Lab 1

Performance analysis

Objective:
The objective of this experiment is to study the analytical and experimental ways of evaluating the performance of different programs in terms of time and space complexity.

Introduction:
Performance Analysis of Algorithms and Data Structures is carried out in terms of:

1) Space complexity.
2) Time complexity.

Space Complexity of a program is the amount of memory it needs to run to completion.

Time Complexity of a program is the amount of computer time it needs to run to completion.

There are two methods to determine the performance of a program, one is analytical, and the other is experimental.

Asymptotic Notation:
Asymptotic analysis of an algorithm or data structure refers to defining the mathematical boundaries of its performance. Using asymptotic analysis, we can conclude the best case, average case, and worst-case scenario of an algorithm. The **BIG-Oh** is the formal way to express the upper bound or worst-case of a program's time and space complexity.

Analytical Analysis:
Analytically finding time complexity of the following code is calculated in Table 1.1:

```
int sum, i, j;
sum = 0;
for (i=0;i<n;++i)
{       for(j=0;j<n;++j)
        {       sum++;
        }
}
```

Result:

Statement	Number of times executed
sum=0	1
i=0	1
i<n	n+1
++i	N
j=0	N
j<n	$n(n+1) = n^2 + n$
++j	n^2
sum++	n^2
Total	**$3n^2+4n+3$**
	$T(n) = 3n^2+4n+3$, $T(n) = O(n^2)$

Table 1.1

Exercise 1: For the following codes carry out the analytical analysis to evaluate time complexity T(n) and then express it in terms of Big Oh:

```
int sum,i;
sum = 0;
for (i=0;i<n;++i)
    sum++;
```

T(n) = 4n + 2
Big O = n

```
int sum,i,j;
sum = 0;
for (i=1;i<n; i=i*2)
    sum++;
```

1 — sum = 0
log2(n) — i = i*2
log2(n) — sum++

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

```
int sum,i,j;
sum = 0;
for (i=1;i<n; ++i)
{
for (j=0;j< i;++j)
    {   sum++;
    }
}
```

T(n) = $\frac{n^2+n}{2}$
Big O = n^2

1 + 2 + 3 ... = n(n+1)/2

= (n^2 + 1) / 2

$= \frac{n^2+n}{2}$

```
int sum,i,j;
sum = 0;
for (i=0;i<n;++i)        n + 1
{
for (j=0;j<n;++j)        n(n+1)
    {   sum++;           n*n
    }
}
```

T(n) = 2n^2+2n+1
Big O = n^2

```
int sum,i,j;
sum = 0;
for (i=1;i<n; i=i*2)
{
for (j=0;j<n;++j)        n.log2 n
    {   sum++;
    }                    → n log2 n
}
```

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

```
int sum,i,j;
sum = 0;
for (i=1;i<n; i=i*2)
{
for (j=0;j<i; j++)
    {   sum++;
    }
}
```

T(n) =
Big O =

j<i doubles with each iteration of the outer loop, the inner loop runs 1,2,4,8,… times across successive iterations of the outer loop.

T(n) = 1+2+4+8+..+2^k=2^k-1

outer loop runs

$2^{\log_2 n} = n$

$T(n) \approx 2n - 1$

$B(o) = n$

Experimental Analysis:

To find time complexity of a code experimentally we need to calculate the time difference between starting and ending time of execution of code. So "**time.h"** library is needed for this purpose.

```
#include<iostream>
#include<time.h>
using namespace std;
void main()
{
clock_t start, end;
double cpu_time_used;

start = clock();
                {     ... Code  ...     }
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
          cout<<cpu_time_used<<" seconds";
}
```

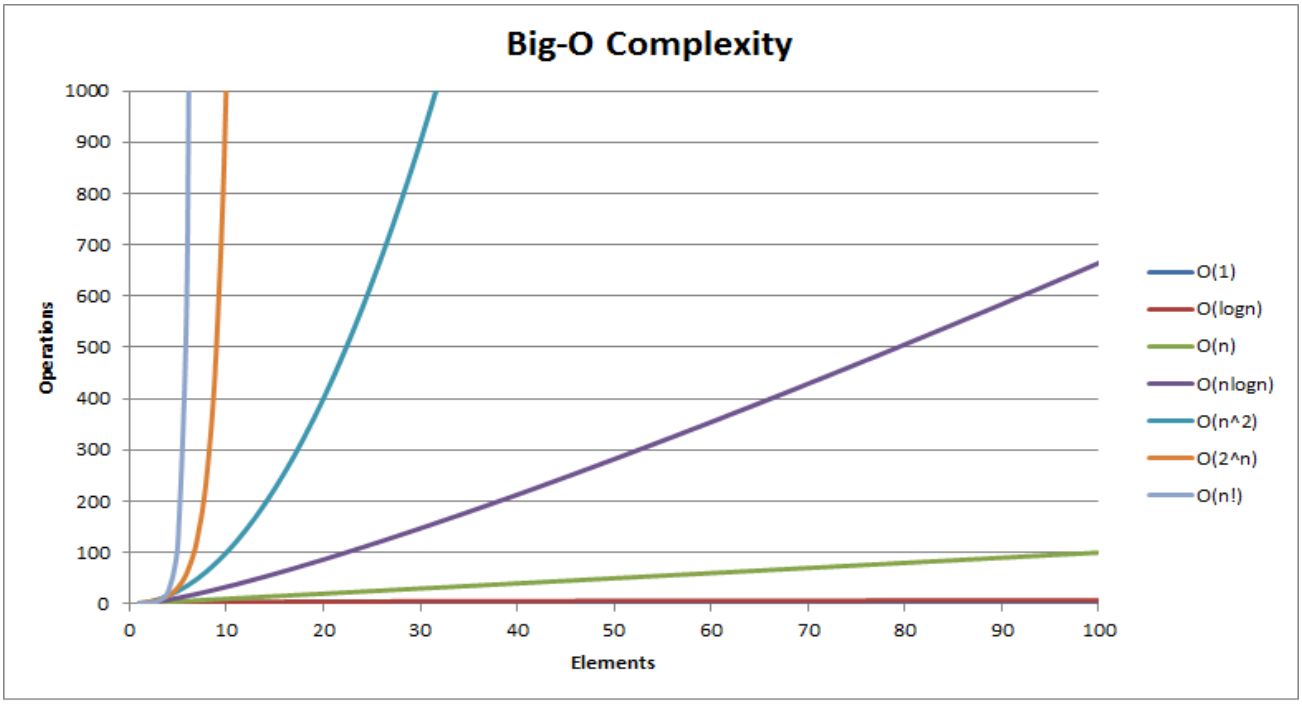

Figure 1.1: Asymptotic notation

Exercise2: Experimental Analysis

Your task is to write a program that takes the following numbers as values of n:

50000, 100000, 500000, 1000000, 5000000

- For the codes given in table 1 find the execution time for each value of n given in the array above.
- Then plot separate graphs for time of execution vs “n”, for all the values of n
- Use excel for plotting graph
- Verify that the graphs you plotted matches the asymptotic notation shown in figure 1.1.
- Sketch or paste the graphs in table 1.2.

Table1.2

Exercise3:

Write a program to find the factorial of a given number

i. Iteratively
ii. Recursively.

Describe the difference in terms of time and space complexity, on top of the code in comments.

POST LAB:

Following is a sorting algorithm. Your task is to run the code for various values of n (you may use the same values of n that you did your inLab for) and note the execution time taken by the sort method. By comparing this execution time graph with the ones in the inLab evaluate the Big Oh.Is this code Iterative or recursive?

```
#include<iostream>
#include<time.h>
Using namespace std;
void print(int *a, int n)
{
	int i =0;
	while(i<n)
	{
		cout<<a[i]<<",";
		i++;
	}
}
void swap(int i,int j, int *a)
{
	int temp = a[i];
	a[i] = a[j];
	a[j] = temp;
}

void sort(int *arr, int left, int right)
{
	int min = (left+right)/2;

	int i = left;
	int j = right;
	int pivot = arr[min];
	while(left<j || i<right)
	{
		while(arr[i]<pivot)
			i++;
		while(arr[j]>pivot)
			j--;
	if(i<=j)
		{
			swap(i,j,arr);
			i++;
			j--;
		}
		else
		{
			if(left<j)
				sort(arr, left, j);
			if(i<right)
				sort(arr,i,right);
			return;
		}
	} // while ends
} // sort ends

int main()
{
	int n=10000;
	int *arrA=new int[n];

	//generate n random numbers
	for (int i=0; i<n; i++)
		arrA[i]=rand()%100;

	clock_t start, end;
	double cpu_time_used;
	start = clock();
	sort(arrA, 0, n-1);
	end = clock();
	cpu_time_used = ((double) (end - start)) /
CLOCKS_PER_SEC;
	cout<<cpu_time_used<<" seconds";

	if(n<=100)
		print(arrA, n);
	return 0;
}
```