Q. 1 Write a program to generate the n unique random numbers by two different algorithms.
Ist Using Fisher Yates Algorithm:

Algorithm: UniqueRandom(n)

Input: Count of number to output.
Output: List of N random unique number.
Assumption: K > N

```
INITIALIZE MAX = K
SET A = List of Number from 0 to MAX
IF N<=MAX THEN
    FOR i=0 to N DO
        index = Random(0,MAX-1)
        print A_index
        Swap(A_index , A_(MAX -1))
        MAX = MAX – 1
    END LOOP
END IF
END
```

Note:
Random() : Generates Psudo Random Number in between given Range.
Swap(): Swap given Elements value

Algorithm Analysis:

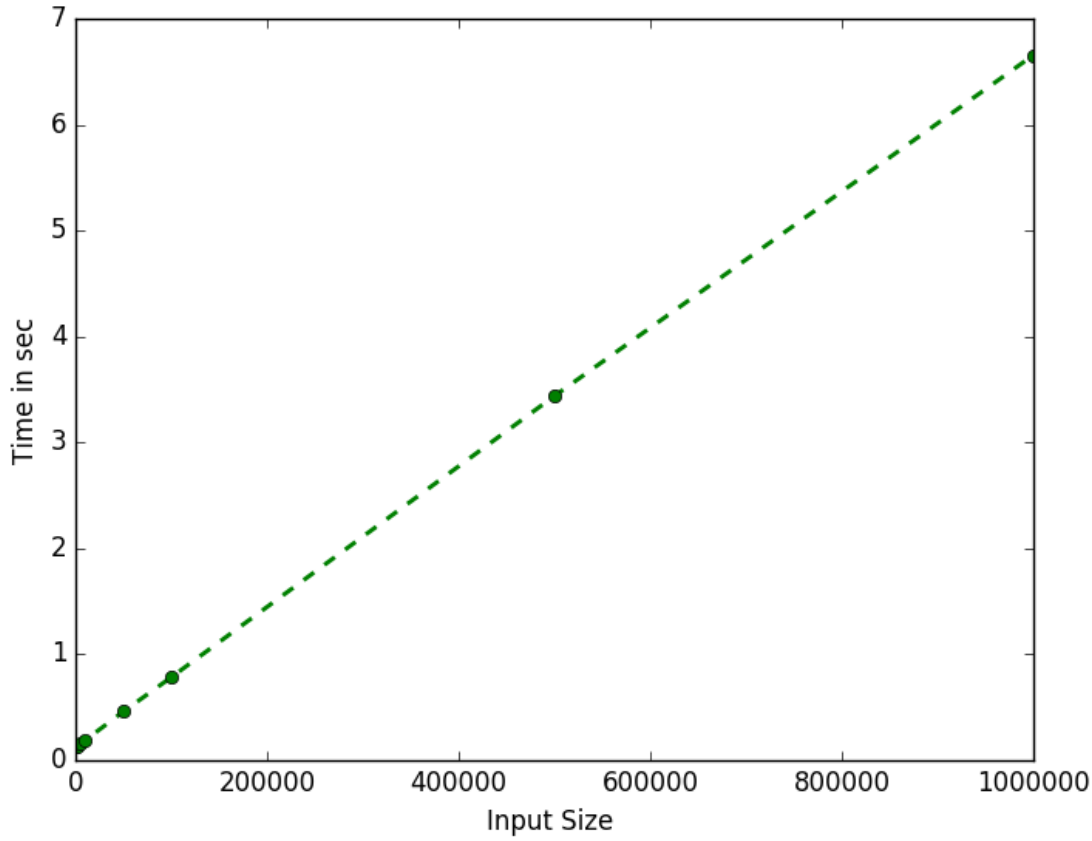
$T(n) = KxC_1 + nxC_2 + C_3$
Where,
K = Value of max Random Number (Constant)
n = Output number list count
hence,
T(n) = O(n) for K>=n

Input/Output:

K=1000000 , Input is n = number of element to print
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output	Time (in Sec)
1	1	0.129
10	10	0.128
100	100	0.124
1000	1000	0.131
5000	5000	0.156
10000	10000	0.188
50000	50000	0.460
100000	**100000**	**0.784**
500000	**500000**	**3.439**
1000000	**1000000**	**6.656**

Complexity Graph:



Conclusion:

As it can be seen from graph initially time only depends upon k(MAX) value but as the n increases it linearly depends on n.

2nd Using Linear Congruential:

Algorithm: UniqueRandom(n)

Input: Count of number to output.
Output: List of N random unique number.
Assumption: L >= N , and

1. M ,C are coprime
2. L-1 divisible by all factors of M
3. L-1 is divisible by 4 if M is divisible by 4

Set X = 1, L = K_1 , C = K_2 , M = K_3
// where K_1 , K_2 and K_3 are Constant as per Assumption.
FOR i = 0 to N DO
X = X ((L*X)+C) mod M
print X
END LOOP
END

Algorithm Analysis:

$T(n) = nxC_1 + C_3$
Where, n = Output number list count

Simplifying ..

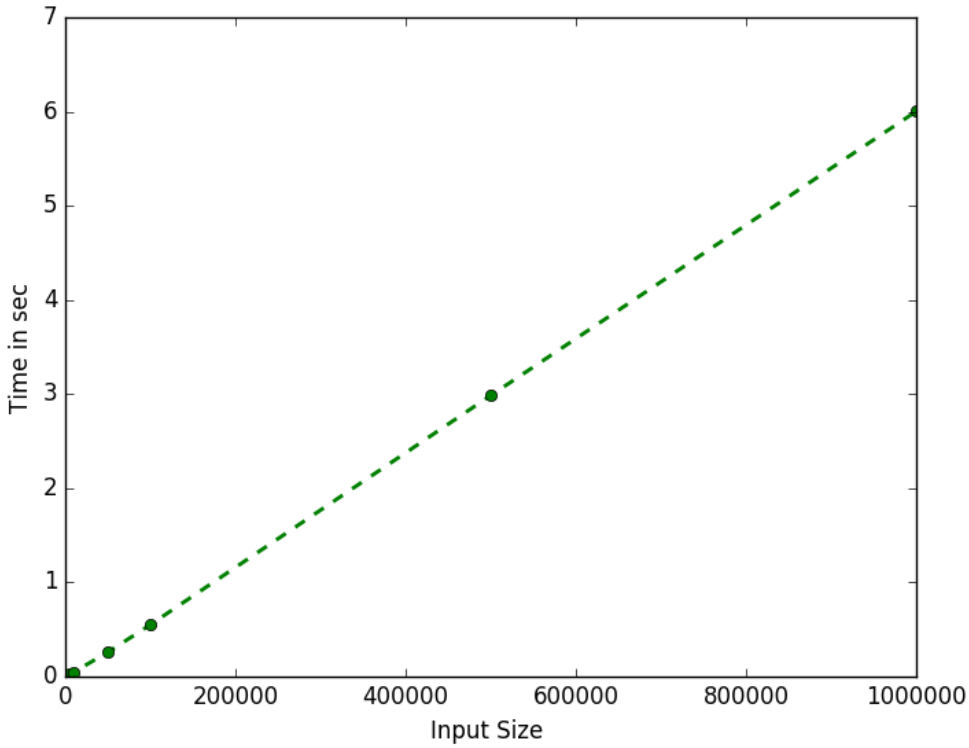
T(n) = O(n) for n<=L

Input/Output:

Input is n = number of element to print
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output	Time (in Sec)
1	1	0.0
10	10	0.0
100	100	0.001
1000	1000	0.007
5000	5000	0.034
10000	10000	0.048
50000	50000	0.258
100000	**100000**	**0.551**
500000	**500000**	**2.985**
1000000	**1000000**	**6.002**

Complexity Graph:



Conclusion:

In comparision to last graph it can be seen initially to end it only depend on value of n, and hence it linearly grow w.r.t n.

Q. 2 There is an array A of integers of size n which cannot directly be accessed. However, you can get true or false response to queries of the form A[i] < A[j]. It is given that A has only one duplicate pair, and rest all the elements are distinct. So it has n-1 distinct elements and 1 element which is same as one of the n-1 elements. Your task is to identify the indices of the two identical elements in A.

Algorithm: DuplicatePair(A)

Input: A as List of N-1 unique number with one duplicate number
Output: Index of Duplicate Pair

```
SET flag = 0
FOR i=0 to N DO
    IF flag != 1 THEN
        FOR j=i+1 to N DO
            IF !(Compare(Ai , Aj) || Compare(Aj , Ai)) THEN
                Print i and j as index
                set flag = 1
                BREAK LOOP
    ELSE
        BREAK LOOP
    END IF
END LOOP
END
```

Note:
Compare(A_i , A_j) : This Function return Boolean value of **$A_i > A_j$**

Algorithm Analysis:

T(n) = C1 + C2x(n+1) + C3x[(1x(n-1)) + (2x(n-2)) + (3x(n-3)) + ...+((n-1)x(1))]

reducing constants and writing for all iterations .....for worst case

Where,
n = number of total input

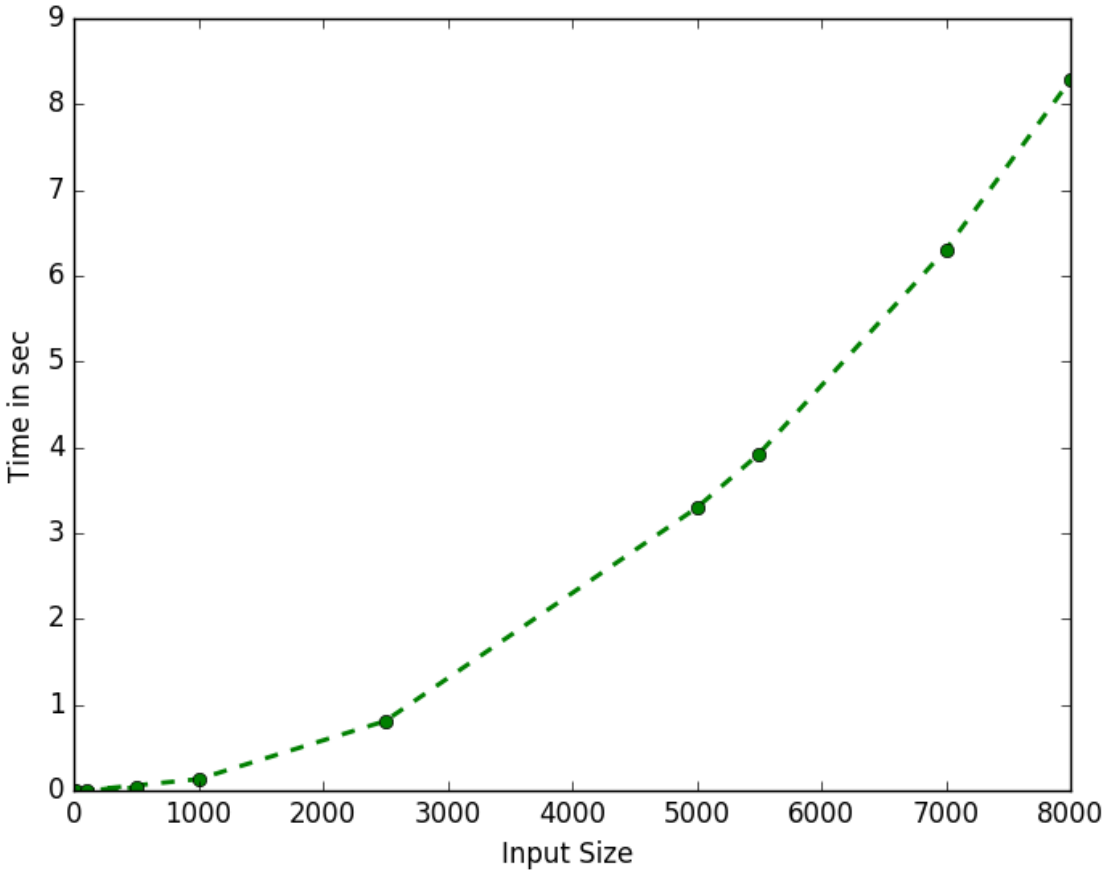
Simplifying ..

T(n) = $O(n^2)$ for worst case
= O(1) for best case

Input/Output: (for worst case i.e. no duplicate pair or in the last of list)
Input is n = Number of element as in Input List
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Number)	Time (in Sec)
1	1	0.0
10	1	0.0
500	1	0.033
100	1	0.002
1000	1	0.132
2500	1	0.81
5000	1	3.295
5500	**1**	**3.924**
7000	**1**	**6.306**
8000	**1**	**8.281**

Complexity Graph:



Conclusion:

The curve now grows non-linearly wrt to input value, this is because of polynomial nature of time complexity.

Q. 3 Write a function customSort takes a "num" array of integers as an input along with the array of "weights" which contains the weights of the corresponding integers. We wish to sort this vector "num" from the indices start to end based on the fact that numbers with higher weights appear first and in case the weights are equal, we put the greater number first in the list. We also wish efficient algorithms. We expect an O (n*log(n)) algorithm here instead of a O(n^2) one.

Algorithm: CustomSort(A)

Input: A, B as 2 List of N values of number and their weights.
Output: Sorted array.

CUSTOMSORT(A):

```
CREATE List C = (B U A)          // Creating 2D List
SORT(C)                          // USE Merge Sort as Basic Algo
```

SORT(C):

SAME As MergeSort With Compare using 1D element

```
IF C_L[0] < C_R[0] THEN          // [0] index for weight
    SWAP()                       // Swap Numbers
ELSE IF C_L[0] == C_R[0] THEN    // [0] index ie weight are equal
    IF C_L[1] > C_R[1] THEN      // SWAP Using Numbers
        SWAP()
    END IF
END IF
```

Note:
Using merge sort as basic sort technique to sort for nlogn complexity, passing a 2d array as parameter and comparing only first element of there weight and then if equal swap element on basis of bigger element of

Algorithm Analysis:

$T(n) = \Theta(n*\log(n)) + C*n$

Where,
n = Size of total input List

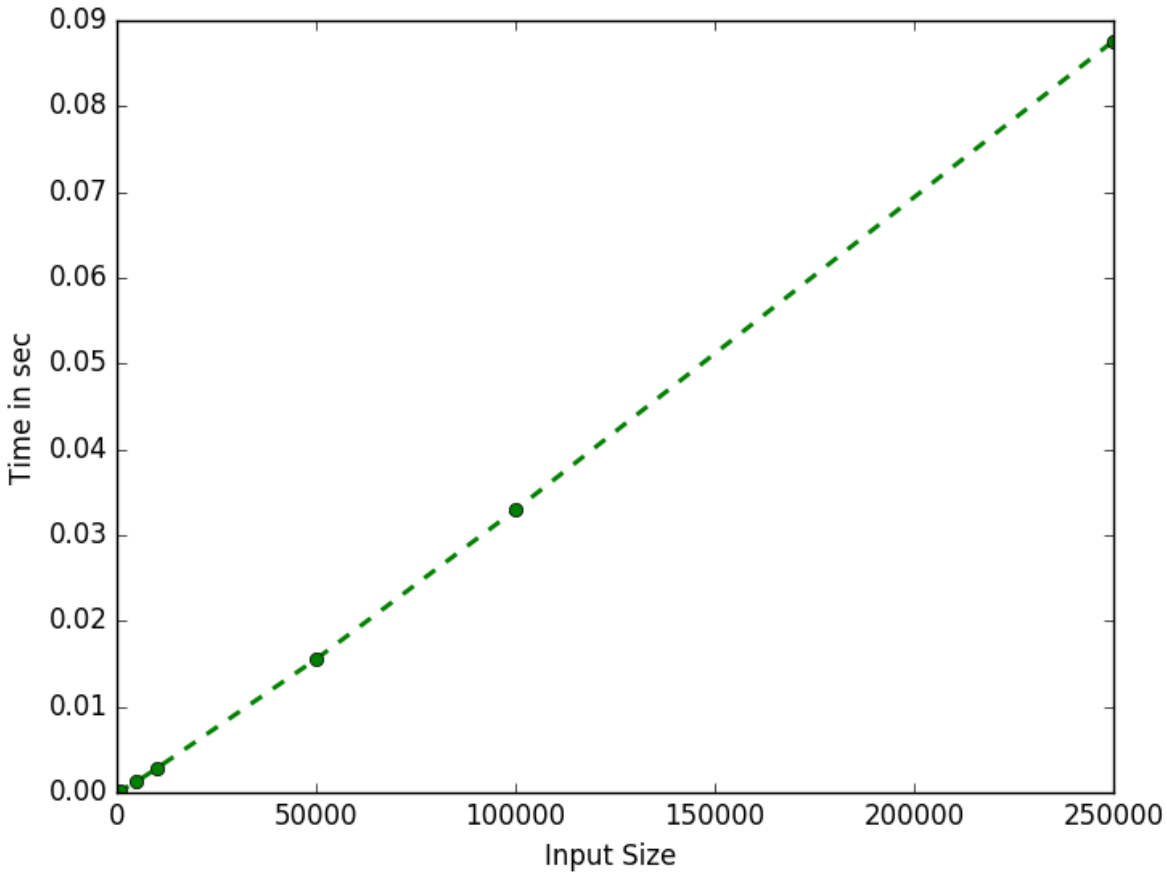
Hence,

$T(n) = O(n*\log(n))$

Input/Output:
Input is n = Number of elements as in Input List.
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Number)	Time (in Sec)
1	1	0.0
10	10	0.000004
100	100	0.000027
1000	1000	0.000223
5000	5000	0.001283
10000	10000	0.002801
50000	50000	0.015544
100000	**100000**	**0.032946**
250000	**250000**	**0.087610**

Complexity Graph:



Conclusion:
The Curve looks similar to that of linear curve, due to its n*logn nature, and since value of logn is small for initial value it looks simialr to linear curve.

Q. 4 Write a program to find the value of x^n, where 'x' and 'n' are the real numbers using 3 different algorithms.

Algorithm: PowerXandY(A)

Input: A, B for A^B
Output: Output Result as A^B

K is the number of terms till taylors series is to be calculated

1st : Using Taylor Series

```
SET   MAX to K
INITIALIZE RESULT,A to 0
INITIALIZE FACT to 1
FOR n=0 to n=MAX DO
        SET A = b*LOG(a) // LOG function calculate LOG of given number
        SET A = POW(A,n) // POW function iterate i times to find A^i
        IF i!=0 THEN DO
          SET FACT = FACT x n
        END IF
        SET A = A/FACT
        STORE  RESULT = RESULT + A
 END LOOP
Return RESULT
END
```

Note:
LOG() function require constant time to calculate since it uses system hardware and preinbuilt table to calculate Log of any value.
POW() is custom function calculate Power in integer value using simple iteration

Algorithm Analysis:

$$\mathbf{T(n) = C_1 + C_2x(K+1) + C_3x(1+2+3+....+K)}$$
$$\mathbf{= C_1 + C_2x(K+1) + C_3x(K^2-K)/2}$$

Where,
K = Fixed value of iteration to calculate Taylor series

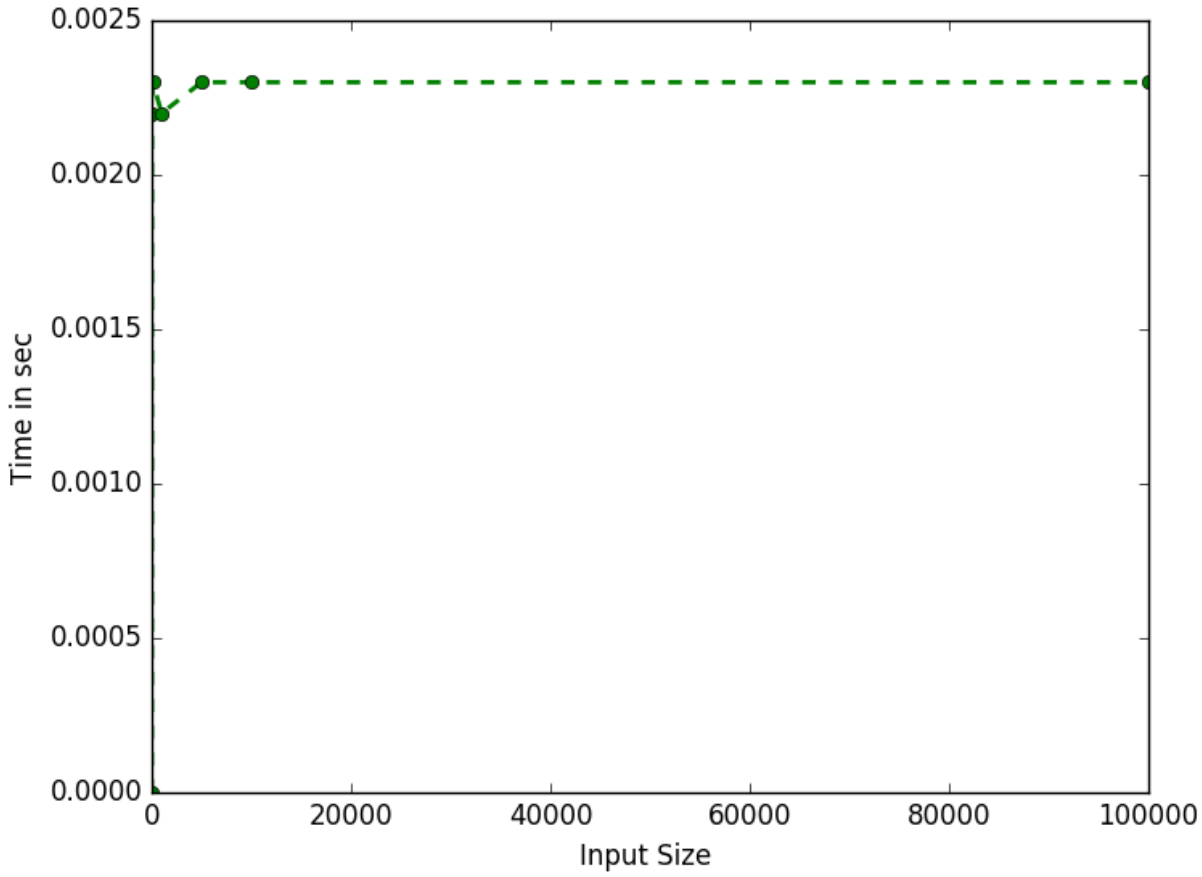
hence its time complexity ..

$$\mathbf{T(n) = O(K^2)}$$ **where K is constant set initially**

Input/Output:
For K=150, A=2 fixed and Varying B
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input (for A=2)	Output (Number)	Time (in Sec)
1	2.0	0.0022
10	1024.0	0.0023
100	1.26765060023e+30	0.0023
1000	-	0.0022
10000	-	0.0023
100000	-	0.0023
2.5	5.65685424949	0.0023
0.5	1.41421356237	0.0022

Complexity Graph:



Conclusion:
The curve as expected not depend upon B or A it only depends upon K and hence the curve is linearly constant.

Q. 5 Write a program to find the factorial of the given number (0 < n < 10,000,000,000).

Algorithm: FACTORIAL(N)

Input: N
Output: Factorial of N

FACTORIALI(N):

```
    INITIALIZE Output = Ø  // empty list
    ADD 1 to Output List
    FOR n=0 to n=N+1 DO
        SET Output = MULTIPLY(OUTPUT, n)
    END LOOP
    RETURN Output
END FACTORIAL
```

MULTIPLYE(A,N):

```
    Set Carry,X = 0
    FOR n=0 to n = | A |
        SET X = A_i * n + Carry
        A_i = X%10
        Carry = x/10
        WHILE carry%10 != 0 DO
            ADD (Carry%10) to the A List
            SET Carry = Carry/10
        END LOOP
    END LOOP
    RETURN A
```

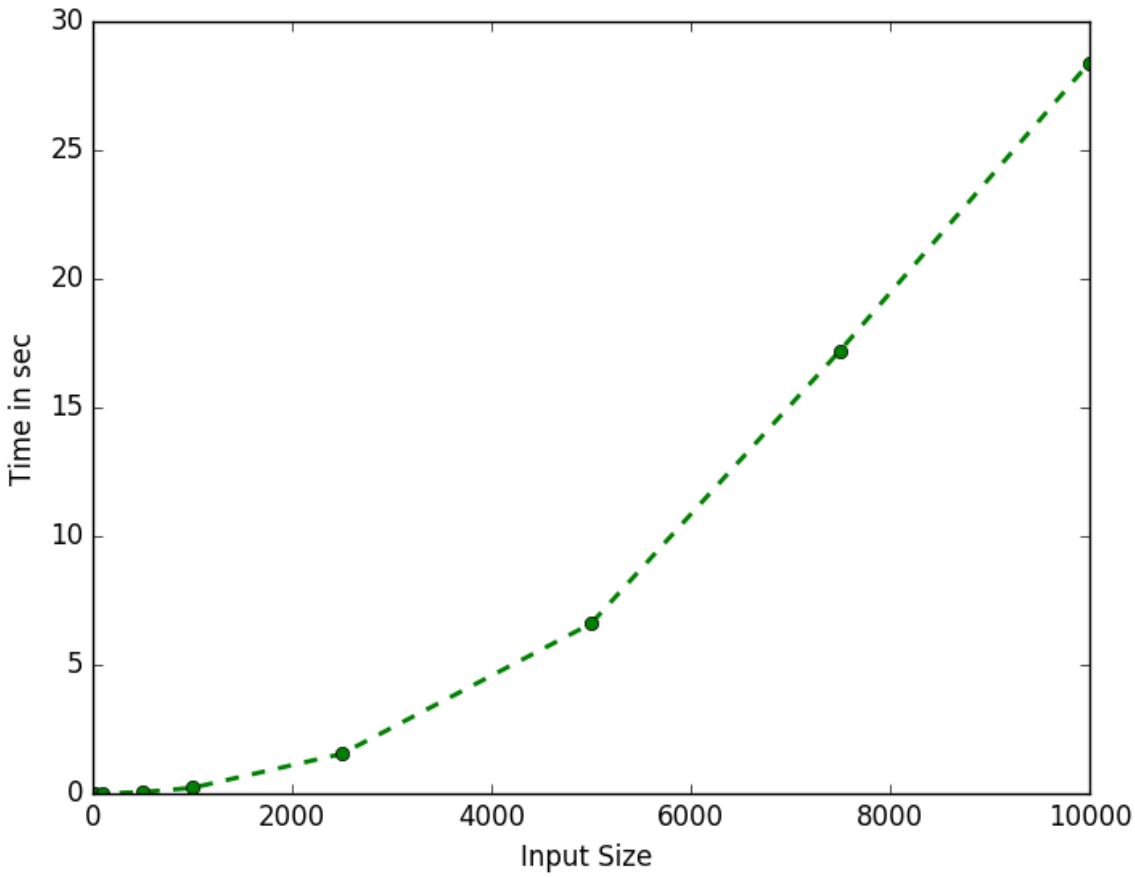
Algorithm Analysis:

T(n) =

Input/Output:
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Number)	Time (in Sec)
1	1	0.0
10	Fact(10)	0.0
100	Fact(100)	0.004
500	Fact(500)	0.053
1000	Fact(1000)	0.22
2500	Fact(2500)	1.543
5000	Fact(5000)	6.601
7500	**Fact(7500)**	**17.217**
10000	**Fact(10000)**	**28.401**

Complexity Graph:



Conclusion:
The curve now grows non-linearly wrt to input value, this is because time depends on number initially and then it depends on number of digits as output.

Q. 6 Write a program to find the nth Fibonacci number with two different approach and compare their time complexity.

Algorithm: Fibonacci(A)

Input: N
Output: Nth Fibonacci term
Assumption: Fibonacci(0)=1 and Fibonacci(1)=1

1. Method: Recursive

```
IF n < 2 THEN
        RETURN n
ELSE
        RETURN Fibonacci(n-1) + Fibonacci(n-2)
END IF
END
```

2. Method: Iterative

```
SET    nextVal = 1, old = 0, temp = 0
    FOR i =0 to N DO
        IF i<2 THEN
                SET old=1
                SET nextVal=1
        ELSE
                 SET temp = old
                 SET old = nextVal
                 SET  nextVal = temp + nextVal
        END IF
   END LOOP
   RETURN nextVal
END
```

Algorithm Analysis:

$T(n) = C_1 x T(n-1) + C_2 x T(n-2) + C_3$
..For Recursive
$T(n) = C_1 + C_2 x (n+1) + n x (C_3)$
..For Iterative
Where,
n = nth number to find fibonacci term

Simplifying ..

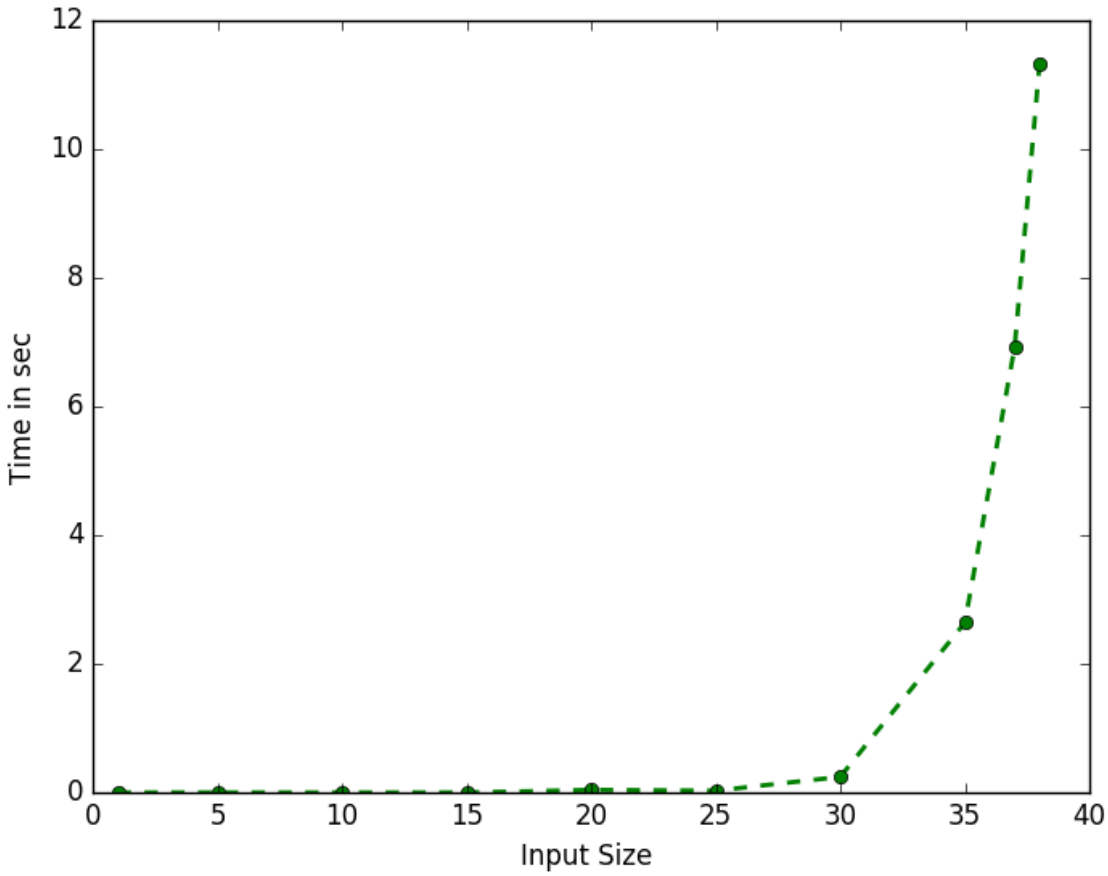
$T(n) = O(2^n)$ for Recursive

$T(n) = O(n)$ for Iterative

Input/Output: (Recursive)
input is n = nth number to find fibonacci term
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Number)	Time (in Sec)
1	1	0.0
5	5	0.0
10	55	0.0
15	610	0.0
20	6765	0.04
25	75025	0.028
30	832040	0.243
35	**9227465**	**2.638**
37	**24157817**	**6.922**
38	**39088169**	**11.327**

Complexity Graph: (Recursive)

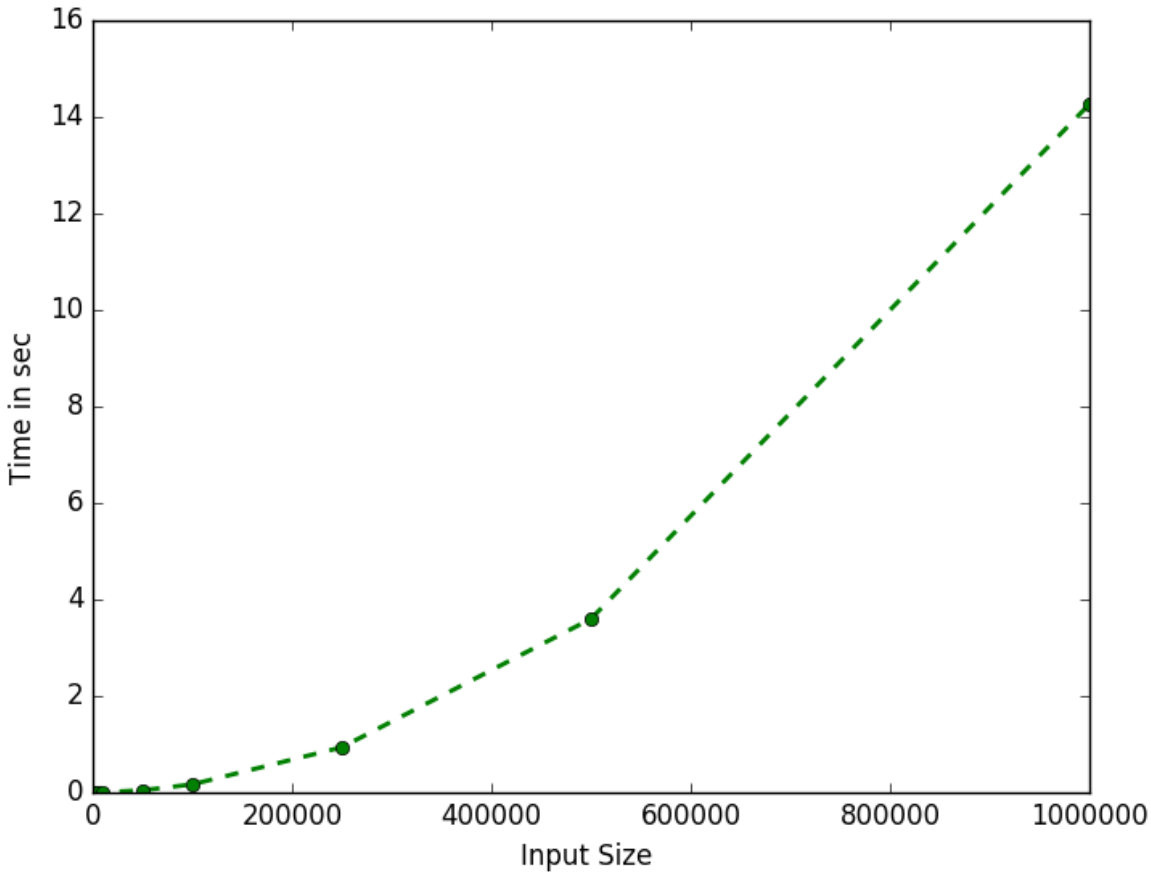


Conclusion:
The curve as expected grows exponetially as the input size increases.

Input/Output: (Iterative)
input is n = nth number to find fibonacci term
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Number)	Time (in Sec)
1	1	0.0
10	55	0.0
100	Fibo(100)	0.0
1000	Fibo(1000)	0.0
5000	Fibo(5000)	0.003
10000	Fibo(10000)	0.007
50000	Fibo(50000)	0.051
100000	**Fibo(100000)**	**0.159**
250000	**Fibo(250000)**	**0.938**
1000000	**Fibo(1000000)**	**14.266**

Complexity Graph: (Iterative)



Conclusion:
The curve is expected to be grow linearly but it is not exactly linear.

Q. 7 An array of integers is said to be a straight-K, if it contains K elements that are K consecutive numbers. For example, the array {6, 1, 9, 5, 7, 15, 8} is a straight because it contains 5, 6, 7, 8, and 9 for K=5. Write a program to finds the maximum value of K for the given number of integers.

Algorithm: StraightK(A)

Input: A as list of N numbers
Output: K-number for Array as per defination

```
SORT(A)                         // SORT function to sort initial list
Set MAX_K, K, temp = 0
INITIALIZE F = Ø                // EMPTY LIST
FOR n=0 to n=N DO               // N = size of list
        SET temp = Ai - K
        IF (temp==1) && (Ai != 1) THEN
                MAX_K = MAX_K + 1
        ELSE
                ADD MAX_K+1 to F LIST
                SET  MAX_K =0
        END IF
        SET K =  Ai
END LOOP
PRINT MAX(F) // MAX of F list element as K number
END
```

Note:
Complexity of SORT function is NlogN.

Algorithm Analysis:

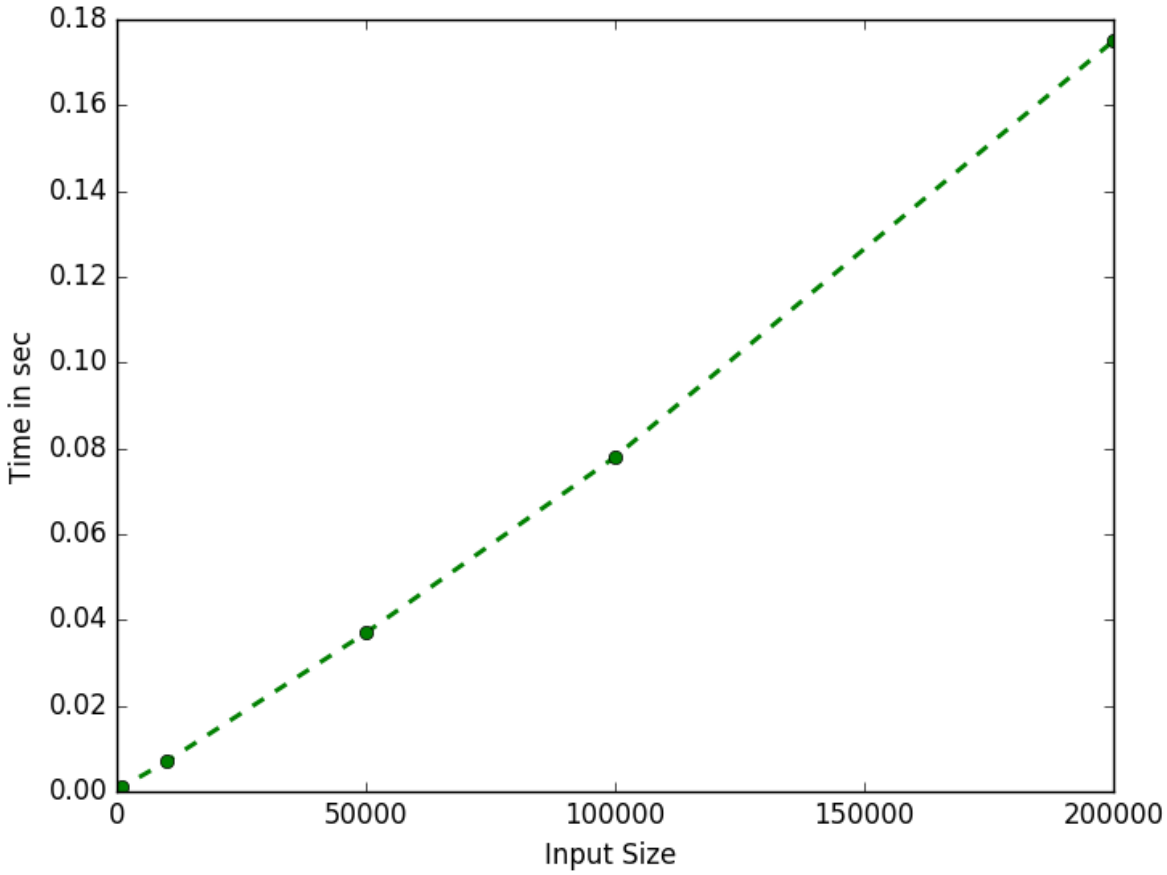
T(n) = Θ(nlog(n)) + C1 + C2*(n+1) + C3*n
Where,
n = Size of Input List
hence,
T(n) = O(n*Log(n))

Input/Output: (Recursive)

At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Number)	Time (in Sec)
1	1	0.0
10	1	0.0
100	1	0.0
1000	1	0.001
10000	1	0.007
50000	1	0.037
100000	1	0.078
200000	**1**	**0.175**

Complexity Graph: (Recursive)



Conclusion:

The Curve looks more linear but it varies at high input due to suffiecient value of LogN hence it verifies that curve grows with nLog(n) Complexity. And it Also verify that this curve only dominate to time due to sorting of initial List.

Q. 8 You have an array like :-{ 5,5,5,13,6,13,6,7,8,7,-1) Write and analyze an algorithm to arrange this array in sorted form based upon the number of occurrence e.g.: above should look like this after execution of your also {5,5,5,13,13,6,6,7,7,8,-1} 13 comes before 6 because it has same number of occurrence as 6 but it come first in the parent array.

Algorithm: FrequencCount(A)

Input: A as list of N numbers
Output: Sorted List as per frequency of occurence

```
SET MAX = 1000000
SET index, freq=0
INITIALIZE List A,B,C = Ø                    // Here B is 2D List
FOR i=0 to i=N DO
        SET t = Ai
        SET List C as  1D element of B
        IF t in C THEN                       // search t in C
                SET index = index of t in C
                Set frequency = frequency at B(index)+1
                Update frequency at B(index)
        ELSE
                ADD T with frequency=1 to B
        END IF
END LOOP
SORT(B,2)                                    // Sort B on basis of frequency
RETURN B
END
```

Note:
SORT(List,KeyIndex) function take List and sort on the basis of Key Index.

Algorithm Analysis:

$T(n) = C1 + C2*(n+1) + n*(C3*n + C4*n +C5) + \Theta(nLog(n))$

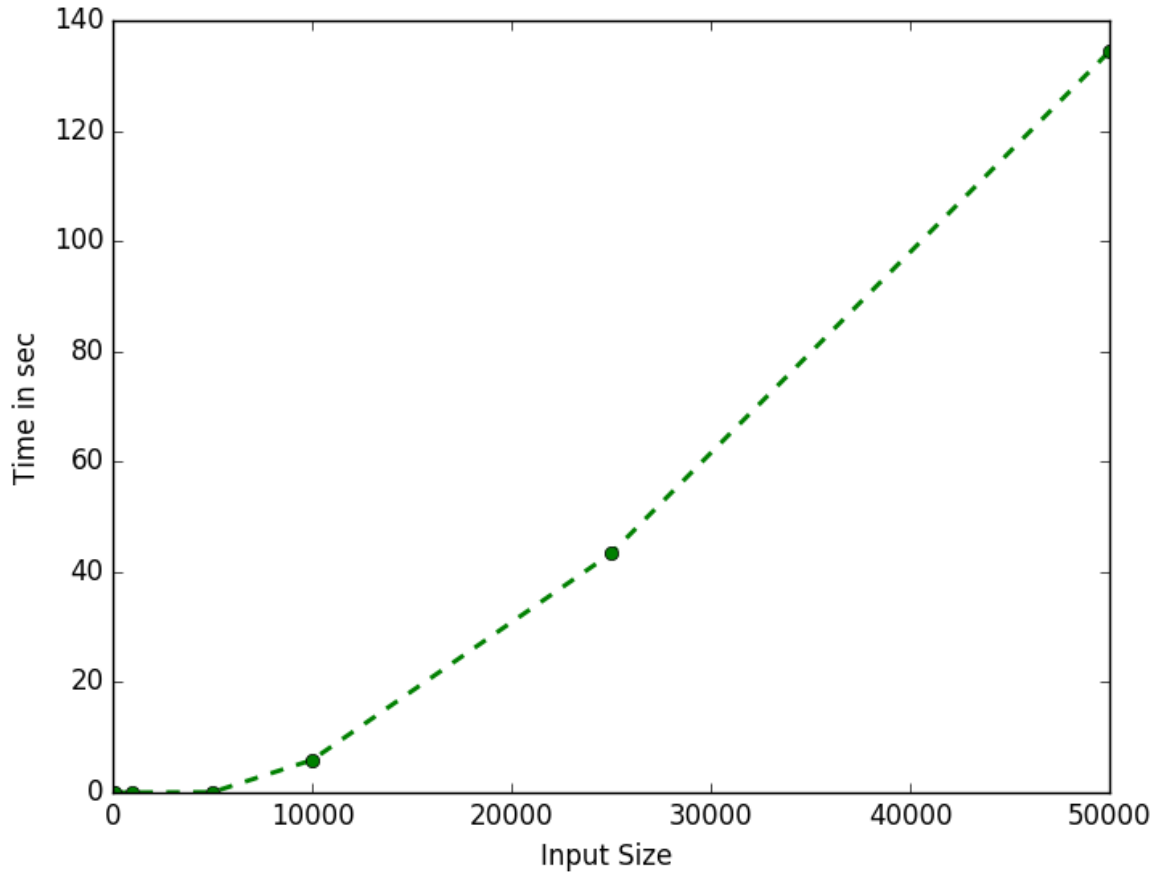
for worst case (i.e. every number with unique occurance)
Where,
n = Size of Input List

hence,
$T(n) = O(n^2)$ Since n^2 is the dominating term

Input/Output:
input is n as list of numbers
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Numbers)	Time (in Sec)
1	1	0.0
10	10	0.0
100	100	0.001
1000	1000	0.008
5000	5000	0.059
10000	10000	5.892
25000	25000	43.345
50000	**50000**	**134.389**

Complexity Graph:



Conclusion:
The curve now grows non-linearly wrt to input value, hence verify its polynomial nature growth wrt to input.

Q. 9 Find Determinant of Square matrix using recursion.

Algorithm: DETERMINANT(A,i,j)

Input: A as 2D Matrix
Output: Determinant of A.

DETERMINANT(A,L,M)

```
Set n = Size(A)                          // Size of A Matrix
IF n<=1 THEN
    RETURN A[0][0]
ELSE
   SET sum = 0
   INITIALIZE temp = [[[Ø]]]             // 3D temp To store Cofactor Size n-1
   INITIALIZE B = [Ø]
   FOR O=0 to O = n DO
          INITIALIZE B = [Ø]
          FOR i=0 to i=n DO
                FOR j=0 to j=n DO
                      IF (j != O and i!= L ):
                          ADD A[i][j] to List B
                      END IF
                END LOOP
          END LOOP
          SET  k =0
          FOR i =0 to i=Size(temp) DO
                FOR j=0 to j=Size(temp) DO
                      Set temp[O][i][j] = B[k]
                      k = k+1
                END LOOP
          END LOOP
    END LOOP

    FOR O = 0 to O=n DO
      IF (l+o) mod 2 == 0 THEN
         sum = sum + A[L][O]*DETERMINANT(temp[O],L,O)
      ELSE
         sum = sum - A[L][O]*DETERMINANT(temp[O],L,O)
      END IF
    END LOOP

   RETURN  sum
```

Algorithm Analysis:

$T(n) = C_1 + C_2 * (n)*(n^2 + (n-1)^2) + C_3 * (n+1) + C_4 * T(n-1)$
putting $C_1 + C_2 * (n)*(n^2 + (n-1)^2) + C_3 * (n+1) = \Theta(n^3)$

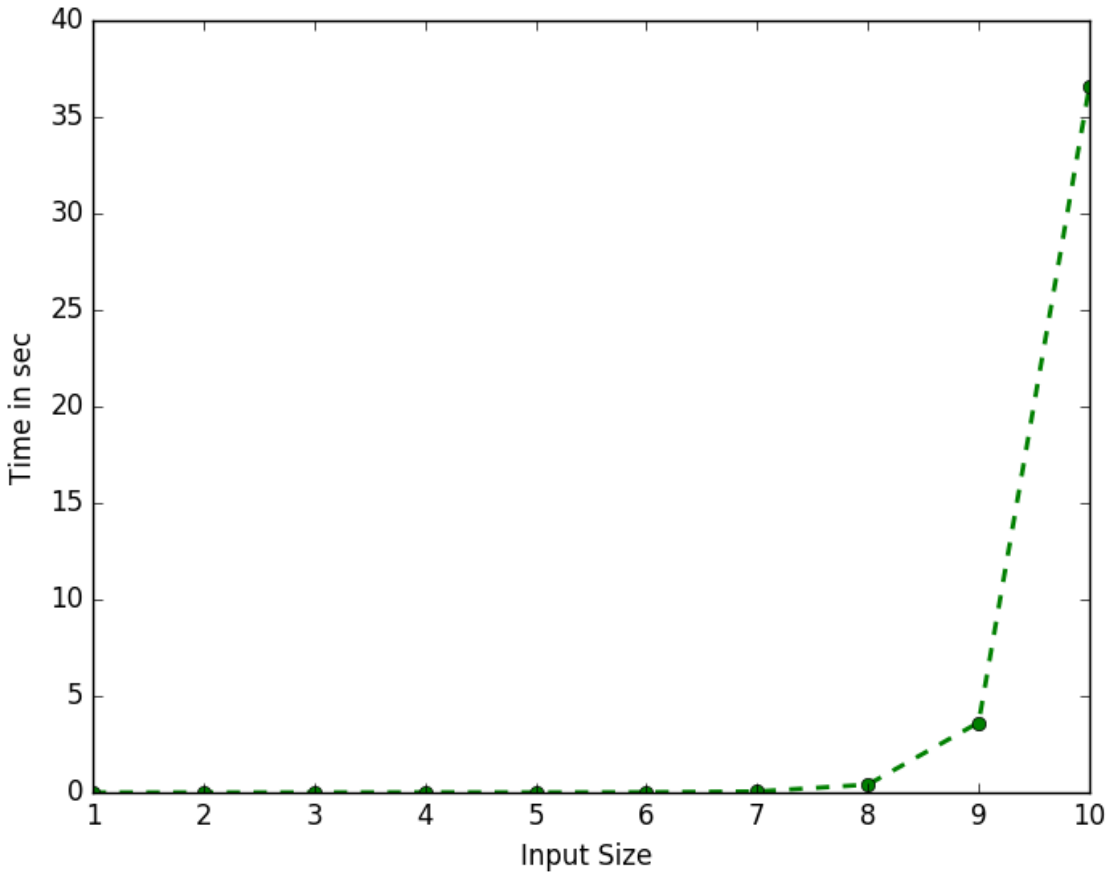
$T(n) = C * n * T(n-1) + \Theta(n^3)$
Where, n = n is Size of Square Matrix

$T(n) = O(n^3 * (n!))$

Input/Output:
input is n as Size of Square Matrix
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Numbers)	Time (in Sec)
1	-	0.0
2	-	0.0
3	-	0.001
4	-	0.008
5	-	0.003
6	-	0.012
7	-	0.053
8	**-**	**0.408**
9	**-**	**3.625**
10	**-**	**36.583**

Complexity Graph:



Conclusion:
Initially Curve is flat but after 8 as Size of Input Increase by one value graph grows very drastically. Graph is nonlinear with very high growth rate.

Q. 10 Spiral traverse of a Square Matrix.

Algorithm: SPIRALTRACE(A,n)

Input: A as 2D Matrix
Output: Element of matrix in Spiral Order.

SPIRALTRACE(A,n)

```
FOR i=0 to i=n-1 DO
        PRINT A[0][i]
END LOOP
FOR i=1 to i=n-1 DO
        PRINT A[i][n-1]
END LOOP
FOR i=n-2 to n=0
        PRINT A[n-1][i]
END LOOP
FOR i=n-2 to n=1
        PRINT A[i][0]
END LOOP
IF (n-2<=0) THEN
       return 0
ELSE
       INITIALIZE Array = Ø
       FOR i=1 to i=n-2
          FOR i=1 to i=n-2
             Array[i-1][j-1] = A[i][j]
          END LOOP
       END LOOP
END IF

RETURN SPIRALTRACE(Array,n-2)
END
```

Algorithm Analysis:

$$\mathbf{T(n) = C_1 * (n) + C_2 * (n-1) + C_3 * (n-2) + C_4 * (n-4) + C_5 * (n-3)^2 + C_6 * T(n-2)}$$

putting $\mathbf{C_1 * (n) + C_2 * (n-1) + C_3 * (n-2) + C_4 * (n-4) + C_5 * (n-3)^2 = \Theta(n^2)}$

$$\mathbf{T(n) = \Theta(n^2) + T(n-2)}$$
$$\mathbf{= \Theta(n^2) + \Theta(n-2)^2 + T(n-4)}$$
$$\mathbf{= \Theta(n^2) + \Theta(n-2)^2 + \Theta(n-4)^2 + T(n-6)}$$
$$\mathbf{.}$$
$$\mathbf{.}$$
$$\mathbf{= \Theta(n^2) + \Theta(n-2)^2 + \Theta(n-4)^2 + \Theta(n-6)^2 + ... + T(1)}$$

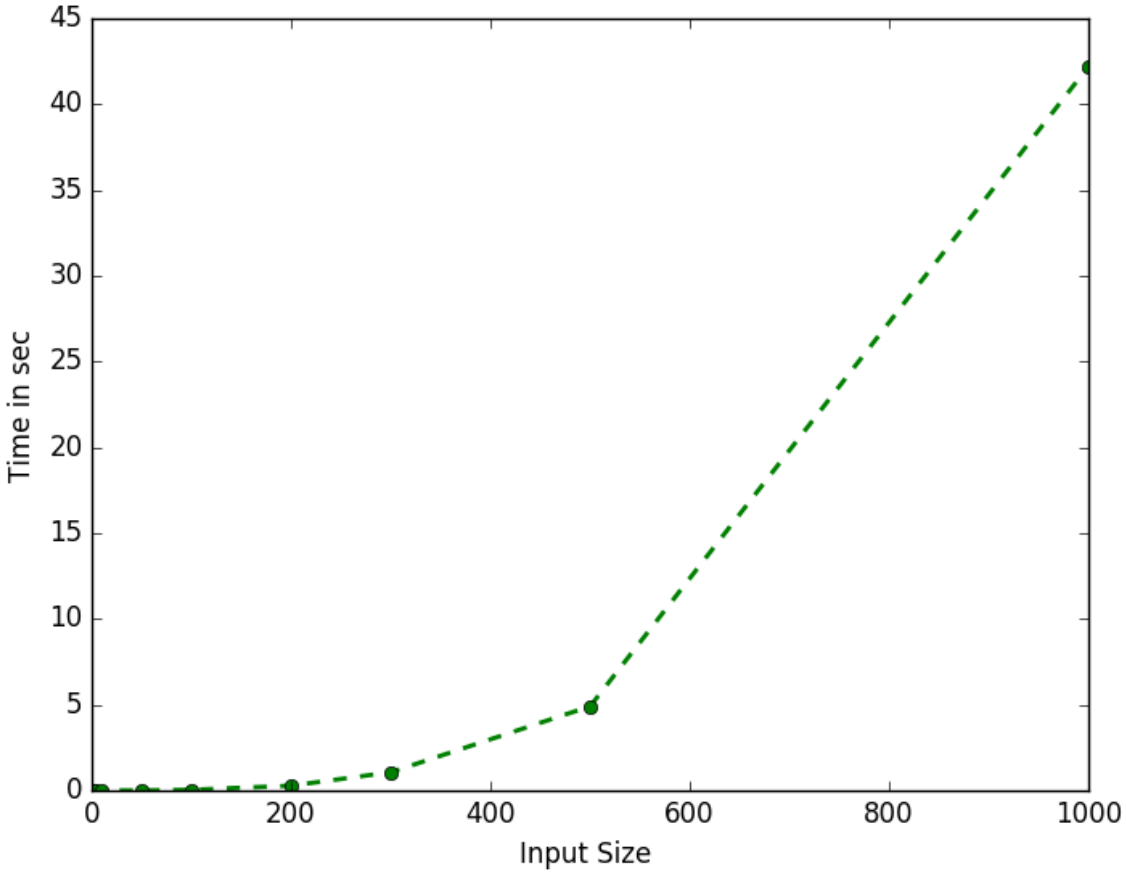
Where, n = n is Size of Square Matrix

$$\mathbf{T(n) = O(n^3)}$$

Input/Output:
input is n as Size of Square Matrix
At, CPU: Dual Core 3Ghz, Memory: 4GB

Input	Output (Numbers)	Time (in Sec)
1	-	0.0
2	-	0.0
3	-	0.0
10	-	0.008
50	-	0.005
100	-	0.042
200	-	0.279
300	**-**	**1.062**
500	**-**	**4.904**
1000	**-**	**42.201**

Complexity Graph:



Conclusion:
The curve grows non linearly but at higher rate than that of n^2 time complexity function, as its expected to be of $O(n^3)$ Time complexity.