Ex. No: 4 29x-04-2023	UIT2201 — Programming and Data Structures

Aim:

To execute the following programs and note the output.

PART – A

The purpose of this exercise is to design and analyze algorithms and perform empirical analysis of algorithms as well.

1. Let $p(x)$ be a polynomial of degree n, that is, $p(x) = \sum_{i=0}^{n} a_i x^i$.
   (a) Implement a simple $O(n^2)$-time algorithm using Python for computing $p(x)$, for a given value of x
   (b) Implement a $O(n \log n)$ algorithm for computing $p(x)$, based upon a more efficient calculation of x^i
   (c) Now, consider rewriting $p(x)$ as

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n)\cdots)))$$

   which is known as the Horner's method. Write a Python function to compute $p(x)$ using this method. Analyze the time complexity of your code and express the same in asymptotic notation.
   (d) Perform empirical analysis of run time of all the three versions: Execute the functions for different values of n (degree of the polynomial) and tabulate the results (note that each entry should be an average over several runs, say m). Use randomly generated values of $a_0, a_1, \ldots, a_{n-1}$ for each value of n. Perform ratio analysis with well known complexity classes to confirm the growth rateof running times of all the three versions.

Code:

implementation.py

```
count = 0
def solve_quadratic(coeffs, x):
    count = 0
    coefficients = coeffs
    count += 1
    value = 0
    count += 1
    for idx in range(0, len(coefficients)):
        temp = coefficients[idx]
        count += 1
        for i in range(len(coefficients) - (idx + 1)):
            temp *= x
            count += 1
        value += temp
        count += 1
    return value, count

def pow(x,y):
    global count
    count += 1
    if y == 0:
```

```
        return 1
    else:
        if y % 2 == 0:
            temp = pow(x, y//2)
            return temp * temp
        else:
            if y > 0:
                temp = pow(x, y//2)
                return x * temp * temp
            else:
                temp = pow(x, y//2)
                return (temp * temp)/x

def efficient_solve_quadratic(coeffs, x):
    global count
    count = 0
    coefficients = coeffs
    count += 1
    value = 0.0
    count += 1
    for idx in range(0, len(coefficients)):
        count += 1
        temp = coefficients[idx] * pow(x, len(coefficients) - (idx + 1))
        count += 1
        value += temp
        count += 1

    return value, count

def horners_method(coeffs, x):
    global count
    count = 0
    sum = coeffs[0]
    count += 1
    for idx in range(1, len(coeffs)):
        sum = sum * x + coeffs[idx]
        count += 1
    return sum ,count
```

analysis.py

```
from implementation import solve_quadratic, efficient_solve_quadratic,
horners_method
import random

if __name__ == "__main__":
    normalmethod = {}
    efficientmethod = {}
    hornersmethod = {}
    n = int(input("Enter degree: "))
    if n == 0:
        print("Degree cannot be zero!")
        exit()
    for degree in range(n, n+1):
        coeffs = []
        normal = []
        efficient = []
        horners = []
        for _ in range(1):
```

```
            coeffs = [random.uniform(-100.0, 100.0) for _ in range(degree)]
            xvalue = random.uniform(-100.0, 100.0)
            normal.append(solve_quadratic(coeffs=coeffs, x=xvalue)[1])
            efficient.append(efficient_solve_quadratic(coeffs=coeffs,
x=xvalue)[1])
            horners.append(horners_method(coeffs=coeffs, x=xvalue)[1])
        normalmethod[degree] = sum(normal)/len(normal)
        efficientmethod[degree] = sum(efficient)/len(efficient)
        hornersmethod[degree] = sum(horners)/len(horners)

    print("Normal Method:", normalmethod)

    print("=====================================")

    print("Efficient Method:", efficientmethod)

    print("=====================================")

    print("Horner's Method:", hornersmethod)
```

Sample Output:

```
Enter degree: 500
Normal Method: {500: 125752.0}
=====================================
Efficient Method: {500: 5991.0}
=====================================
Horner's Method: {500: 500.0}
```

Tabulation:

Analysis of Algorithm by Normal Method							
Degree(n)	f(n)	n	n^2	n^3	f(n)/n	$f(n)/n^2$	$f(n)/n^3$
1	4	1	1	1	4	4	4
10	67	10	100	1000	6.7	0.67	0.067
50	1327	50	2500	125000	26.54	0.5308	0.010616
100	5152	100	10000	1000000	51.52	0.5152	0.005152
500	125752	500	250000	1.25E+08	251.504	0.503008	0.001006
1000	501502	1000	1000000	1E+09	501.502	0.501502	0.000502
5000	12507502	5000	25000000	1.25E+11	2501.5	0.5003	0.0001
10000	50015002	10000	1E+08	1E+12	5001.5	0.50015	5E-05

Analysis of Algorithm by Efficient Method									
Degree(n)	f(n)	n	n^2	n * log n	n^3	f(n)/n * log n	f(n)/n	$f(n)/n^2$	$f(n)/n^3$
1	6	1	1	0	1	#DIV/0!	6	6	6
10	67	10	100	10	1000	6.7	6.7	0.67	0.067
50	439	50	2500	84.94850022	125000	5.16783697	8.78	0.1756	0.003512
100	975	100	10000	200	1000000	4.875	9.75	0.0975	0.000975
500	5991	500	250000	1349.485002	125000000	4.439471347	11.982	0.023964	0.000047928
1000	12979	1000	1000000	3000	1000000000	4.326333333	12.979	0.012979	0.000012979
5000	76811	5000	25000000	18494.85002	1.25E+11	4.153102075	15.3622	0.00307244	6.14488E-07
10000	163619	10000	100000000	40000	1E+12	4.090475	16.3619	0.00163619	1.63619E-07

Analysis of Algorithm by Horner's Method							
Degree(n)	f(n)	n	n^2	n^3	f(n)/n	$f(n)/n^2$	$f(n)/n^3$
1	1	1	1	1	1	1	1
10	10	10	100	1000	1	0.1	0.01
50	50	50	2500	125000	1	0.02	0.0004
100	100	100	10000	1000000	1	0.01	0.0001
500	500	500	250000	1.25E+08	1	0.002	0.000004
1000	1000	1000	1000000	1E+09	1	0.001	0.000001
5000	5000	5000	25000000	1.25E+11	1	0.0002	4E-08
10000	10000	10000	100000000	1E+12	1	0.0001	1E-08

Inference:

From the above tabulation, we can infer that,

- If f(n)/O(n) diverges to some random value, then our complexity is under-estimated.
- If f(n)/O(n) converges to zero, then our complexity is over-estimated.
- If the value tends to a particular constant value, then our complexity is the best estimate.

In normal method, the time complexity is $O(n^2)$.

In recursive method, the time complexity is O(nlogn)

In Horner's method, the time complexity is O(n).

Lower the time complexity, better and efficient is the algorithm.

Here Horner's method is the best and efficient algorithm as it has a time complexity of O(n).

==========