for (i=0;i<n;i++)
        scanf("%d", &arr[i]);

    mergeSort(arr,0, n-1);

    printf("Sorted array:\n");
    for (i=0;i<n;i++)
        printf("%d ", arr[i]);

    return 0;
}
```

OUPTUT:

```
C:\Documents\sem4\daa\Week-4>gcc merge_sort.c

C:\Documents\sem4\daa\Week-4>a
Enter number of elements:10
Enter elements:
6 87 12 64 97 50 132 4 30 64
Sorted array:
4 6 12 30 50 64 64 87 97 132
C:\Documents\sem4\daa\Week-4>gcc quick_sort.c
```

Time complexity: O(nlogn)

Always divides the array into halves and merges them in linear time at each level.

Space complexity: O(n)

Merge sort's space increases because it uses recursion and extra temporary arrays during merging.

2. Write a program to implement quick sort

CODE:

```c
#include <stdio.h>

void swap(int *a,int *b){
    int t=*a;
    *a=*b;
    *b=t;
}

int partition(int arr[],int left,int right){
    int pivot=arr[right];
    int i=left-1;
    int j;

    for(j=left;j<right;j++){
        if(arr[j]<=pivot){
            i++;
            swap(&arr[i],&arr[j]);
        }
    }

    swap(&arr[i+1],&arr[right]);
    return i+1;
}

void quickSort(int arr[],int left,int right){
    if(left<right){
        int pi=partition(arr,left,right);
        quickSort(arr,left,pi-1);
        quickSort(arr,pi+1,right);
    }
}

int main(){
    int n,i;
    int arr[50];
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Enter elements:\n");
```

```
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);

    quickSort(arr,0,n-1);
    printf("Sorted array:\n");

    for(i=0;i<n;i++)
        printf("%d ",arr[i]);
    return 0;
}
```

OUPTUT:

```
C:\Documents\sem4\daa\Week-4>gcc quick_sort.c

C:\Documents\sem4\daa\Week-4>a
Enter number of elements:6
Enter elements:
10 942 343 1 7 30
Sorted array:
1 7 10 30 343 942
```

Time complexity: $O(n^2)$

Quick sort becomes $O(n^2)$ when the pivot always produces one empty partition and one partition of size $n - 1$.

Space complexity: O(logn) - Average, **O(n)** - Worst

Quick sort uses **O(log n)** space for balanced partitions and **O(n)** space when partitions are highly unbalanced.