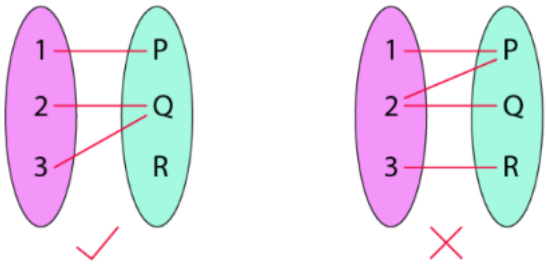
Name	ARYAMAN AGARWAL
UID no.	2021700002
Experiment No.	1

AIM:	Experiment on finding the running time of an algorithm and comparison of time complexities for different sorting methods.
Program 1-A	
PROBLEM STATEMENT :	To implement 10 functions, take the input (i.e. n) to all the above functions varies from 0 to 100 with increment of 1. Then add the function n! in the list and execute the same for n from 0 to 20.
THEORY:	A function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output. Let A & B be any two non-empty sets; mapping from A to B will be a function only when every element in set A has one end, only one image in set B. 
ALGORITHM:	1. Start 2. Create 10 functions namely F1,F2, …….. till F10 each performing different mathematical operation. 3. Declare a variable N which will get the values from 1 to 100. 4. Start a for loop which will get the values of different functions for 1 to 100 values. 5. Print the value of all the 10 functions in each loop iteration. 6. Declare a long integer type variable initializing it with 1.

	7. Start a for loop iterating from 1 to 20 and in each iteration print the value of factorial with factorial equals to factorial multiplied with loop iteration number. 8. End
PROGRAM:	#include <stdio.h> #include<stdlib.h> #include <math.h> double F1(int n){ return n*n*n; } int F2(int n){ return n; } double F3(int n){ return pow(2,n); } double F4(int n){ return log(n); } double F5(int n){ return log(log(n)); } double F6(int n){

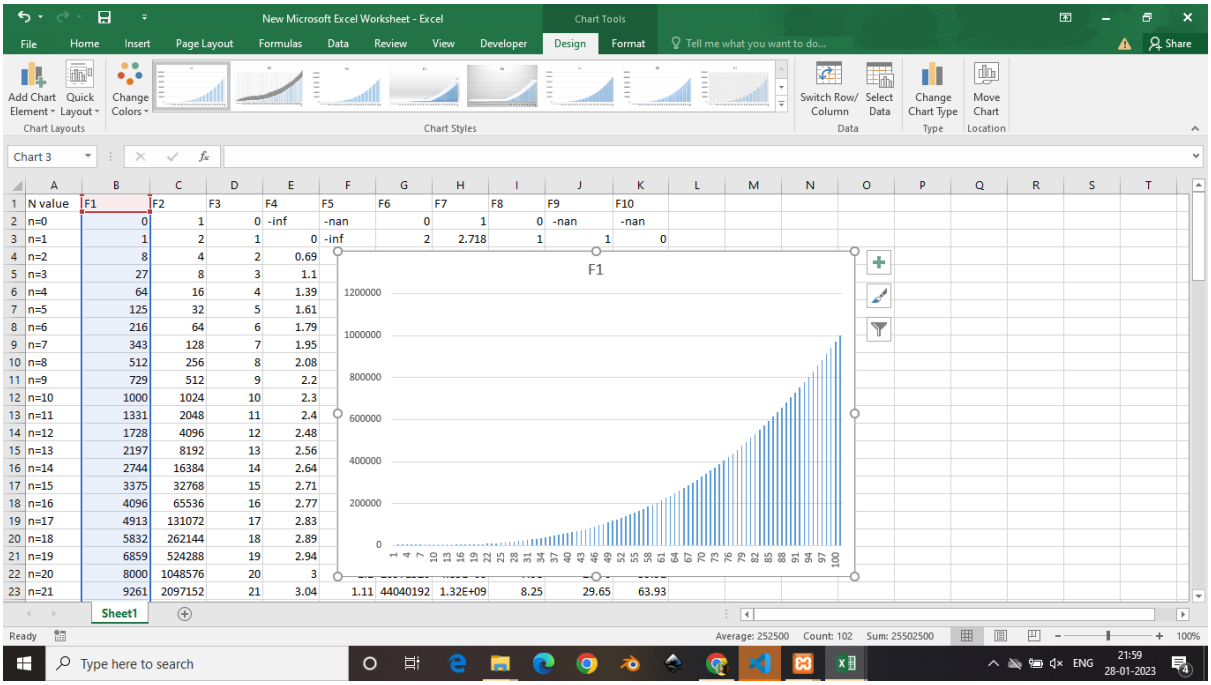
```
return n*pow(2,n);


}

double F7(int n){

return exp(n);


}

double F8(int n){

return pow(2,log(n));


}

double F9(int n){

return pow(n,log(log(n)));

}

double F10(int n){

return n*log(n);


}



int main()
{
   int n;

    printf("N value\tF1\tF2\tF3\tF4\tF5\tF6\tF7\tF8\tF9\tF10\n");
    for(int i=0;i<101;i++)
    {

printf("n=%d\t %.2f\t %.2f\t %d\t %.2f\t %.2f\t",i,F1(i),F2(i),F3(i),F4(i),F5(i));
```

```
        printf("%.3f\t %.3f\t %.2f\t  %.2f\t  %.2f\n",F6(i),F7(i),F8(i),F9(i),F10(i));
    }

    long int fact = 1;
    for (long int a = 1; a<= 20; a++)
    {
    fact = fact * a;
    printf(" %ld \n", fact);
    }
    return 0;

}
```

RESULT ANALYSIS:

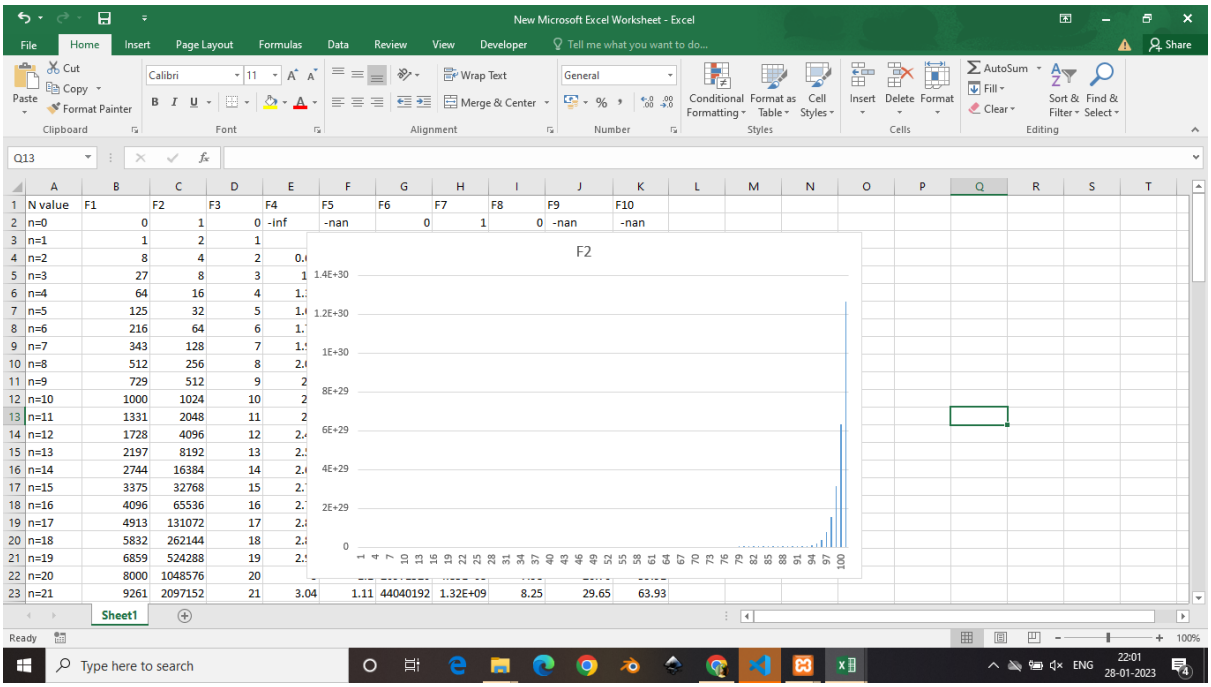
FUNCTIONS OUTPUT:

1. **n^3**



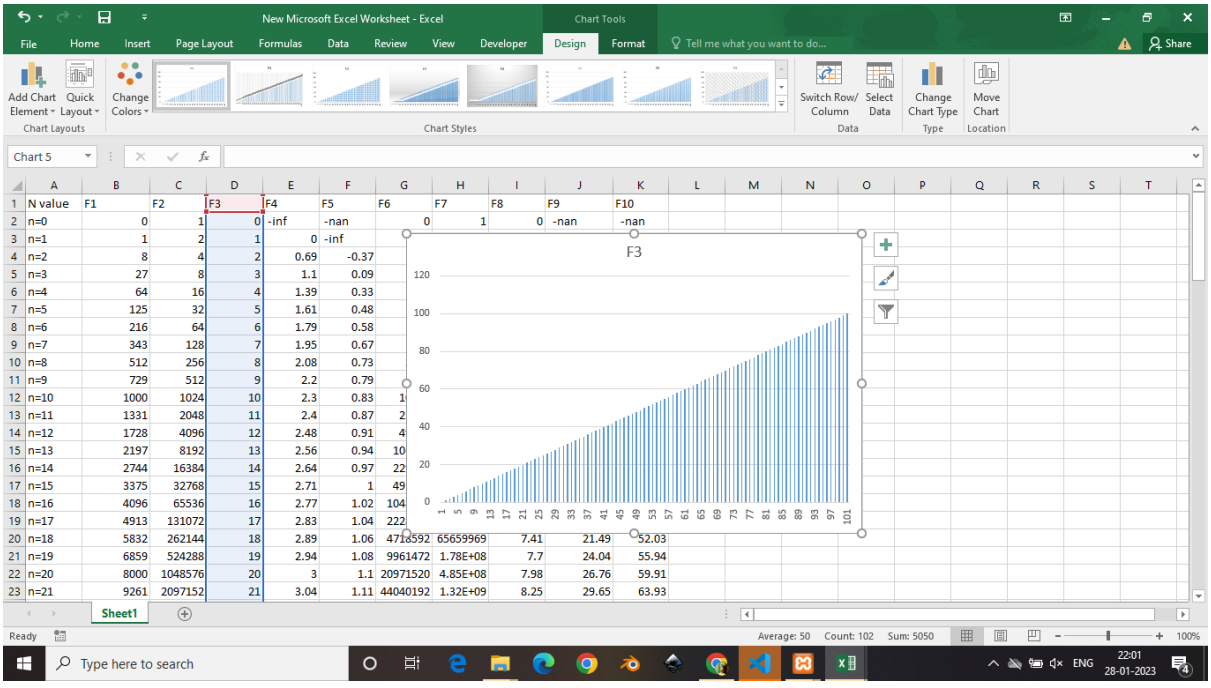
It is a cubic function. It increases exponentially throughout the values that range from 1 to 100. It would approach towards infinity as the value of I keeps on increasing. It increases by more than linear or logarithmic graph.

2. **n**



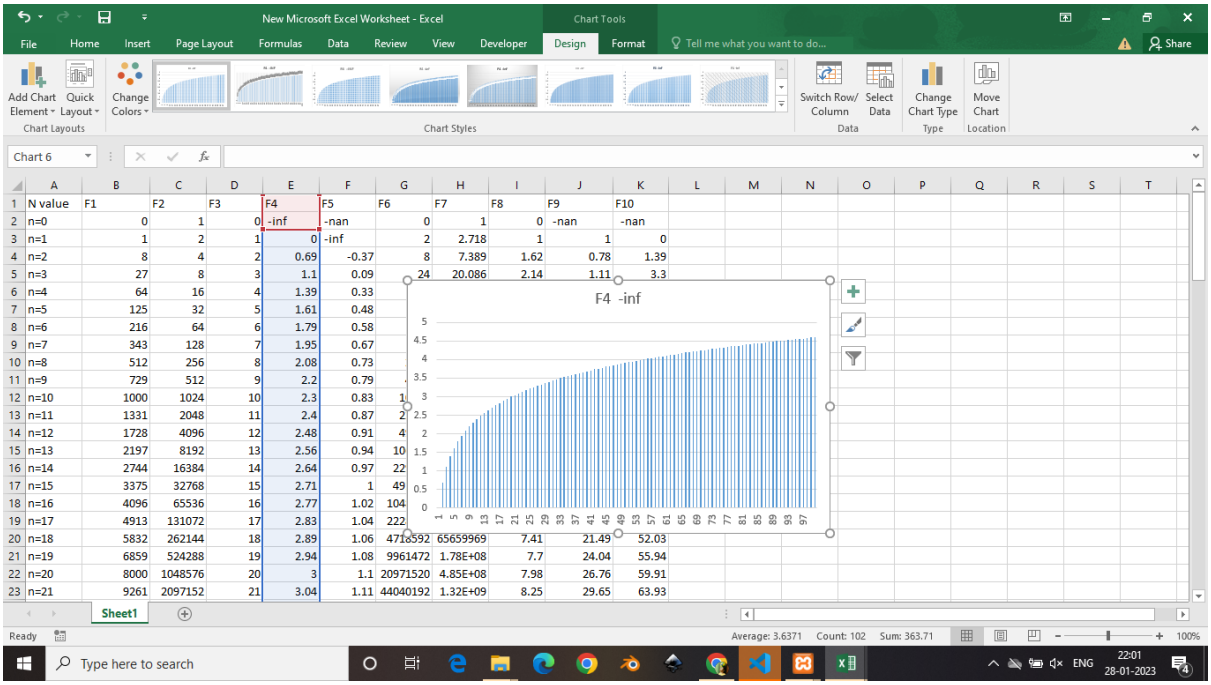
It is a linear graph which increases periodically and linearly. It also approaches the value of infinity but slowly in comparison to exponential or cubic functions. In this graph the value at y axis is always equal to x axis.

3. 2^n



It also is an exponential graph that increases and will reach to the value of infinity as the value of n keeps on increasing.

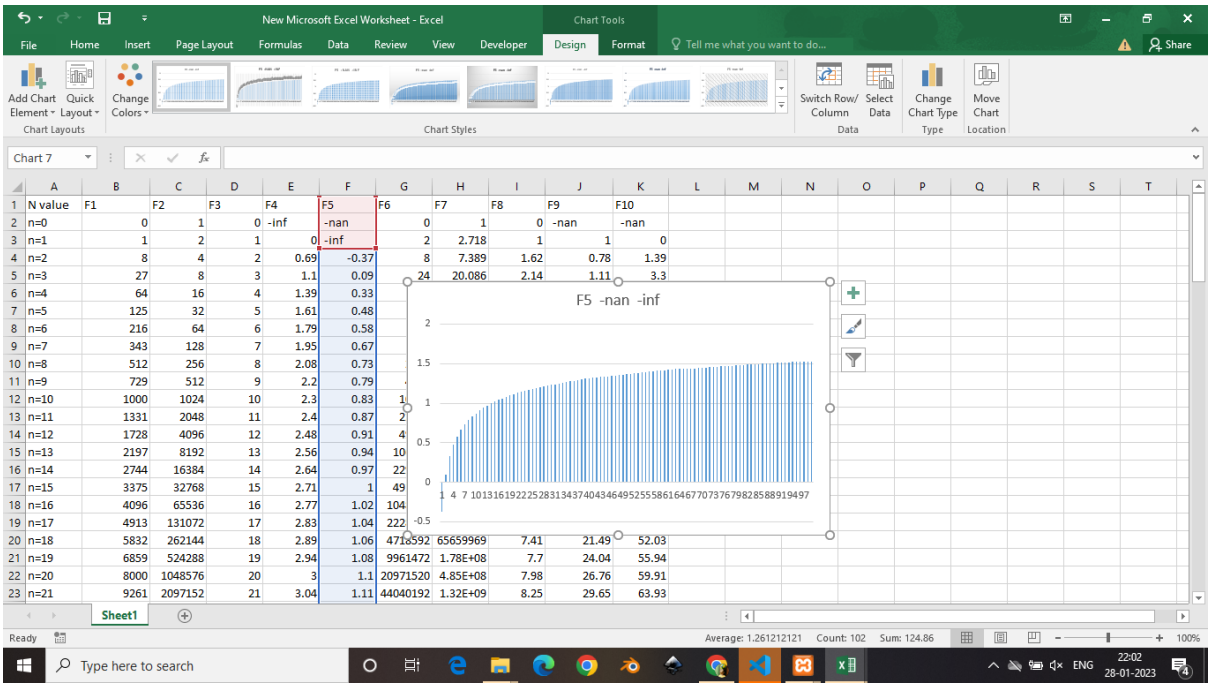
4. Log(n)



N value	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
n=0	0	1	0	-inf	-nan	0	1	0	-nan	-nan
n=1	1	2	1	0	-inf	2	2.718	1	1	0
n=2	8	4	2	0.69	-0.37	8	7.389	1.62	0.78	1.39
n=3	27	8	3	1.1	0.09	24	20.086	2.14	1.11	3.3
n=4	64	16	4	1.39	0.33					
n=5	125	32	5	1.61	0.48					
n=6	216	64	6	1.79	0.58					
n=7	343	128	7	1.95	0.67					
n=8	512	256	8	2.08	0.73					
n=9	729	512	9	2.2	0.79					
n=10	1000	1024	10	2.3	0.83	1				
n=11	1331	2048	11	2.4	0.87	2				
n=12	1728	4096	12	2.48	0.91	4				
n=13	2197	8192	13	2.56	0.94	10				
n=14	2744	16384	14	2.64	0.97	22				
n=15	3375	32768	15	2.71	1	49				
n=16	4096	65536	16	2.77	1.02	104				
n=17	4913	131072	17	2.83	1.04	222				
n=18	5832	262144	18	2.89	1.06	4718592	65659969	7.41	21.49	52.03
n=19	6859	524288	19	2.94	1.08	9961472	1.78E+08	7.7	24.04	55.94
n=20	8000	1048576	20	3	1.1	20971520	4.85E+08	7.98	26.76	59.91
n=21	9261	2097152	21	3.04	1.11	44040192	1.32E+09	8.25	29.65	63.93

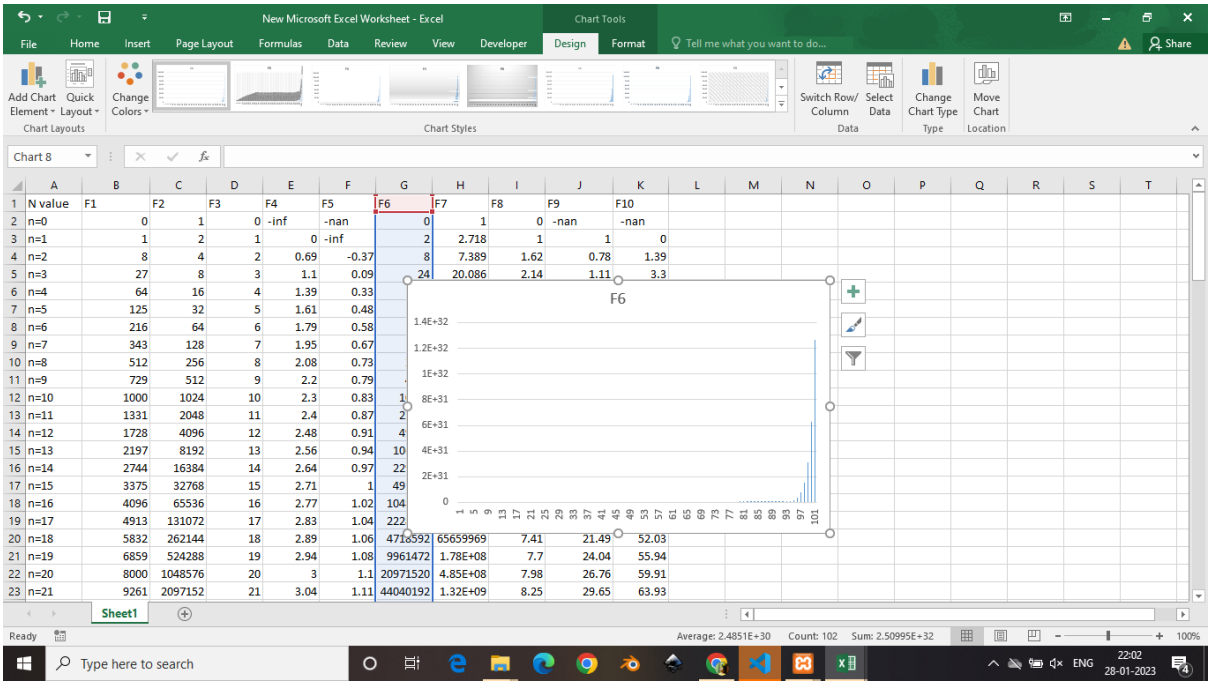
It is a logarithmic graph that first has a steep increasing curve at the start and as the value of n keeps on increasing the graph plateaus and increases less steeply.

5. Log(log(n))



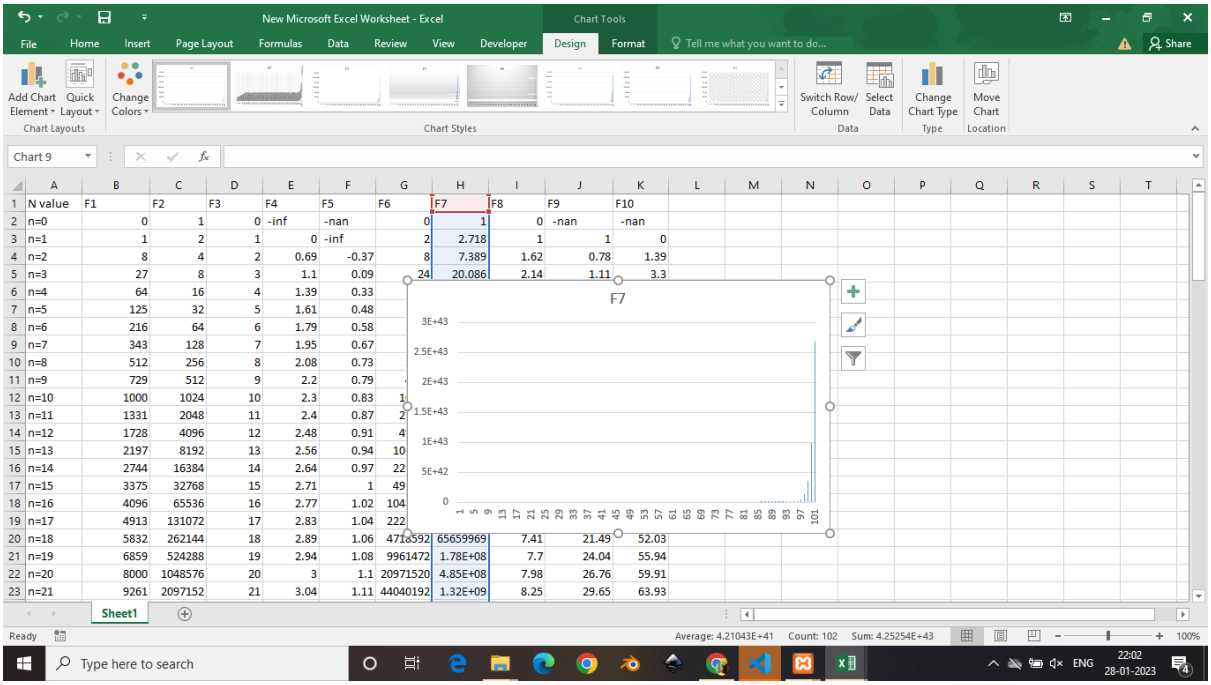
It is also a logarithmic graph that first has a steep increasing curve at the start and as the value of n keeps on increasing the graph plateaus and increases less steeply.

6. n*2^n



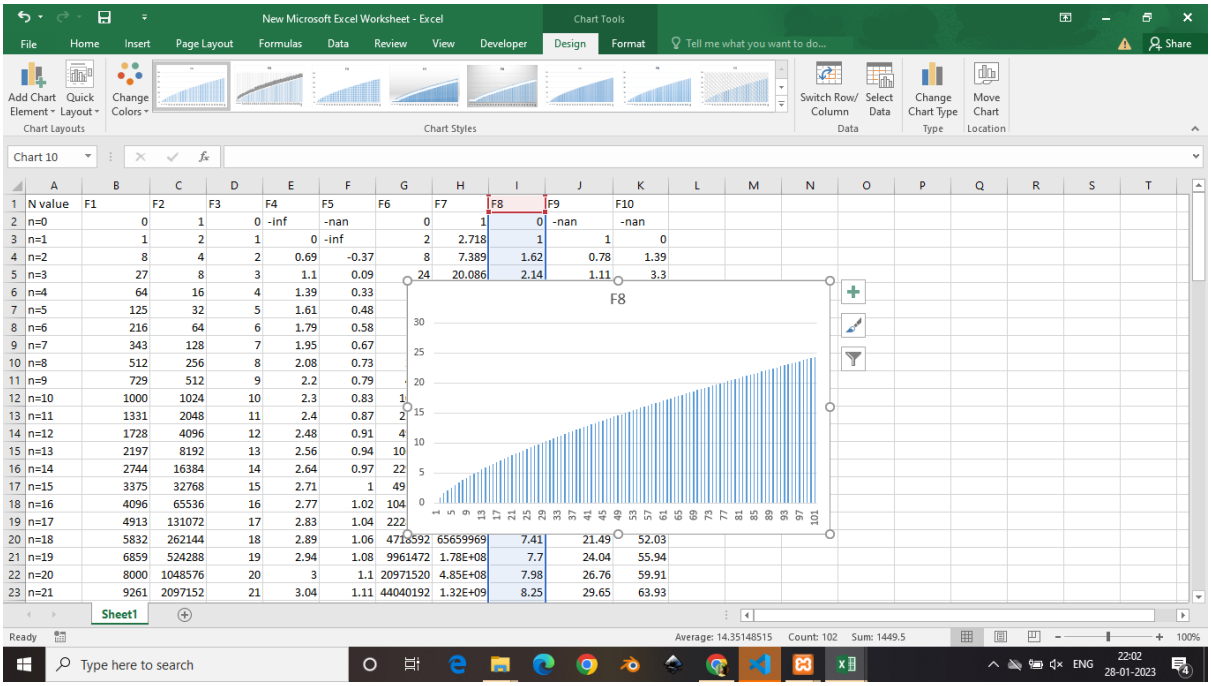
It also is an exponential graph as 2 power n is an exponential ,that increases and will reach to the value of infinity as the value of n keeps on increasing.

7. e^n



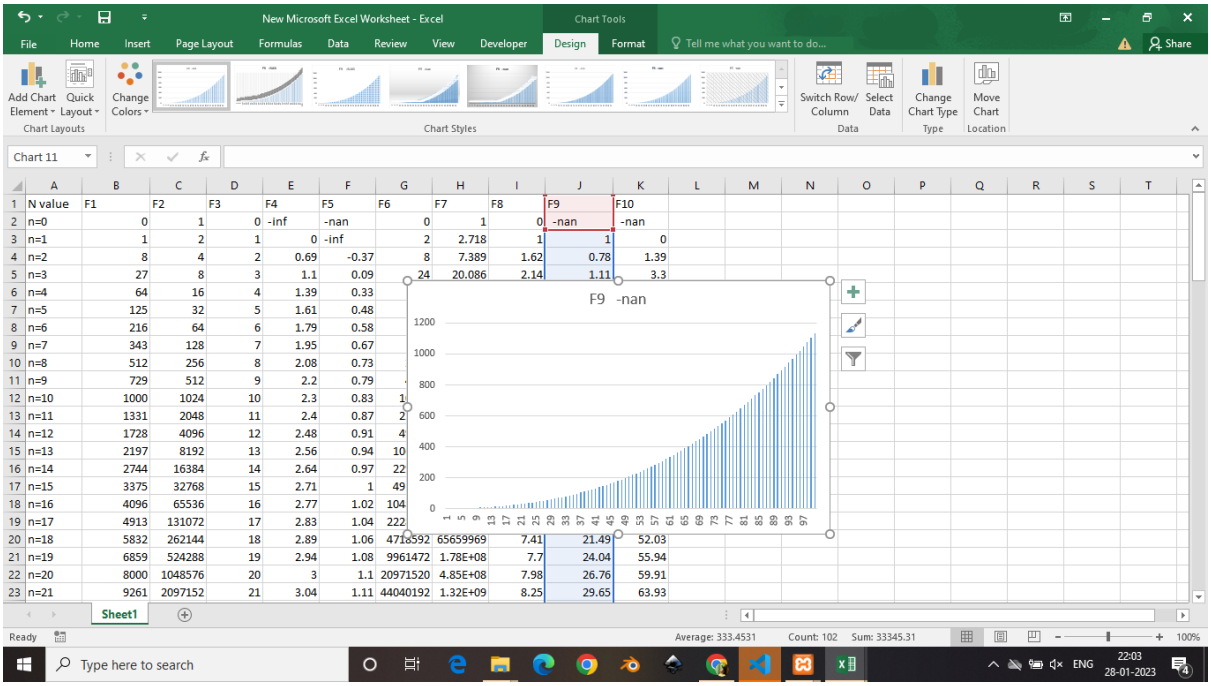
It also is an exponential graph as e power n is an exponential ,that increases and will reach to the value of infinity as the value of n keeps on increasing.

8. 2^log(n)



It is an exponential graph that increases and will reach to the value of infinity as the value of n keeps on increasing. The base here is constant number 2 but the power here is increasing in logarithmic function.

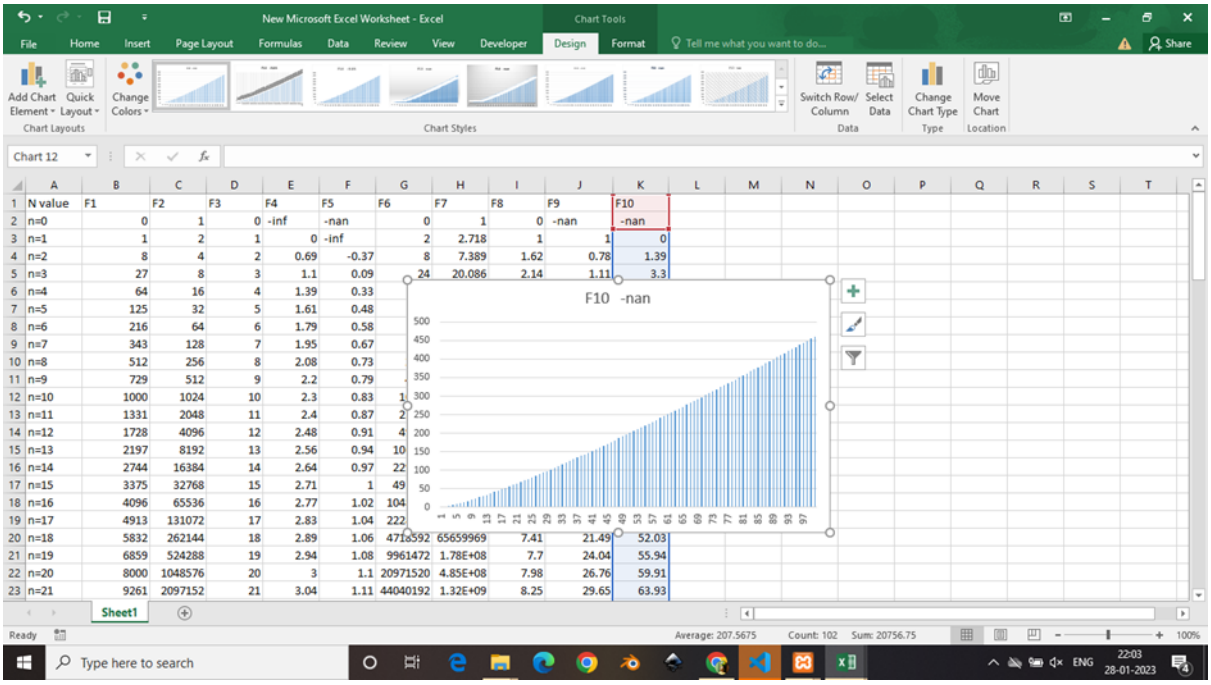
9. n^log(log(n))



N value	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
n=0	0	1	0	-inf	-nan	0	1	0	-nan	-nan
n=1	1	2	1	0	-inf	2	2.718	1	1	0
n=2	8	4	2	0.69	-0.37	8	7.389	1.62	0.78	1.39
n=3	27	8	3	1.1	0.09	24	20.086	2.14	1.11	3.3
n=4	64	16	4	1.39	0.33					
n=5	125	32	5	1.61	0.48					
n=6	216	64	6	1.79	0.58					
n=7	343	128	7	1.95	0.67					
n=8	512	256	8	2.08	0.73					
n=9	729	512	9	2.2	0.79					
n=10	1000	1024	10	2.3	0.83	1				
n=11	1331	2048	11	2.4	0.87	2				
n=12	1728	4096	12	2.48	0.91	4				
n=13	2197	8192	13	2.56	0.94	10				
n=14	2744	16384	14	2.64	0.97	22				
n=15	3375	32768	15	2.71	1	49				
n=16	4096	65536	16	2.77	1.02	104				
n=17	4913	131072	17	2.83	1.04	222				
n=18	5832	262144	18	2.89	1.06	4718592	65659969	7.41	21.49	52.03
n=19	6859	524288	19	2.94	1.08	9961472	1.78E+08	7.7	24.04	55.94
n=20	8000	1048576	20	3	1.1	20971520	4.85E+08	7.98	26.76	59.91
n=21	9261	2097152	21	3.04	1.11	44040192	1.32E+09	8.25	29.65	63.93

It is an exponential graph that increases and will reach to the value of infinity as the value of n keeps on increasing. The base here is increasing linearly but the power here is increasing in logarithmic function. It has a steeper increasing curve than the graph in the previous function.

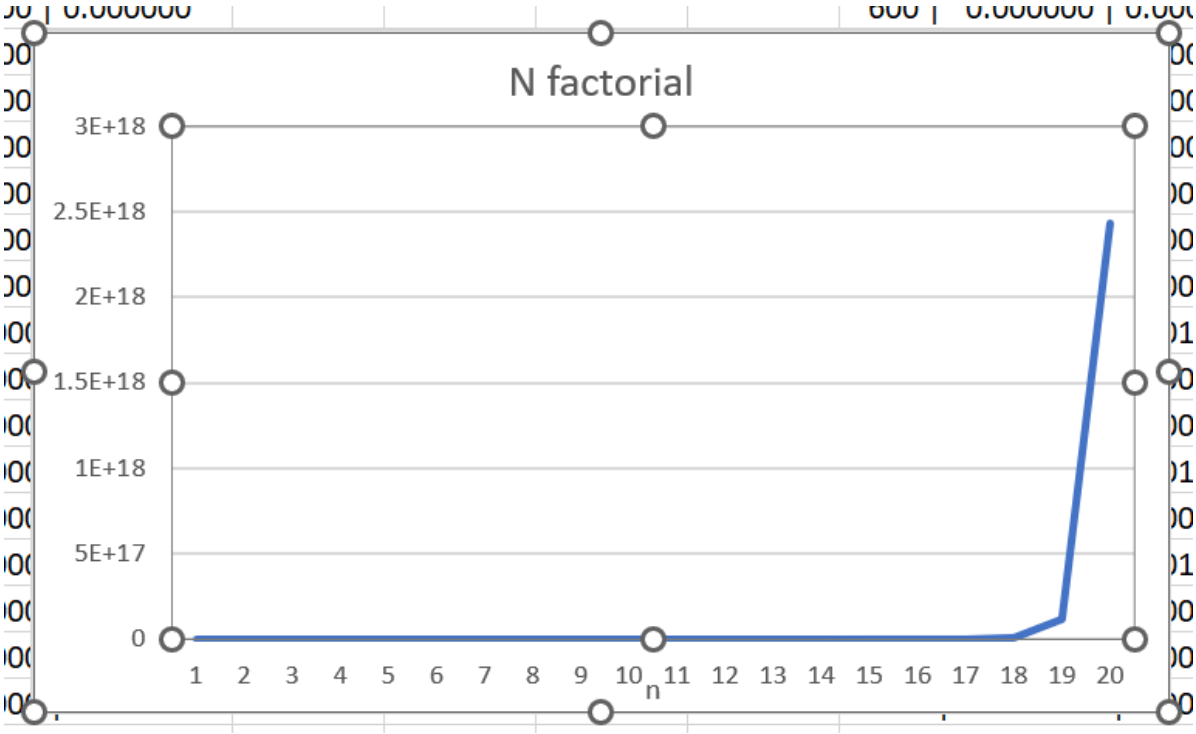
10.n*log(n)



N value	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
n=0	0	1	0	-inf	-nan	0	1	0	-nan	-nan
n=1	1	2	1	0	-inf	2	2.718	1	1	0
n=2	8	4	2	0.69	-0.37	8	7.389	1.62	0.78	1.39
n=3	27	8	3	1.1	0.09	24	20.086	2.14	1.11	3.3
n=4	64	16	4	1.39	0.33					
n=5	125	32	5	1.61	0.48					
n=6	216	64	6	1.79	0.58					
n=7	343	128	7	1.95	0.67					
n=8	512	256	8	2.08	0.73					
n=9	729	512	9	2.2	0.79					
n=10	1000	1024	10	2.3	0.83	1				
n=11	1331	2048	11	2.4	0.87	2				
n=12	1728	4096	12	2.48	0.91	4				
n=13	2197	8192	13	2.56	0.94	10				
n=14	2744	16384	14	2.64	0.97	22				
n=15	3375	32768	15	2.71	1	49				
n=16	4096	65536	16	2.77	1.02	104				
n=17	4913	131072	17	2.83	1.04	222				
n=18	5832	262144	18	2.89	1.06	4718592	65659969	7.41	21.49	52.03
n=19	6859	524288	19	2.94	1.08	9961472	1.78E+08	7.7	24.04	55.94
n=20	8000	1048576	20	3	1.1	20971520	4.85E+08	7.98	26.76	59.91
n=21	9261	2097152	21	3.04	1.11	44040192	1.32E+09	8.25	29.65	63.93

It is an exponential graph that increases and will reach to the value of infinity as the value of n keeps on increasing. The base here is increasing linearly but the power here is increasing in logarithmic function.

11. N factorial



It is a factorial function. The factorial function always overtakes an exponential function. Factorial function has the most steepest curve of the following functions and its value reaches to infinity the fastest. Hence we have taken the value of n only till 20.

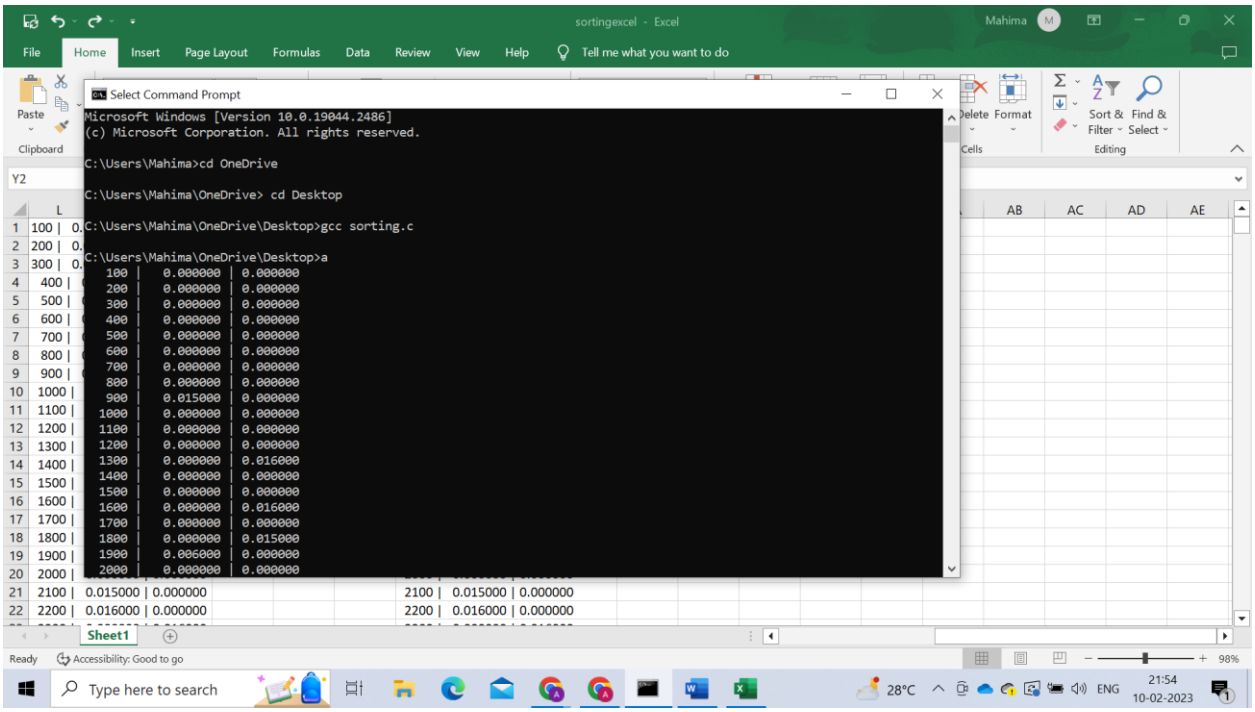
Program 1-B	
PROBLEM STATEMENT :	Implement two sorting algorithms namely Insertion and Selection sort methods. Compare these algorithms based on time and space complexity. Time required to sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono. Compare two algorithms namely Insertion and Selection by plotting the time required to sort 100000 integers.
ALGORITHM/ THEORY:	**Selection Sort:** In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

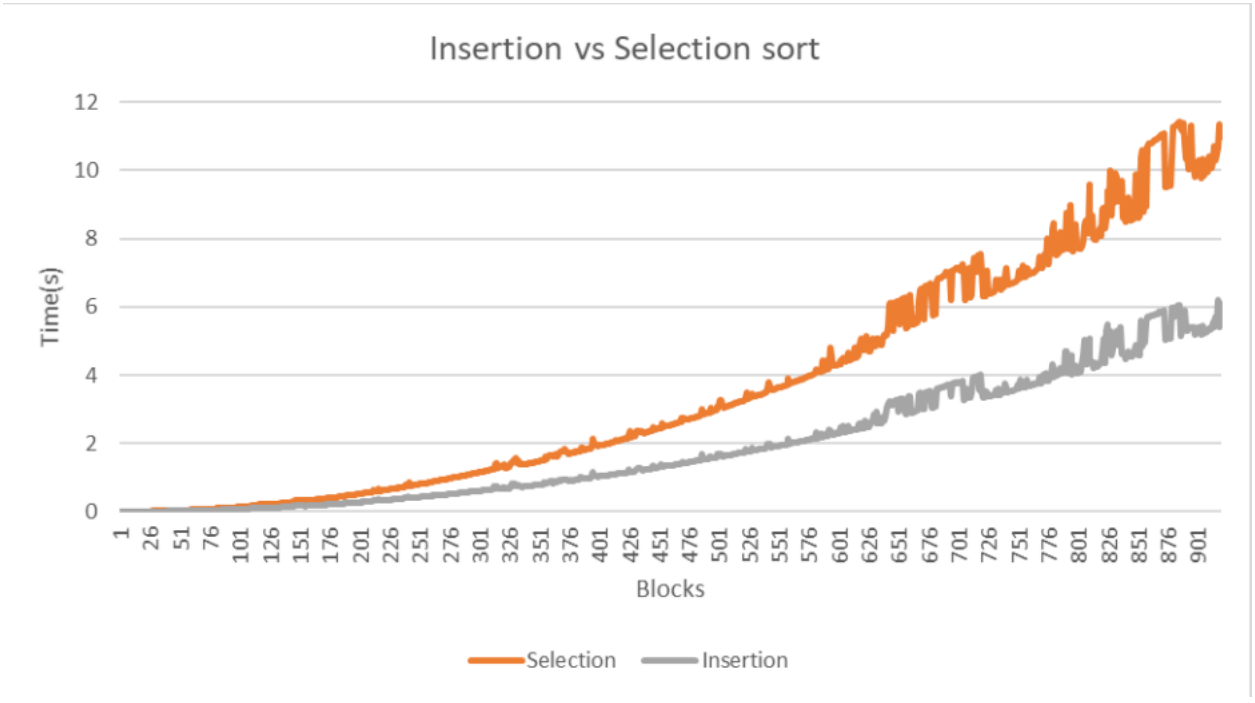
	The average and worst-case complexity of selection sort is O(n2), where n is the number of items. Due to this, it is not suitable for large data sets. Step 1 − Set MIN to location ith index. Step 2 − Search the minimum element in the list Step 3 − Swap with value at location MIN Step 4 − Increment MIN to point to next element Step 5 − Repeat until list is sorted **Insertion Sort:** The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is O(n2), where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc. Step 1 - If the element is the first element, assume that it is already sorted. Step2 - Pick the next element, and store it separately in a key. Step3 - Now, compare the key with all elements in the sorted array. Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right. Step 5 - Insert the value. Step 6 - Repeat until the array is sorted. Time Complexity of selection sort is always n*(n - 1)/2, whereas insertion sort has better time complexity as its worst case complexity is n*(n - 1)/2. Generally it will take lesser or equal comparisons then n*(n - 1)/2.
PROGRAM:	```c #include <stdio.h> #include <time.h> void selectionSort(int *arr, int len) { int min_i, temp; for (int i = 0; i < len; i++) { min_i = i; ```

```
        for (int j = i + 1; j < len; j++)
        {
            if (arr[j] < arr[min_i])
            {
                min_i = j;
            }
        }
        if (i != min_i)
        {
            temp = arr[min_i];
            arr[min_i] = arr[i];
            arr[i] = temp;
        }
    }
}

void insertionSort(int *arr, int len)
{
    int key;
    for (int i = 1; i < len; i++)
    {
        key = arr[i];
        for (int j = 0; j < i; j++)
        {
            if (arr[j] > key)
            {
                for (int k = i; k > j; k--)
                {
                    arr[k] = arr[k - 1];
                }
                arr[j] = key;
                break;
            }
        }
    }
}

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

```
}

int deepCopy(int *source, int *dest, int len)
{
    for (int i = 0; i < len; i++)
    {
        dest[i] = source[i];
    }
}

int main()
{
    FILE *fptr = fopen("randomnumbers.txt", "r");
    clock_t start, end;
    double time1, time2;
    for (int i = 100; i <= 100000; i += 100)
    {
        int arr1[i];
        int arr2[i];
        for (int j = 0; j < i; j++)
        {
            fscanf(fptr, "%d", &arr1[j]);
        }
        deepCopy(arr1, arr2, i);
        fseek(fptr, 0, SEEK_SET);
        start = clock();
        selectionSort(arr1, i);
        end = clock();
        time1 = (end - start) * 1.0 / CLOCKS_PER_SEC;
        start = clock();
        insertionSort(arr2, i);
        end = clock();
        time2 = (end - start) * 1.0 / CLOCKS_PER_SEC;
        printf("%6d | %10f | %f\n", i, time1, time2);
    }
    fclose(fptr);
}
```

RESULT ANALYSIS:





Insertion sort takes less time as compared to selection sort.
Both insertion sort and selection sort have an outer loop (over every index), and an inner loop

(over a subset of indices). Each pass of the inner loop expands the sorted region by one element, at the expense of the unsorted region, until it runs out of unsorted elements.

The difference is in what the inner loop does:

- In selection sort, the inner loop is over the unsorted elements. Each pass selects one element, and moves it to its final location (at the current end of the sorted region).
- In insertion sort, each pass of the inner loop iterates over the sorted elements. Sorted elements are displaced until the loop finds the correct place to insert the next unsorted element.

So, in a selection sort, sorted elements are found in output order, and stay put once they are found. Conversely, in an insertion sort, the unsorted elements stay put until consumed in input order, while elements of the sorted region keep getting moved around.

CONCLUSION:	• Implemented Selection sort and Insertion sort in c. • Calculated Time taken to sort each range of blocks by both algorithms. Selection sort: - Time Complexity: Worst Case: O(n2), Best Case: O(n2) - Space Complexity: O(1) Insertion sort: - Time Complexity: Worst Case: O(n2), Best Case: O(n) - Space Complexity: O(1) • Both algorithms have same time complexity for worst case, however insertion sort has better time complexity in some cases. Hence also by observing the graph we can conclude insertion sort is faster.