Lab Exercises:

1). Write a program to find total number of nodes in a binary tree and analyse its efficiency. Obtain the experimental result of order of growth and plot the result.

Program:

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

int opcount = 0;
int max(int a, int b) { return (a > b) ? a : b; }

typedef struct node* nodeptr;
typedef struct node {
	int data;
	nodeptr left, right;
} node;

nodeptr newNode(int data) {
	nodeptr temp = (nodeptr) malloc(sizeof(node));
	temp -> data = data;
	temp -> left = NULL;
	temp -> right = NULL;
	return temp;
}

void insertTree(nodeptr *root, int data)
{
	if (*root == NULL) {
		*root = newNode(data);
		return;
	}

	printf("\nEnter 1 to insert left of %d or 2 to insert right: ", (*root)->data);
	int n;
	scanf("%d", &n);

	if (n == 1)
		insertTree(&(*root)->left, data);
	else
		insertTree(&(*root)->right, data);
}

int countNodes(nodeptr root)
{
	opcount++;
```

```
        if (root == NULL)
                return 0;

        return 1 + countNodes(root->left) + countNodes(root->right);
}


int main()
{
        nodeptr root = NULL;

        int n, m;
        printf("Enter number of nodes: ");
        scanf("%d", &n);

        for (int i = 0; i < n; i++)
        {
                printf("Enter value:  ");
                scanf("%d", &m);
                insertTree(&root, m);
        }

        printf("\nThe no. of nodes of the tree is: %d", countNodes(root));
        printf("\nOpcount is: %d", opcount);

        return 0;
}
```

Analysis:

$C(n)=1+2*C(n/2)$

$=1+2*(1+2*C(n/2^2)) = 3 + 2^2*C(n/2^2)$

$=1+2*(1+2*C(n/2^3)) = 7 + 2^3*C(n/2^3)$

. . .

$=2^k-1+2^k*C(n/2^k)$

Now, for base condition of the recurrence when $n/2^k = 1$. We get:

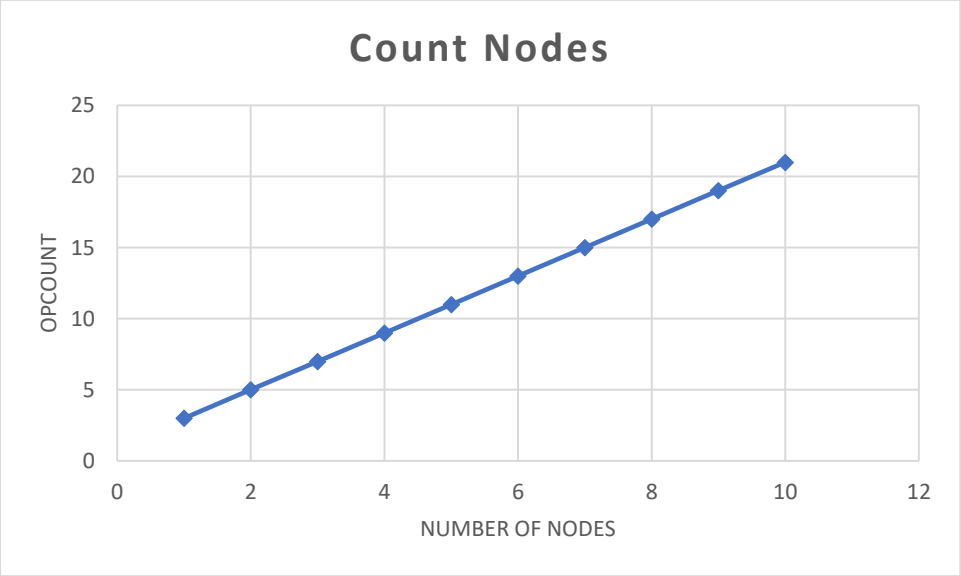
$C(n)=n - 1 + n*C(1) = n - 1 + n = 2*n + 1$

Therefore, $C(n) \in \theta(n)$.

Output:

```
Enter number of nodes: 3
Enter value:  5
Enter value:  22

Enter 1 to insert left of 5 or 2 to insert right: 1
Enter value:  33

Enter 1 to insert left of 5 or 2 to insert right: 2

The no. of nodes of the tree is: 3
Opcount is: 7
Process returned 0 (0x0)   execution time : 27.826 s
Press any key to continue.
```

Graph:



2). Write a program to sort given set of integers using Quick sort and analyse its efficiency. Obtain the experimental result of order of growth and plot the result.

Program:

```
#include<stdio.h>
int opcount = 0;
void swap(int* a, int* b)
{
        int t = *a;
        *a = *b;
        *b = t;
}

int partition (int arr[], int low, int high)
```

```
{
	int pivot = arr[high];
	int i = (low - 1);

	for (int j = low; j <= high- 1; j++)
	{
		opcount++;
		if (arr[j] < pivot)
		{
			i++;
			swap(&arr[i], &arr[j]);
		}
	}
	swap(&arr[i + 1], &arr[high]);
	return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
	if (low < high)
	{
		int pi = partition(arr, low, high);

		quickSort(arr, low, pi - 1);
		quickSort(arr, pi + 1, high);
	}
}

void printArray(int arr[], int size)
{
	int i;
	for (i=0; i < size; i++)
		printf("%d ", arr[i]);
	printf("\n");
}

int main()
{
	int n;
	printf("Enter number of elements to be sorted: ");
	scanf("%d",&n);
	int arr[n];
	printf("\nEnter elements: ");
	for(int i=0;i<n;i++)
		scanf("%d",&arr[i]);

	printf("\nGiven array is: ");
	printArray(arr, n);
```

```
        quickSort(arr, 0, n-1);
        printf("\nSorted array: ");
        printArray(arr, n);
        printf("\nOpcount: %d", opcount);
        return 0;
}
```

Analysis:

Best-Case:
The number of key comparisons in the best case satisfies the recurrence
$C_{best}(n) = 2\ C_{best}\ (n/2) + n$ for $n > 1$, $C_{best}\ (1) = 0$.
According to the Master Theorem, $C_{best}\ (n) \in \theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

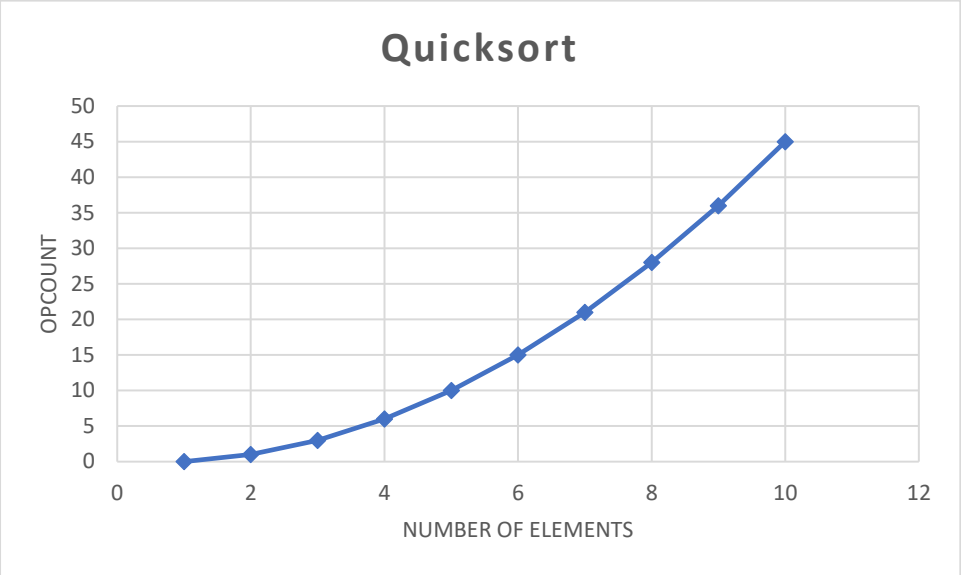
Worst-Case
The total number of key comparisons in worst-case made will be equal to
$C_{worst}(n) = (n + 1) + n + \ldots + 3 = [\{(n + 1)(n + 2)\}/2] - 3 \in \theta\ (n^2)$.

Average-Case
Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s ($0 \le s \le n - 1$) after n + 1comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and n - 1 - s elements, respectively. Assuming that the partition split can happen in each position s with the same probability 1/n, we get the following recurrence relation:
$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1}[(n + 1) + C_{avg}(s) + C_{avg}(n - 1 - s)]$ for $n > 1$,
$C_{avg}(0) = 0$, $C_{avg}(1) = 0$
$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log 2\ n$.

Output:

```
Enter number of elements to be sorted: 8

Enter elements: 45 23 123 35 11 99 10 0

Given array is: 45 23 123 35 11 99 10 0

Sorted array: 0 10 11 23 35 45 99 123

Opcount: 19
Process returned 0 (0x0)   execution time : 31.040 s
Press any key to continue.
```

Graph:



3). Write a program to sort given set of integers using Merge sort and analyse its efficiency. Obtain the experimental result of order of growth and plot the result.

Program:

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

int opcount=0;

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[n1], R[n2];

        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)
        {
                opcount++;
                if (L[i] <= R[j])
                {
                        arr[k] = L[i];
                        i++;
                }
                else
```

```
                {
                        arr[k] = R[j];
                        j++;
                }
                k++;
        }

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

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

void mergeSort(int arr[], int l, int r)
{
        if (l < r)
        {
                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)
{
        int i;
        for (i=0; i < size; i++)
                printf("%d ", A[i]);
        printf("\n");
}

int main()
{
        int n;
        printf("Enter number of elements to be sorted: ");
```

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

    printf("\nGiven array is: ");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("\nSorted array is: ");
    printArray(arr, n);
    printf("\nOpcount: %d", opcount);
    return 0;
}
```

Analysis:

The recurrence relation for the number of key comparisons C(n) is

$C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$, $C(1) = 0$.

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence:

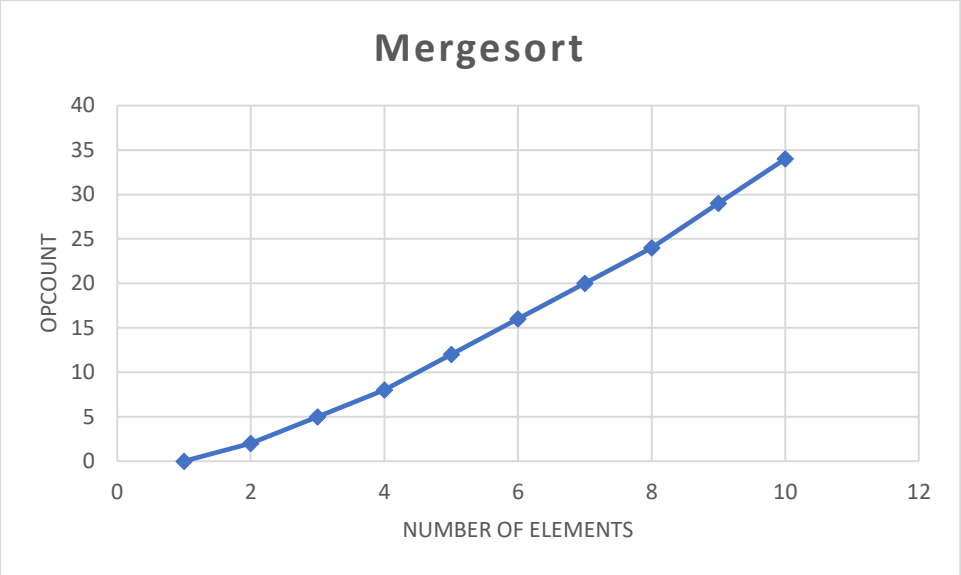
$C_{worst}(n) = 2C_{worst}(n/2) + n - 1$ for $n > 1$, $C_{worst}(1) = 0$.

Hence, according to the Master Theorem, $C_{worst}(n) \in \theta (n \log n)$.

Output:

```
Enter number of elements to be sorted: 9

Enter elements: 13 15 134 0 411 41 4315 95 322

Given array is: 13 15 134 0 411 41 4315 95 322

Sorted array is: 0 13 15 41 95 134 322 411 4315

Opcount: 29
Process returned 0 (0x0)   execution time : 29.939 s
Press any key to continue.
```

Graph:


Mergesort
OPCOUNT
0
5
10
15
20
25
30
35
40
0
2
4
6
8
10
12
NUMBER OF ELEMENTS