PRACTICAL 3

Divide and Conquer Strategy.

PRACTICAL 3.1

AIM:

Implement and perform analysis of worst case of Merge Sort and Quick sort. Compare both algorithms.

SOFTWARE REQUIREMENT:

C++ Compiler, MS Word.

MERGE SORT

SOFTWARE REQUIREMENT:

C++ Compiler, MS Word.

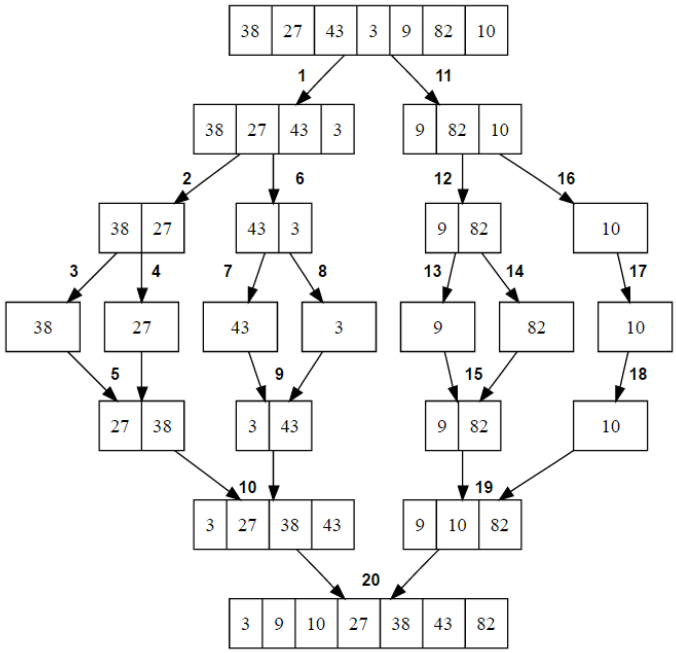
THEORY (MERGE SORT):

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list. The idea:

- Divide the unsorted list into N sublists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

Example is as shown:



ALGORITHM (MERGE SORT):

- **Function MergeSort()**
  Function Merge() takes 3 arguments that are: int arr[], int const begin, int const end. Consider int arr[] as A, int const begin as p and int const end as r.
  - **Step 1:** If p<r.
  - **Step 2:** Then q → (p+ r)/2.
  - **Step 3:** Call MERGE-SORT (A, p, q)
  - **Step 4:** Call MERGE-SORT (A, q+1, r)
  - **Step 5:** Call MERGE (A, p, q, r)

- **Function Merge()**
  Function Merge() takes 4 arguments that is: int arr[], int const left, int const mid, int const right. Consider int arr[] as A, int const left as p, int const mid as q and int const right as r.
  - **Step 1:** subarrayone = q-p+1.
  - **Step 2:** subarraytwo = r-q.
  - **Step 3:** create arrays [1..... subarrayone + 1] and R [1..... subarraytwo +1].
  - **Step 4:** for i ← 1 to subarrayone
  - **Step 5:** do [i] ← A [p+ i-1]
  - **Step 6:** for j ← 1 to subarraytwo
  - **Step 7:** do R[j] ← A[q + j]
  - **Step 8:** L [subarrayone + 1] ← ∞
  - **Step 9:** R[subarraytwo + 1] ← ∞

- **Step 10:** I ← 1
- **Step 11:** J ← 1
- **Step 12:** For k ← p to r
- **Step 13:** Do if L [i] ≤ R[j]
- **Step 14:** then A[k] ← L[i]
- **Step 15:** i ← i +1
- **Step 16:** else A[k] ← R[j]
- **Step 17:** j ← j+1

CODE (MERGE SORT):

```
#include<bits/stdc++.h>
using namespace std;
int c1=0;

//to Print the array
void printArray(int arr[],int n){
    for(int i=0;i<n;i++){
        cout<<arr[i]<<" ";
    }
}

// Merge two subarrays leftArray and rightArray into arr
void merge(int arr[],int const left,int const mid,int const right){
    c1++;
    int const subarrayone=mid-left+1;
    int const subarraytwo=right-mid;

    auto *leftArray=new int[subarrayone];
    auto *rightArray=new int[subarraytwo];

    for(int i=0;i<subarrayone;i++){
        c1++;
        leftArray[i]=arr[left+i];
    }
    for(int j=0;j<subarraytwo;j++){
        c1++;
        rightArray[j]=arr[mid+1+j];
    }

    // Maintain current index of sub-arrays and main array
    int indexofsubarrayone=0,indexofsubarraytwo=0,indexofmergedarray=left;
```

```
    // Until we reach either end of either leftArray or rightArray, pick larger a
mong
    // elements leftArray and rightArray and place them in the correct position a
t arr[left..right]
   while(indexofsubarrayone<subarrayone && indexofsubarraytwo<subarraytwo) {

       c1++;

       if(leftArray[indexofsubarrayone]<=rightArray[indexofsubarraytwo]){
           c1++;
           arr[indexofmergedarray]=leftArray[indexofsubarrayone];c1++;
           indexofsubarrayone++;c1++;
       }
       else{
           c1++;
           arr[indexofmergedarray]=rightArray[indexofsubarraytwo];c1++;
           indexofsubarraytwo++;c1++;
       }
       c1++;
       indexofmergedarray++;
   }

    // When we run out of elements in either leftArray or rightArray,
    // pick up the remaining elements and put in arr[left..right]
   while(indexofsubarrayone<subarrayone){

       c1++;

       arr[indexofmergedarray]=leftArray[indexofsubarrayone];c1++;
       indexofsubarrayone++;c1++;
       indexofmergedarray++;c1++;
   }

    while(indexofsubarraytwo<subarraytwo){
        c1++;
        arr[indexofmergedarray]=rightArray[indexofsubarraytwo];c1++;
        indexofsubarraytwo++;c1++;
        indexofmergedarray++;c1++;
            }

}

// Divide the array into two subarrays, sort them and merge them
void mergesort(int arr[],int const begin,int const end){
    c1++;
```

```
    if(begin>=end)
    return;

    // mid is the point where the array is divided into two subarrays
    auto mid=begin+(end-begin)/2;

    mergesort(arr,begin,mid);
    mergesort(arr,mid+1,end);
    // Merge the sorted subarrays
    merge(arr,begin,mid,end);
}

int main(){
    int n;
    cout<<"Enter no. of elements:";cin>>n;
    // int *arr=new int(n);
    int arr[n];
    cout<<"Enter the elements:";
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }

    // int arr[] = { 12, 11, 13, 5, 6, 7 };
    // auto n = sizeof(arr) / sizeof(arr[0]);

    cout<<"Entered Elements are: ";
    printArray(arr,n);
    mergesort(arr,0,n-1);

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

    cout<<"\nCount:"<<c1;
    return 0;
}
```

OUTPUT (MERGE SORT):

1. **Best Case**

```
Enter no. of elements:5
Enter the elements:1 2 3 4 5
Entered Elements are: 1 2 3 4 5
Sorted array is
1 2 3 4 5
Count:80
```

2. **Average Case**

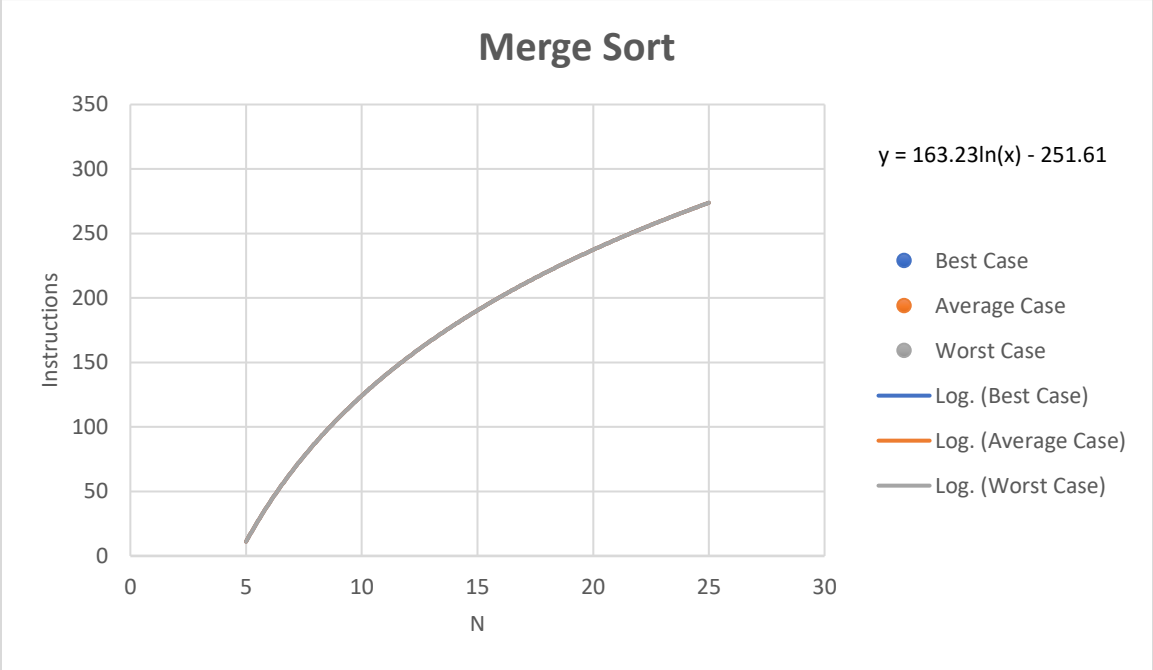
```
Enter no. of elements:5
Enter the elements:2 5 1 6 3
Entered Elements are: 2 5 1 6 3
Sorted array is
1 2 3 5 6
Count:80
```

3. **Worst Case**

```
Enter no. of elements:5
Enter the elements:5 4 3 2 1
Entered Elements are: 5 4 3 2 1
Sorted array is
1 2 3 4 5
Count:78
```

ANALYSIS OF ALGORITHM (MERGE SORT):

N	Best Case	Average Case	Worst Case
5	37	37	37
10	96	96	96
15	161	161	161
20	234	234	234
25	309	309	309



CONCLUSION (MERGE SORT):

Performed bubble sort operation for different cases and found out the time complexities for each case. On performing this algorithm, we found out that the time complexity to implement merge sort in all the three cases is same.

i. Best Case: O(n*log(n))
ii. Average Case: O(n*log(n))
iii. Worst Case: O(n*log(n))

QUICK SORT

SOFTWARE REQUIREMENT:

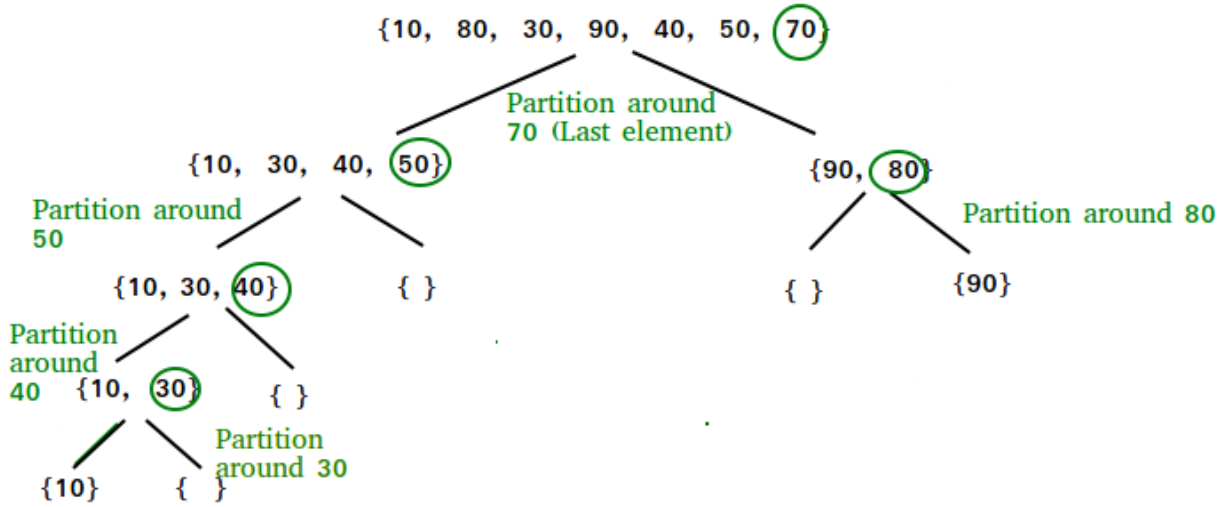
C++ Compiler, MS Word.

THEORY (QUICK SORT):

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conquer strategy. In quick sort, the partition of the list is performed based on the element called pivot. Here pivot element is one of the elements in the list.

The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot". Example:



ALGORITHM (QUICK SORT):

- **Funtion quickSort():**
  Function quickSort () takes 3 arguments that is: int arr[], int low, int high.
  - **Step 1:** IF low < high
  - **Step 2:** THEN pivot <- partition(arr, low, high)
  - **Step 3:** THEN Call quickSort(array, low, pivot - 1)
  - **Step 4:** THEN Call quickSort(array, pivot, high)
- **Function Partition():**
  Function Partition() takes 3 arguments that is: int arr[], int low, int high.
  - **Step 1:** Set high as pivot.
  - **Step 2:** storeIndex <- high - 1
  - **Step 3:** for i <- low + 1 to high
  - **Step 4:** IF element[i] < pivot
  - **Step 5:** THEN swap element[i] and element[storeIndex]
  - **Step 6:** THEN storeIndex++
  - **Step 7:** swap pivotElement and element[storeIndex+1]
  - **Step 8:** return storeIndex + 1

CODE (QUICK SORT):

```
#include<bits/stdc++.h>

int c=0;
//Display the Array
void display(int a[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        printf("%d ",a[i]);
    }
}

//Main Quicksort Working
void QuickSort(int a[],int left,int right)
{
    c++;
    int pi,temp,i,j,k,temp2;

        if(left<right)
        {
            c++;
            k=(left+right)/2;   c++;
```

```
            temp=a[left];   c++;
            a[left]=a[k];   c++;
            a[k]=temp;  c++;

            temp2=a[left];  c++;
            i=left+1;   c++;
            j=right;    c++;
            while(i<=j)
            {
                c++;
                while((i<=right) && (a[i] <= temp2))
                {
                    c++;
                    i++;
                }
                while((j>=left) && (a[j] >temp2))
                {
                    c++;
                    j--;
                }
                if(i<j)
                {
                    c++;
                    temp=a[i];
                    a[i]=a[j];
                    a[j]=temp;
                }
            }
                temp=a[left];   c++;
                a[left]=a[j];   c++;
                a[j]=temp;  c++;
                QuickSort(a,left,j-1);  c++;
                QuickSort(a,j+1,right); c++;
        }
}
int main()
{
    int *array,choice,n,i,cont;
    do{

    printf("Enter the number of elements:");
    scanf("%d",&n);
    array=(int*)malloc(n*sizeof(int));
    printf("Enter the elements:");
    for(i=0;i<n;i++)
```

```
    {
    scanf("%d,",array+i);
    }
    printf("\nEntered elements:\n");
    display(array,n);
    printf("\n---->Options<----\n");
    printf("\n1)Quick Sort\n0)Exit");

    printf("\nEnter your choice:");
    scanf("%d",&choice);
    switch(choice)
    {

        case 1:
            QuickSort(array,0,n-1);
            printf("\nSorted array:\n");
            display(array,n);
            printf("\nCount:");
            printf("\n%d",c);
            printf("\n");
            printf("\n");
             break;
        case 0:
             exit(1);
        default:
                printf("Wrong choice");
                break;
    }
    printf("\n Want to continue?( 1(yes) || 0(no) ):");
    scanf("%d",&cont);
 }while(cont!=0);
 printf("\n\n\t Jaydipsinh Padhiyar(19CE081)");
}
```

OUTPUT (QUICK SORT):

1. **Best Case**

```
Enter the number of elements:5
Enter the elements:1 2 3 4 5

Entered elements:
1 2 3 4 5
---->Options<----

1)Quick Sort
0)Exit
Enter your choice:1

Sorted array:
1 2 3 4 5
Count:
55
```

2. **Average Case**

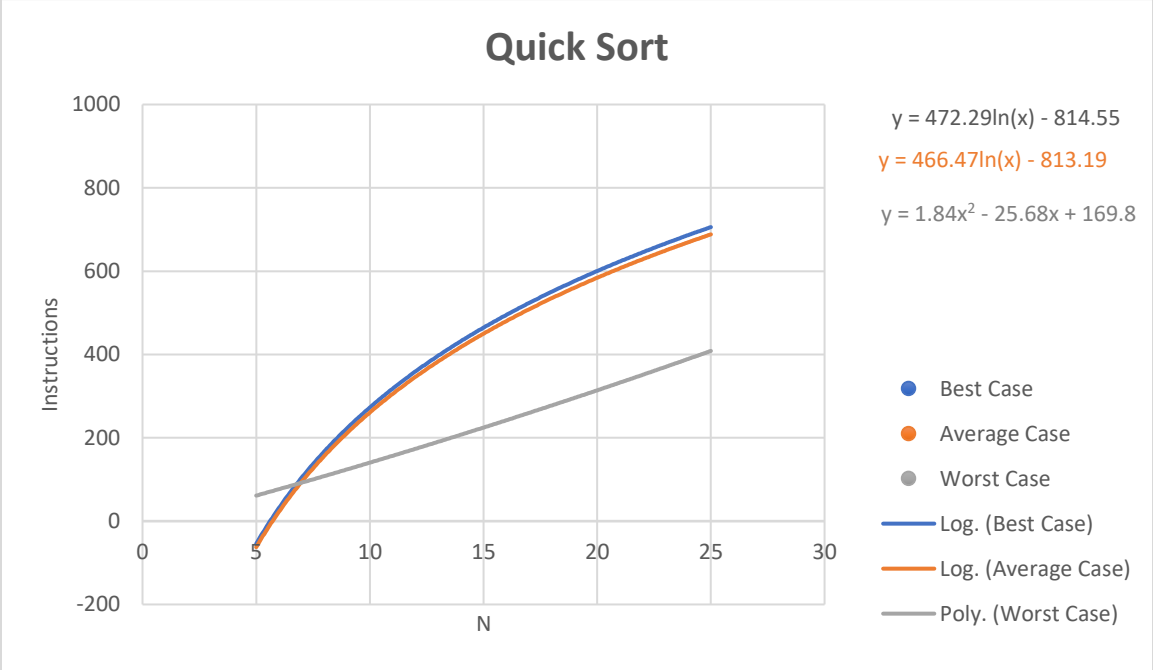
```
Enter the number of elements:5
Enter the elements:5 2 1 3 4

Entered elements:
5 2 1 3 4
---->Options<----

1)Quick Sort
0)Exit
Enter your choice:1

Sorted array:
1 2 3 4 5
Count:
75
```

3. **Worst Case**

```
Enter the number of elements:5
Enter the elements:5 4 3 2 1

Entered elements:
5 4 3 2 1
---->Options<----

1)Quick Sort
0)Exit
Enter your choice:1

Sorted array:
1 2 3 4 5
Count:
59
```

ANALYSIS OF ALGORITHM (QUICK SORT):

N	Best Case	Average Case	Worst Case
5	55	58	59
10	171	189	143
15	318	228	231
20	565	519	303
25	880	927	413



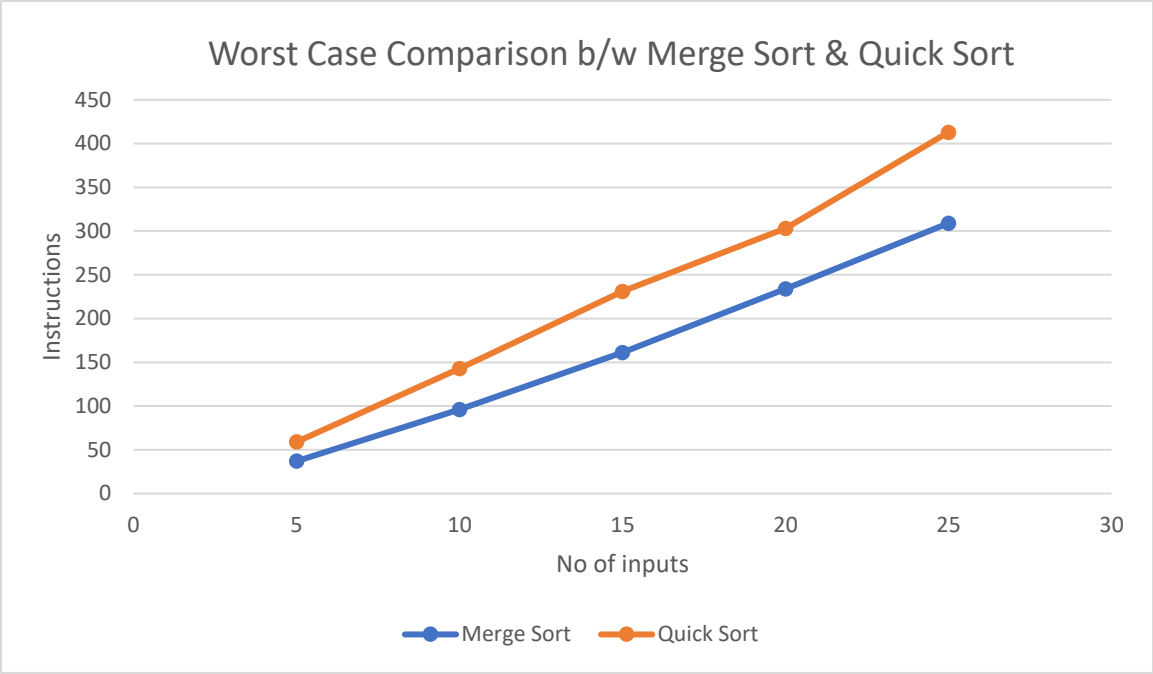
CONCLUSION (QUICK SORT)

Performed Quick sort operation for different cases and found out the time complexities for each case Here for Quick sort in two cases, time complexity remains the same. But in the worst cases to changes to $n^{2.}$

i. Best Case: O(n*log(n))
ii. Average Case: O(n*log(n))
iii. Worst Case: $O(n^2)$

COMPARISION OF ALGORITHMS (WORST CASE (MERGE SORT) v WORST CASE (QUICK SORT))

N	Merge Sort	Quick Sort
5	37	59
10	96	143
15	161	231
20	234	303
25	309	413



CONCLUSION:

By performing this practical we understood two different sorting algorithms (namely merge sort and quick sort) and how they are different to each other. On comparing the time complexities of both the algorithms we found out that merge sort takes less time to execute compared to quick sort in the Worst-case scenario as $O(nlog(n)) < O(n^2)$.

PRACTICAL 3.2

AIM:

Implement the program to find X^Y using divide and conquer strategy and print the total number of multiplications required to find X^Y. Test the program for following test cases:

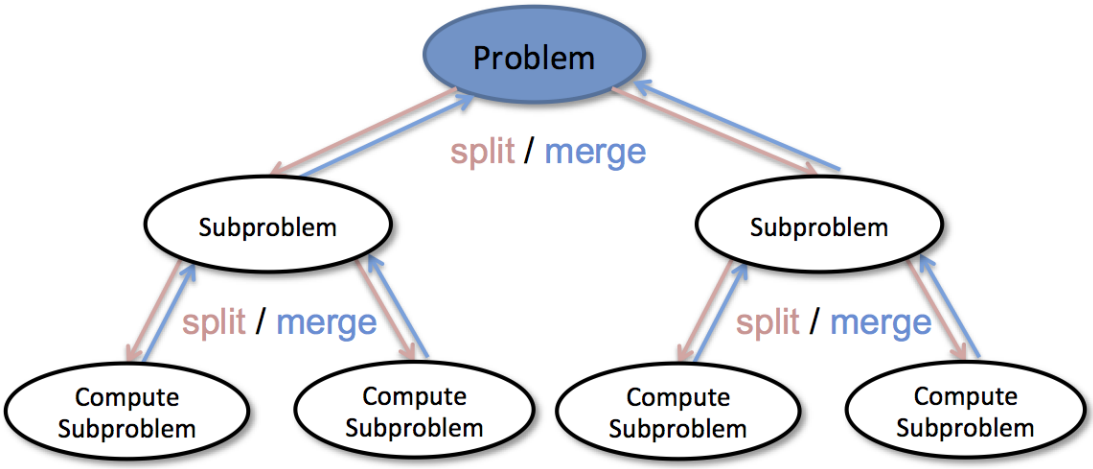
Test Case	X	Y
1	2	6
2	7	25
3	5	34

SOFTWARE REQUIREMENT:

C++ Compiler, MS Word.

THEORY:

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) is solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem. In this problem we divided the task according to exponent, that is exponent%2==1 and if exponent%2!=1.



ALGORITHM:

- **Step 1:** Entered value of X and Y.
- **Step 2:** Call function **Power()** which takes two parameters X and Y.
- **Step 3:** Initialize xy = 1.
- **Step 4:** IF Y%2==1 THEN xy=xy*X and Y=Y-1.
- **Step 5:** ELSE X=X*X, Y=Y/2

CODE:

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

int count1 = 1;
//Divide And Conquer
double power(int x, unsigned int y) {

    count1++;
    if (y == 0) {
        return 1;
    }
    double temp = power(x, y / 2);//Store Power Value Temporarily
    if (y % 2 == 0) {
        return temp * temp;//For Even Y
    }
    else {
        return x * temp * temp; //For Odd Y
    }
}


int main() {

   const char separator = ' ';
    const int nameWidth = 15;
    const int numWidth = 15;
    int x[3] = {2,7,5};
   unsigned int y[3]={6,25,34};
   int n=sizeof(x)/sizeof(x[0]);

   cout<<"Answer is:\n";

    cout << left << setw(nameWidth) << setfill(separator) << "I";
     cout << left << setw(nameWidth) << setfill(separator) << "X";
```

```
     cout << left << setw(nameWidth) << setfill(separator) << "Y";
     cout << left << setw(nameWidth) << setfill(separator) << "X^Y";
     cout << left << setw(nameWidth) << setfill(separator) <<"Count"<<"\n";

    for(int i=0;i<n;i++){
        cout << left << setw(nameWidth) << setfill(separator) << i;
     cout << left << setw(nameWidth) << setfill(separator) << x[i];
     cout << left << setw(nameWidth) << setfill(separator) << y[i];
     cout << left << setw(nameWidth) << setfill(separator) << power(x[i],y[i]);
     cout << left << setw(nameWidth) << setfill(separator) <<count1<<"\n";
        count1=0;
    }

    return 0;
}
```

OUTPUT:

```
Answer is:
I               X               Y               X^Y             Count
0               2               6               64              5
1               7               25              1.34107e+021    6
2               5               34              5.82077e+023    7
```

CONCLUSION

By performing this practical we learnt about the divide and conquer strategy and to how we can divide the tasks and get the optimal result by conquering/combining them. Here we also found out how many times number of multiplications takes place. Time complexity of the above program turned out to be O(log(n)).