ALGORITHMS LABORATORY
[CS-2098]
Individual Work

Lab. No.-2 Date.: 23rd July

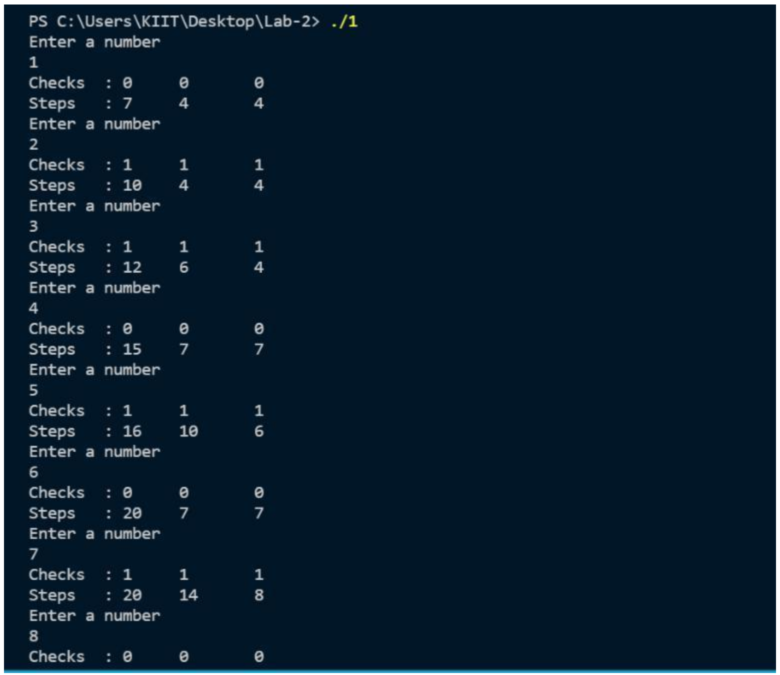
Roll Number:	1905083	**Branch/Section:**	CSE/CS2
Name in Capital:	AMIL UTKARSH GUPTA		

2.1:

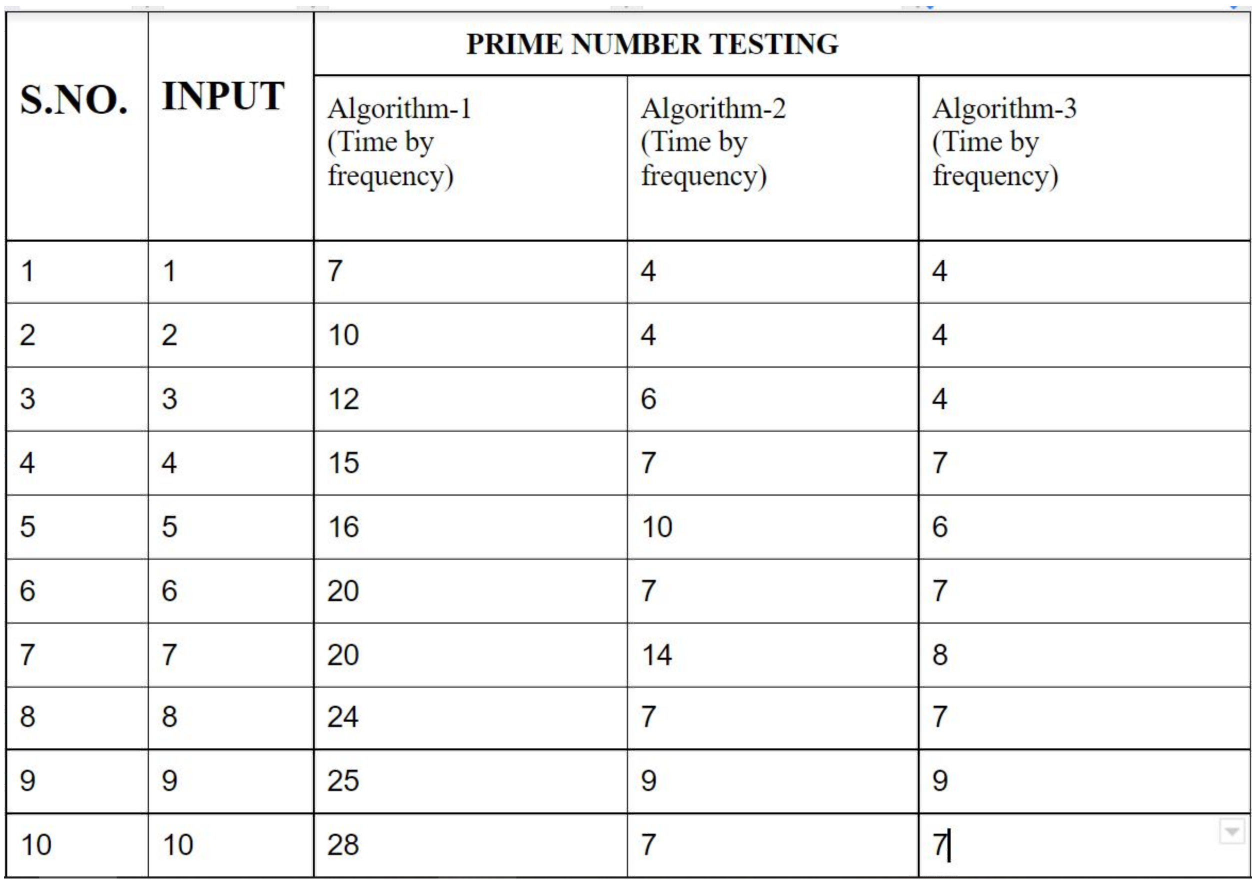
Program Title:

Write a program to test whether a number n, entered through keyboard is prime or not by using different algorithms you know for atleast 10 inputs and note down the time complexity by step/frequency count method for each algorithm & for each input. Finally make a comparision of time complexities found for different inputs, plot an appropriate graph & decide which algothm is faster.

Output:



```
PS C:\Users\KIIT\Desktop\Lab-2> ./1
Enter a number
1
Checks  : 0     0       0
Steps   : 7     4       4
Enter a number
2
Checks  : 1     1       1
Steps   : 10    4       4
Enter a number
3
Checks  : 1     1       1
Steps   : 12    6       4
Enter a number
4
Checks  : 0     0       0
Steps   : 15    7       7
Enter a number
5
Checks  : 1     1       1
Steps   : 16    10      6
Enter a number
6
Checks  : 0     0       0
Steps   : 20    7       7
Enter a number
7
Checks  : 1     1       1
Steps   : 20    14      8
Enter a number
8
Checks  : 0     0       0
```



S.NO.	INPUT	PRIME NUMBER TESTING		
		Algorithm-1 (Time by frequency)	Algorithm-2 (Time by frequency)	Algorithm-3 (Time by frequency)
1	1	7	4	4
2	2	10	4	4
3	3	12	6	4
4	4	15	7	7
5	5	16	10	6
6	6	20	7	7
7	7	20	14	8
8	8	24	7	7
9	9	25	9	9
10	10	28	7	7

Source Code:

```
#include<stdio.h>

int isPrime1(int n, int *count)

{

    int i, c=0;

    (*count)++;

    for(i=1; i<=n; i++)

    {

        (*count)++;

        if(n % i == 0)

        {
```

```
        c++;
        (*count)++;
    }
}
(*count) += i;
```

```
    (*count)++;
    if(c == 2)
    {
        (*count)++;
        return 1;
    }
    (*count)++;
    return 0;
}
int isPrime2(int n, int *count)
{
    int i, c=0;
    (*count)++;
    for(i=2; i<n; i++)
    {
        (*count)++;
        if(n%i == 0)
        {
            c++;
            (*count)++;
            (*count)++;
            break;
        }
    }
    (*count) += i-1;
    (*count)++;
    if(c == 0 && n>1)
    {
        (*count)++;
        return 1;
    }
    (*count)++;
    return 0;
}
int isPrime3(int n, int *count)
{
    int i, c=0;
    (*count)++;
```

```
 for(i=2; i <= n/2; i++)
  {
      (*count)++;
      if(n%i == 0)
      {
          c++;
          (*count)++;
          (*count)++;
          break;
      }
  }
  (*count) += i-1;
  (*count)++;
  if(c == 0 && n>1)
  {
      (*count)++;
      return 1;
  }
  (*count)++;
  return 0;
}
int main()
{
  int n, i, check1, check2, check3, count1=0, count2=0, count3=0;
  for(i=0; i<10; i++)
  {
      count1 = count2 = count3 = 0;
```

```
        printf("Enter a number\n");

        scanf("%d", &n);

        check1 = isPrime1(n, &count1);

        check2 = isPrime2(n, &count2);

        check3 = isPrime3(n, &count3);

        printf("Checks\t: %d\t%d\t%d\n", check1, check2, check3);

        printf("Steps\t: %d\t%d\t%d\n", count1, count2, count3);

    }

    return 0;

}
```

Conclusion:

Given query was implemented successfully.

2.2:

Program Title:

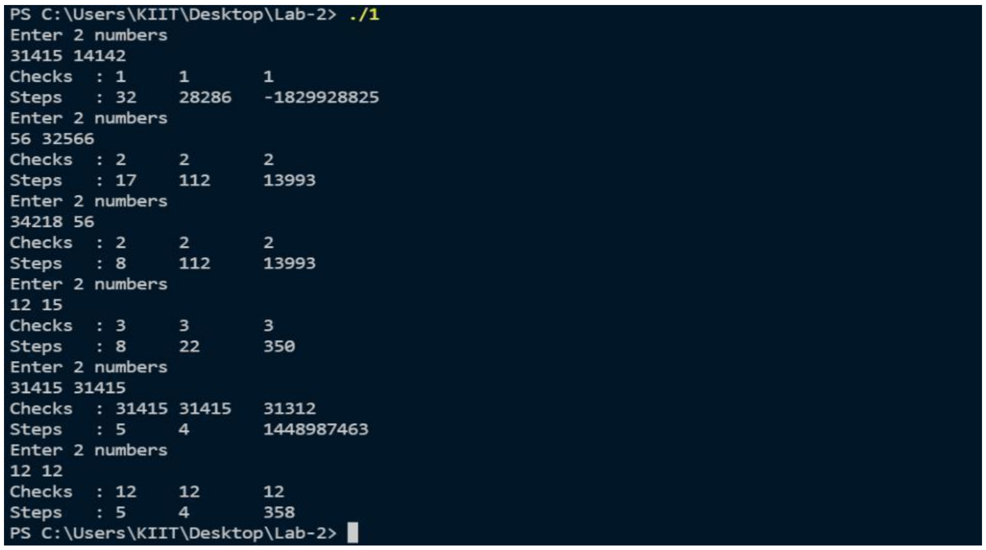
Write a program to find out GCD (greatest common divisor) using the following three algorithms.

a) Euclid's algorithm

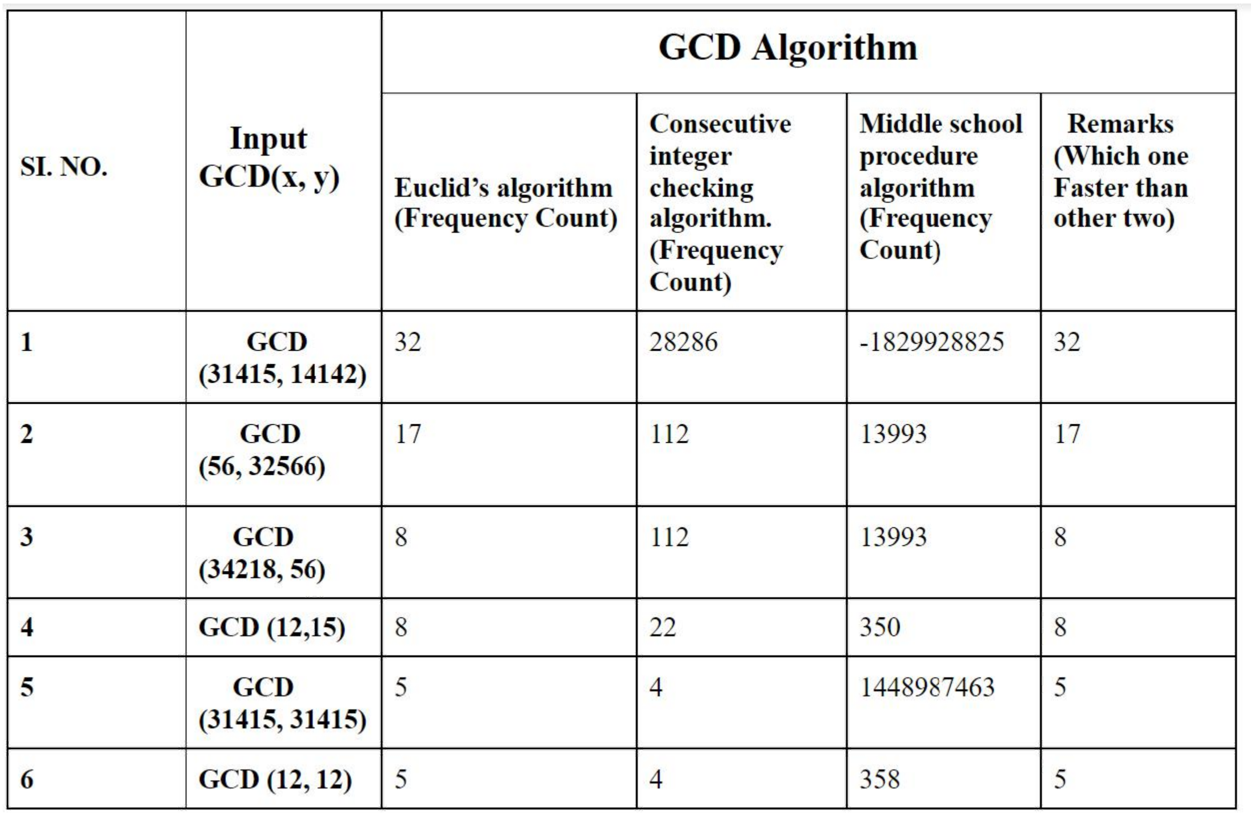
b) Consecutive integer checking algorithm.

c) Middle school procedure which makes use of common prime factors. For finding list of primes implement sieve of Eratosthenes algorithm.

Write a program to find out which algorithm is faster for the following data. Estimate how many times it will be faster than the other two by step/frequency count method in each case.
i. Find GCD of two numbers when both are very large i.e.GCD(31415, 14142) by applying each of the above algorithms.
ii. Find GCD of two numbers when one of them can be very large i.e.GCD(56, 32566) or GCD(34218, 56) by applying each of the above algorithms.
iii. Find GCD of two numbers when both are very small i.e.GCD(12,15) by applying each of the above algorithms.
iv. Find GCD of two numbers when both are same i.e.GCD(31415, 31415) or GCD(12, 12) by applying each of the above algorithms.
Write the above data in the following format and decide which algorithm is faster for the particular data.
Output:



```
PS C:\Users\KIIT\Desktop\Lab-2> ./1
Enter 2 numbers
31415 14142
Checks   : 1     1       1
Steps    : 32    28286   -1829928825
Enter 2 numbers
56 32566
Checks   : 2     2       2
Steps    : 17    112     13993
Enter 2 numbers
34218 56
Checks   : 2     2       2
Steps    : 8     112     13993
Enter 2 numbers
12 15
Checks   : 3     3       3
Steps    : 8     22      350
Enter 2 numbers
31415 31415
Checks   : 31415 31415   31312
Steps    : 5     4       1448987463
Enter 2 numbers
12 12
Checks   : 12    12      12
Steps    : 5     4       358
PS C:\Users\KIIT\Desktop\Lab-2>
```



SI. NO.	Input GCD(x, y)	GCD Algorithm			
		Euclid's algorithm (Frequency Count)	Consecutive integer checking algorithm. (Frequency Count)	Middle school procedure algorithm (Frequency Count)	Remarks (Which one Faster than other two)
1	GCD (31415, 14142)	32	28286	-1829928825	32
2	GCD (56, 32566)	17	112	13993	17
3	GCD (34218, 56)	8	112	13993	8
4	GCD (12,15)	8	22	350	8
5	GCD (31415, 31415)	5	4	1448987463	5
6	GCD (12, 12)	5	4	358	5

Source Code:

```
#include<stdio.h>

#include<math.h>

int gcd1(int a, int b, int *count) //Euclid's
Algorithm {
  while(a * b != 0)

  {

    (*count)++;

    if(a > b)

    {

      (*count)++;

      a = a % b;

    }
```

```
else
{
    (*count)++;
    b = b % a;
}
```

```
        (*count)++;

    }

    (*count)++;

    if(a == 0)

    {

        (*count)++;

        return b;

    }

    (*count)++;

    return a;

}

int gcd2(int a, int b, int *count) //Consecutive Integer Checking
Algorithm {
    int t;

    (*count)++;

    t = (a<b) ? a : b;

    (*count)++;

    while((a%t != 0) || (b%t != 0))

    {

        (*count)++; //for every condition check which returns true

        t--;

        (*count)++;

    }

    (*count)++; //for the last condition check

    (*count)++;

    return t;

}

int gcd3(int a, int b, int *count) //Middle School Method using sieve of Eratosthenes to find

primes
```

```
{
    int min, i, j, temp_a, temp_b, gcd=1;

    (*count)++;

    //Sieve of Eratosthenes

    min = (a<b) ? a : b; //As GCD will always be less than or equal to minimum of the two
    values (*count)++;
    int primes[min+1];

    (*count)++;

    primes[0] = primes[1] = 0; //As 0 and 1 are not prime
    (*count)++;
```

```
for(i=2; i<=min; i++) //sets all numbers from 2 onwards as prime unless specified otherwise
{
    primes[i] = 1;

    (*count)++;

}

(*count) += i-1;

for(i=2; i<min; i++) //sieves out the prime numbers
{
    if(primes[i])

    {

        for(j=i*i; j<=min; j += i)

        {

            primes[j] = 0;

            (*count)++;

        }

        (*count) += j;

    }

    (*count)++; //if statement condition checking

}

(*count) += i-1;

//Middle school method

//We can take advantage of the primes[] array itself to store the powers of the prime factors

for(i=2; i<=min; i++)

{

    if(primes[i])

    {

        temp_a = a;

        (*count)++;

        temp_b = b;
```

```
            (*count)++;

            primes[i] = 0; //Since we are counting powers now, we can ignore any prime that doesn't
divide a or b
            (*count)++;

            while( (temp_a%i == 0) && (temp_b%i == 0) ) //Keep counting while both are divisible
by the next power
            {

                (*count)++; //for every condition check which returns true

                primes[i]++;

                (*count)++;

                temp_a = temp_a/i; //Divide the number by the prime as much as possible, and count

powers
```

```
                (*count)++;

                temp_b = temp_b/i;

                (*count)++;

            }

            (*count)++; //For the last check in the while loop

        }

        (*count)++; //if statement condition checking

    }

    (*count) += i-1;

    for(i=2; i<=min; i++)

    {

        gcd = gcd * pow(i, primes[i]);

        (*count)++;

    }

    (*count) += i-1;

    //Alternatively, we could have multiplied and calculated the gcd in the previous loop itself, //but
    since the method used in middle school involved calculating it at the end, that's what has
been done.

    (*count)++;

    return gcd;

}

int main()

{

    int a, b, i, check1, check2, check3, count1=0, count2=0, count3=0;

    for(i=0; i<6; i++)

    {

        count1 = count2 = count3 = 0;

        printf("Enter 2 numbers\n");
        scanf("%d%d", &a, &b);
        check1 = gcd1(a, b, &count1);
        check2 = gcd2(a, b, &count2);
        check3 = gcd3(a, b, &count3);
        printf("Checks\t: %d\t%d\t%d\n", check1, check2, check3);
        printf("Steps\t: %d\t%d\t%d\n", count1, count2, count3);
    }
     return 0;}
```

Conclusion:

Given query was implemented successfully.

<u>2.3:</u>

<u>Program Title:</u>

Write a menu driven program as given below, to sort an array of n integers in ascending order by insertion sort algorithm and determine the time required (in terms of step/frequency count) to sort the elements. Repeat the experiment for different values of n and different nature of data (i.e.apply insertion sort algorithm on the data of array that are already sorted, reversely sorted and random data). Finally plot a graph of the time taken versus n for each type of data. The elements can be read from a file or can be generated using the random number generator. INSERTION SORT MENU

0. Quit

1. n Random numbers=>Array

2. Display the Array

3. Sort the Array in Ascending Order by using Insertion Sort Algorithm
4. Sort the Array in Descending Order by using any sorting algorithm

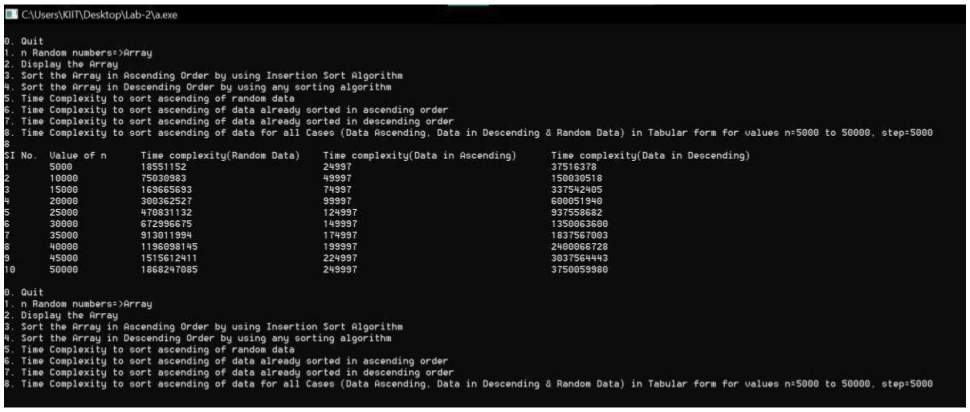
5. Time Complexity to sort ascending of random data

6. Time Complexity to sort ascending of data already sorted in ascending order
7. Time Complexity to sort ascending of data already sorted in descending order

8. Time Complexity to sort ascending of data for all Cases (Data Ascending, Data in Descending & Random Data) in Tabular form for values n=5000 to 50000, step=5000
Enter your choice:

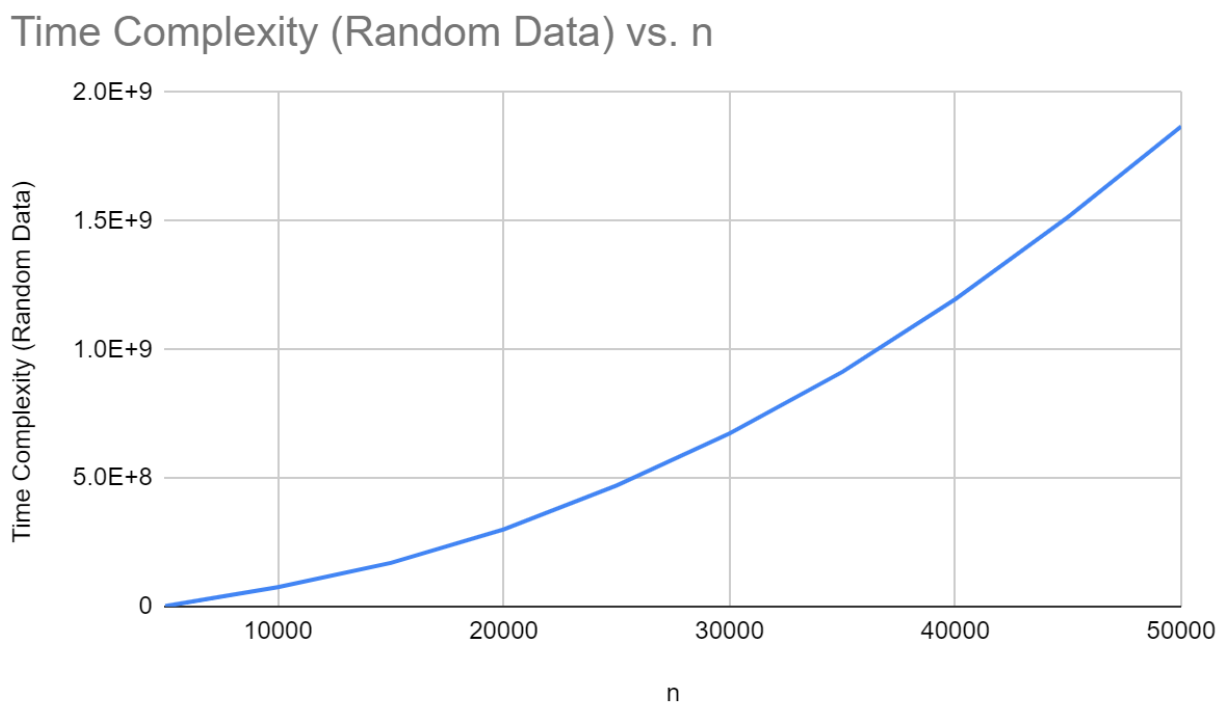
If the choice is option 8, the it will display the tabular

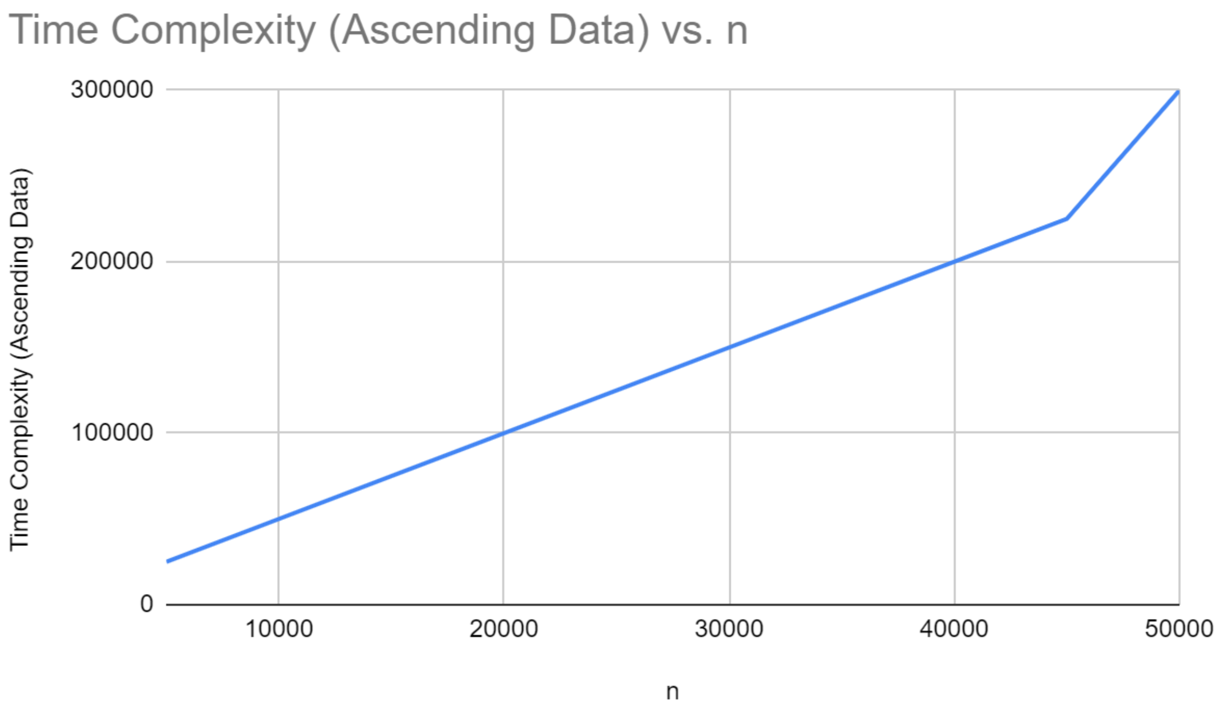
<u>Output:</u>

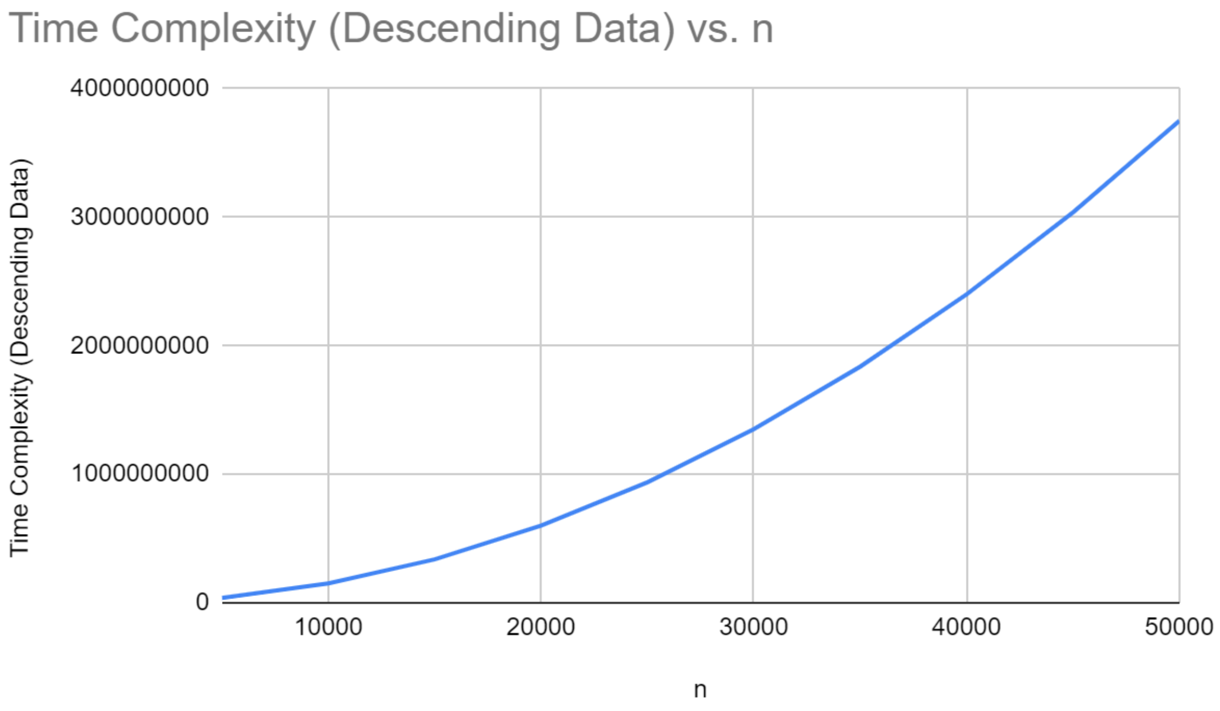


```
C:\Users\KIIT\Desktop\Lab-2\a.exe

0. Quit
1. n Random numbers=>Array
2. Display the Array
3. Sort the Array in Ascending Order by using Insertion Sort Algorithm
4. Sort the Array in Descending Order by using any sorting algorithm
5. Time Complexity to sort ascending of random data
6. Time Complexity to sort ascending of data already sorted in ascending order
7. Time Complexity to sort ascending of data already sorted in descending order
8. Time Complexity to sort ascending of data for all Cases (Data Ascending, Data in Descending & Random Data) in Tabular form for values n=5000 to 50000, step=5000
8
SI No.  Value of n      Time complexity(Random Data)    Time complexity(Data in Ascending)      Time complexity(Data in Descending)
1       5000            18551152                        24997                                   37516378
2       10000           75030983                        49997                                   150030518
3       15000           169665693                       74997                                   337542405
4       20000           300362527                       99997                                   600051940
5       25000           470831132                       124997                                  937558682
6       30000           672996675                       149997                                  1350063600
7       35000           913011994                       174997                                  1837567003
8       40000           1196098145                      199997                                  2400066728
9       45000           1515612411                      224997                                  3037564443
10      50000           1868247085                      249997                                  3750059980

0. Quit
1. n Random numbers=>Array
2. Display the Array
3. Sort the Array in Ascending Order by using Insertion Sort Algorithm
4. Sort the Array in Descending Order by using any sorting algorithm
5. Time Complexity to sort ascending of random data
6. Time Complexity to sort ascending of data already sorted in ascending order
7. Time Complexity to sort ascending of data already sorted in descending order
8. Time Complexity to sort ascending of data for all Cases (Data Ascending, Data in Descending & Random Data) in Tabular form for values n=5000 to 50000, step=5000
```

S.NO.	VALUE OF (n)	Time Complexity (Random Data)	Time Complexity (Ascending Data)	Time Complexity (Descending Data)
1	5000	18551152	24997	37516378
2	10000	75030983	49997	150030518
3	15000	169665693	74997	337542405
4	20000	300362527	99997	600051940
5	25000	470831132	124997	937558682
6	30000	672996675	149997	1350063600
7	35000	913011994	174997	1837567003
8	40000	1196098145	199997	2400066728
9	45000	1515612411	224997	3037564443
10	50000	1868247085	249997	3750059980







Source Code:

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

```
#include<stdlib.h>

#include<math.h>

void destroy_prev_allocation(int *a, int n) // Destroys previously (dynamically allocated) memory

from the array

{

    int i;

    for(i=0; i<n; i++)

    {

        free(a + i);

    }

}

void insert_rand_array(int *a, int n) // Inserts n random values into the
array {
    int i;

    for(i=0; i<n; i++)

    {

        a[i] = rand();
    }

}

void display(int *a, int n) // Displays array
elements {
    int i;

    for(i=0; i<n; i++)

    {

        printf("%d\t",a[i]);

    }

    printf("\n");

}

void insertion_sort_ascending(int *a, int n, long long int *count) // Sorts the array in ascending

order using insertion sort

{

    int i, j, temp;

    (*count)++;

    for(i=1; i<n; i++)

    {

        j = i;

        (*count)++;

        temp = a[i];

        (*count)++;

        while( (temp < a[j-1]) && (j > 0) )

        {

            (*count)++; // for true checks of while loop

            a[j] = a[j-1];

            (*count)++;

            j--;
            (*count)++;
        }
        (*count)++; // for last check of while loop
        a[j] = temp;
        (*count)++;
    }
    (*count) += i;
}
void selection_sort_descending(int *a, int n) // Sorts the array in descending order using
selection sort
{
    int i, j, max, temp;
    for(i=0; i<n; i++)
    {
        max = i;
        for(j=i+1; j<n; j++)
        {
            if(a[j] > a[max])
            {
                max = j;
            }
        }
        temp = a[i];
        a[i] = a[max];
        a[max] = temp;
    }
}
void row_display(int *a, int n)
{
    long long int count1=0, count2=0, count3=0;
    insert_rand_array(a, n);
    // Random
    insertion_sort_ascending(a, n, &count1);
    // Ascending
    insertion_sort_ascending(a, n, &count2); //Array is already sorted in ascending order
    // Descending
    selection_sort_descending(a, n);
    insertion_sort_ascending(a, n, &count3);
    printf("%d\t\t%lld\t\t\t%lld\t\t\t\t\t%lld\n", n, count1, count2, count3);
}
int main()
```

```
{
    int choice=1, *a, n=0;
    long long int count=0;
    while(choice)
    {
        printf("\n0. Quit"
        "\n1. n Random numbers=>Array"
        "\n2. Display the Array"
        "\n3. Sort the Array in Ascending Order by using Insertion Sort Algorithm"

        "\n4. Sort the Array in Descending Order by using any sorting algorithm"

        "\n5. Time Complexity to sort ascending of random data"

        "\n6. Time Complexity to sort ascending of data already sorted in ascending order" "\n7.
        Time Complexity to sort ascending of data already sorted in descending order" "\n8. Time
        Complexity to sort ascending of data for all Cases (Data Ascending, Data in
    Descending & Random Data) in Tabular form for values n=5000 to 50000, step=5000\n"
        );

        scanf("%d",&choice);

        switch(choice)

        {

          case 0:

          printf("Exiting\n");

          break;

          case 1:

          destroy_prev_allocation(a, n);

          printf("Enter n\n");

          scanf("%d", &n);

          a = (int*) malloc( sizeof(int) * n );

          insert_rand_array(a, n);

          break;

          case 2:

          display(a, n);

          break;

          case 3:

          insertion_sort_ascending(a, n, &count);

          break;

          case 4:

          selection_sort_descending(a, n);
```

```
break;

case 5:

count = 0;

insertion_sort_ascending(a, n, &count);

printf("Step-count : %lld\n", count);

break;

case 6:
        insertion_sort_ascending(a, n, &count);
        count = 0;
        insertion_sort_ascending(a, n, &count); // The array is already sorted in ascending order
        printf("Step-count : %lld\n", count);
        break;
        case 7:
        selection_sort_descending(a, n);
        count = 0;
        insertion_sort_ascending(a, n, &count); // The array is already sorted in descending
order
        printf("Step-count : %lld\n", count);
        break;
        case 8:
        printf("Sl No."
        "\tValue of n"
        "\tTime complexity(Random Data)"
        "\tTime complexity(Data in Ascending)"
        "\tTime complexity(Data in Descending)\n"
        );
        for(n=5000; n<=50000; n += 5000)
        {
            a = (int*) malloc( sizeof(int) * n );
            printf("%d\t", (n/5000));
            row_display(a, n);
            destroy_prev_allocation(a, n);
        }
        break;
```

```
            default:

            printf("\t\tINVALID CHOICE\n");

        }

    }

    return 0;

}
```

<u>Conclusion:</u>

Given query was implemented successfully.
