# EE441 – Programming Assignment 1

## Part – 1

```
// Add two matrices and return result as a new Matrix object
template <int N>
Matrix<N> matrixAddition(Matrix<N> m1, Matrix<N> m2)
{
    Matrix<N> newMatrix;
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            // Get pairwise elements of matrix 1 and 2
            int m1Data = m1.getElement(i,j);
            int m2Data = m2.getElement(i,j);

            // Add m1 to m2 and set result matrix data
            newMatrix.setElement(i,j,m1Data+m2Data);
        }
    }
    return newMatrix;
}
```

*Figure 1. matrixAddition() function*

As you can see from *Figure 1*, the ***matrixAddition()*** function accepts two inputs namely, *Matrix<N> m1 and Matrix<N> m2*. When we are calling this function, we are utilizing **pass by value** argument passing method. We are creating new copies rather than reading from the original address and returning new matrix. This type of argument passing method is utilized for ***matrixSubtraction()*** and ***matrixMultiplication()*** as well.

# Part – 2

## Question – 1e

```
void solve_hanoi_by_recursion(Hanoi &game,int number_of_disc, int from_rod, int
to_rod, int middle_rod)
{

    // If all discs move to to_rod, return
    if (number_of_disc == 0) ); //Line1 T1
    {
        cout << "Move count: " << game.move_count << endl;

        return;
    }
    //Line2 T2
    solve_hanoi_by_recursion(game,number_of_disc-1,from_rod,middle_rod,to_rod);


    //Line3 T3

    game.move(from_rod,to_rod);

    //Continue recursively to move all discs to to_rod

    //Line4 T4
    solve_hanoi_by_recursion(game,number_of_disc-1,middle_rod,to_rod,from_rod);
}

void solve_hanoi(Hanoi& game)
{
    solve_hanoi_by_recursion(game, game.number_of_disc,0,2,1);

}
```

*Figure 2. solve_hanoi() and solve_hanoi_by_recursion() functions*

As it can be seen from the code snippet in *Figure 2*, ***solve_hanoi()*** function has the algorithmic complexity of **O($2^N$)**. The function of ***solve_hanoi_by_recursion()*** is called multiple times under ***solve_hanoi()*** function.

For N=**0**, ***solve_hanoi_by_recursion()*** returns directly. **$2^0$-1=0**
For N=**1**, ***solve_hanoi_by_recursion()*** calls *Line2* to *Line4*. ***move()*** is called only once . **$2^1$ – 1=1**
For N=**2**, ***solve_hanoi_by_recursion()*** calls *Line2* to *Line4* but this times 2 times per each recursion. This results that ***move()*** is called 4 times. **$2^2$ – 1=3**
For N=**3**, ***solve_hanoi_by_recursion()*** calls *Line2* to *Line4* but this times 4 times per each recursion. This results that ***move()*** is called 8 times. **$2^3$ – 1=7**

This observation allows us to see that complexity of ***solve_hanoi_by_recursion()*** and therefore ***solve_hanoi()*** is **O($2^N$ – 1 )**. This can be approximated as **O($2^N$)**.

Deniz Karakay - 2443307

## Question – 3

```
int nth_prime(int n)
{
    int x = 1; // Line1 -T1
    int i = 2; // Line2 -T2
    int count = 0; // Line3 -T3
    // If count is less than n
    while(count < n) // Line4 -T4
    {
        x +=1; // Increment x by 1 Line5 -T5
        for(i = 2; i<x; i++) // Line6 – T6, T7, T8
        {
            if(x%i == 0) // Line7 -T9
            {
                break; //Line8 -T10
            }
        }
        if(i == x) // Line9 -T11
        {
            count +=1; // Line10 -T12
        }
    }
    return x; // Line11 -T13
}
```

*Figure 3. nth_prime() function*

As it can be seen from the code snippet in *Figure 3*, there are **13** lines to be executed in total. As you can see **Line 4** is a while loop that runs from **0** to **n** (input **n**). This loop runs like **T(n)**. There is an extra for loop on **Line 6** inside the while loop, but this loop is for checking if **x** is prime number or not. **T7** is dependent on **x**.

For instance when we begin for any n, **x** will be **2** firstly and the loop **(Line6)** is skipped. For n > 2, **x** will be incremented by **1** and loop **(Line6)** works only for **1** time. Then next time, **x** will be **4** and the loop works **2** times. Then this process goes till we have reached **n prime numbers**. As you can see the inside loop **(Line6)** is executed like **T(x)** especially **T7.** Therefore, in the end, we have kind of T(X) * T(N).

For N=**7**, X will be **16** in the end. For N = **50**, X will be **228** in the end. Hence, in the end we have almost worse [**O(N*N*4)**] result than **O($N^2$).** Since we have two loops our results approximate to **O($N^2$).**

Deniz Karakay - 2443307

# Main.Cpp Outputs

## Part 1 Matrix Construction - Matrix Addition, Subtraction, Multiplication – Finding Determinant

```
Select C:\Users\sedna\Desktop\CodeBlocks\Assignment1\assignment\bin\Debug\assignment.exe
M1:
10000
01000
00100
00010
00001

Getting element(0,1): 0
Getting element(0,0): 1
Setting element(0,0,1): 5

50000
01000
00100
00010
00001

Addition m1+m1
100000
02000
00200
00020
00002

Subtraction m1-m1
00000
00000
00000
00000
00000

M2:
222
333
444

M22:
111
555
777

Multiplication m2*m22
262626
393939
525252

100
010
001
determinant of m33: 1

111
555
777
determinant of m22: 0

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

*Figure 4. Output of main.cpp of part 1*

## Part 2 – Question 1 - Hanoi

```
Select C:\Users\sedna\Desktop\CodeBlocks\Assignment1\question2\qtwo\bin\Debug\qtwo.exe
#disc: 4
1 - 0 - 0
2 - 0 - 0
3 - 0 - 0
4 - 0 - 0
Move count: 0
2 - 0 - 0
3 - 0 - 0
4 - 1 - 0
Move count: 1
3 - 0 - 0
4 - 1 - 2
Move count: 2
3 - 0 - 1
4 - 0 - 2
Move count: 3
0 - 0 - 1
4 - 3 - 2
Move count: 4
1 - 0 - 0
4 - 3 - 2
Move count: 5
1 - 2 - 0
4 - 3 - 0
Move count: 6
0 - 1 - 0
0 - 2 - 0
4 - 3 - 0
Move count: 7
0 - 1 - 0
0 - 2 - 0
0 - 3 - 4
Move count: 8
0 - 2 - 1
0 - 3 - 4
Move count: 9
0 - 0 - 1
2 - 3 - 4
Move count: 10
1 - 0 - 0
2 - 3 - 4
Move count: 11
1 - 0 - 3
2 - 0 - 4
Move count: 12
0 - 0 - 3
2 - 1 - 4
Move count: 13
0 - 0 - 2
0 - 0 - 3
0 - 1 - 4
Move count: 14
0 - 0 - 1
0 - 0 - 2
0 - 0 - 3
0 - 0 - 4
Move count: 15

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

*Figure 5. Output of main.cpp of part 2 – Hanoi*

## Part 2 – Question 2 – Print Backwards

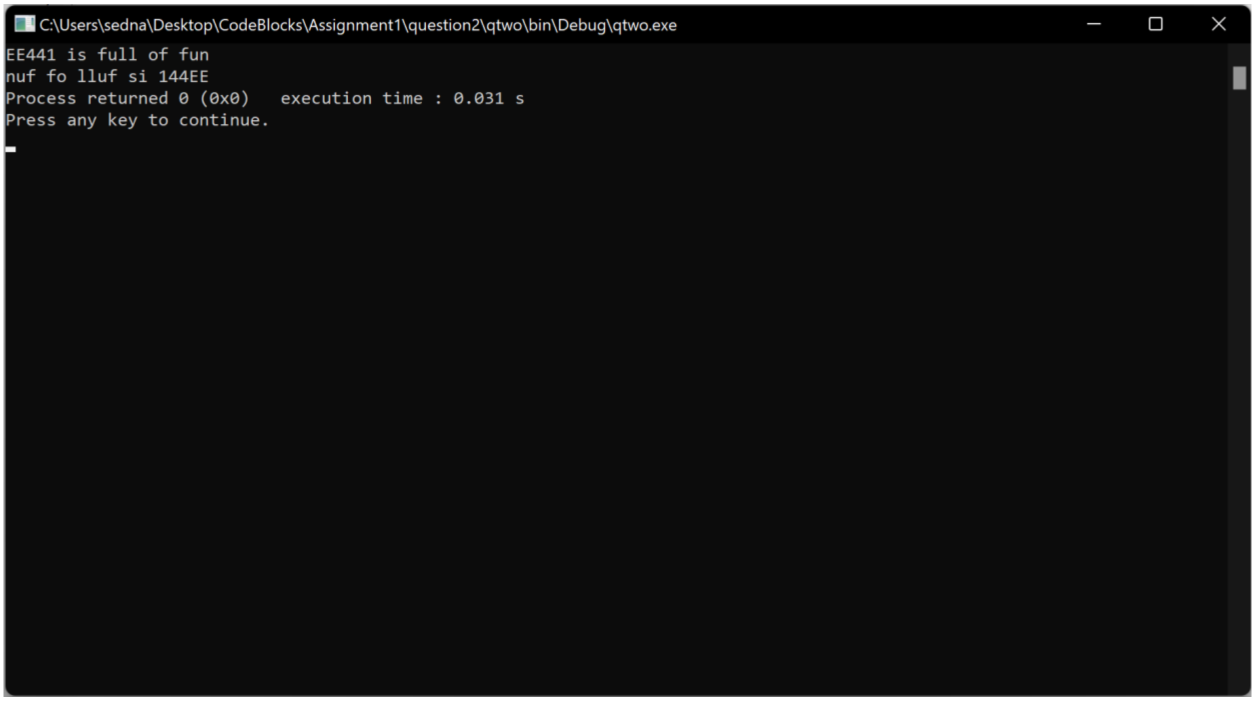



*Figure 6. Output of main.cpp of part 2 – Print Backwards*

## Part 2 – Question 3 – Nth Prime Number

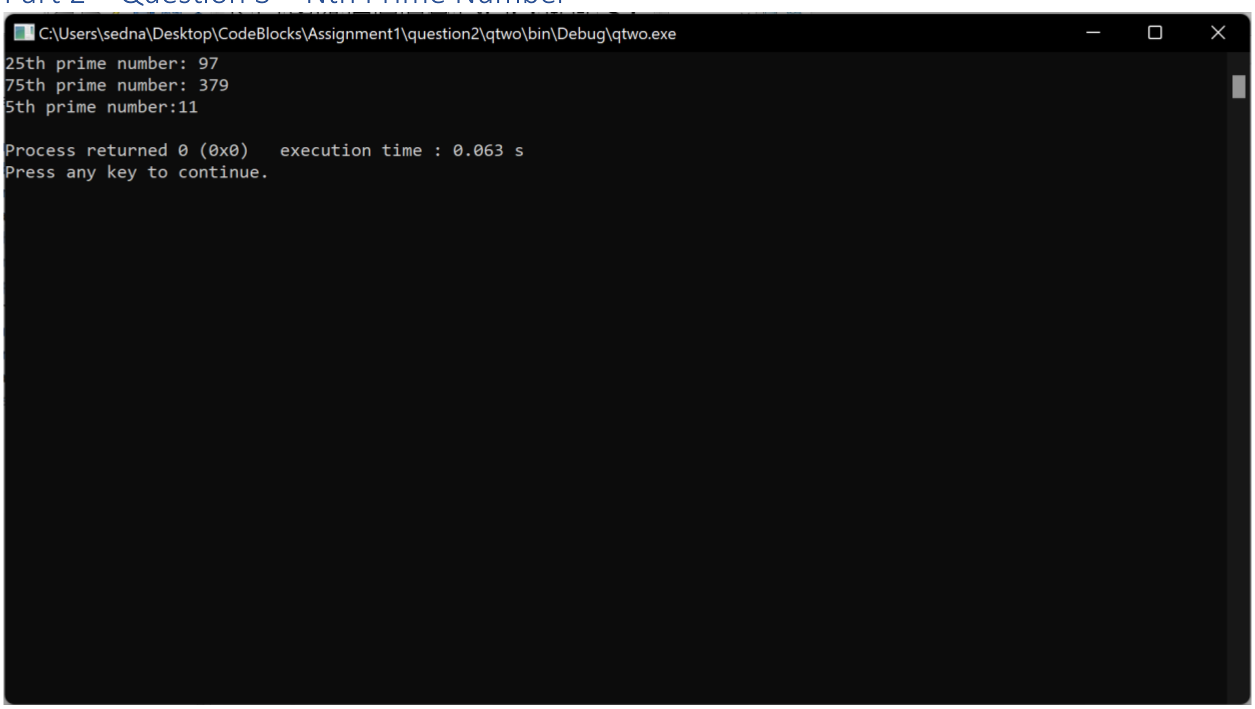



*Figure 7. Output of main.cpp of part 2 – Nth Prime*

# Q4 - Benchmark Results

## Part 2 – Question 1 - Hanoi

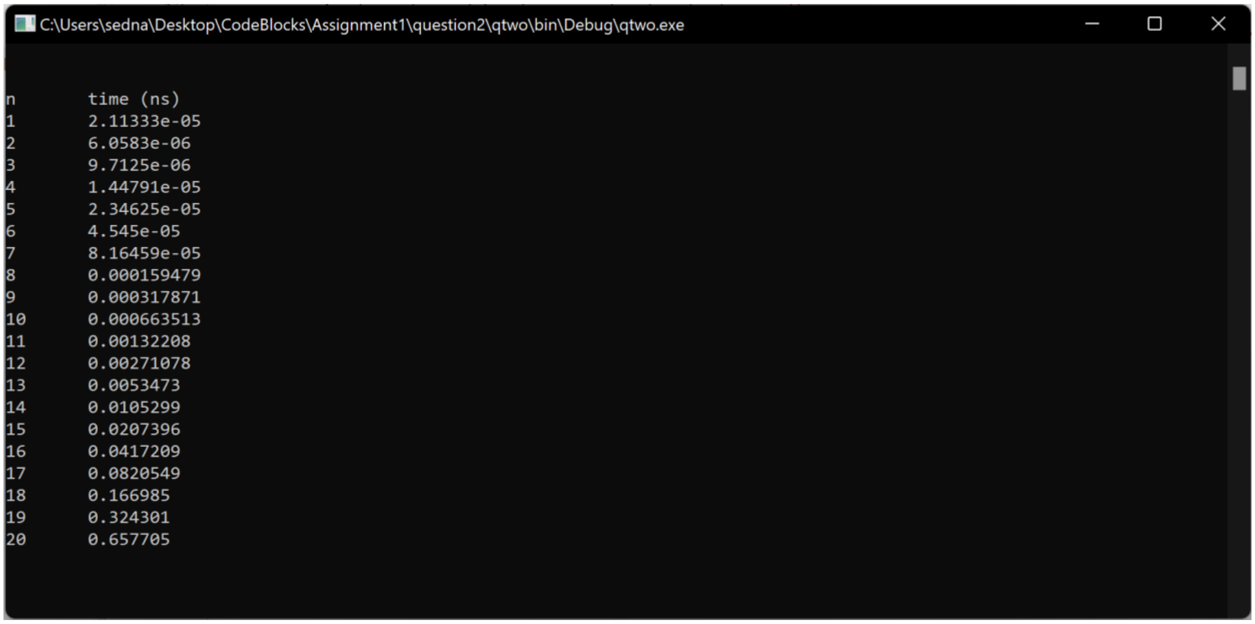



*Figure 8. Output of benchmark.cpp of part 2 – Hanoi*

As you can see from *Figure 8*, Hanoi has **O($2^N$)** type of algorithmic complexity, and it is same as my prediction.

## Part 2 – Question 2 – Print Backwards

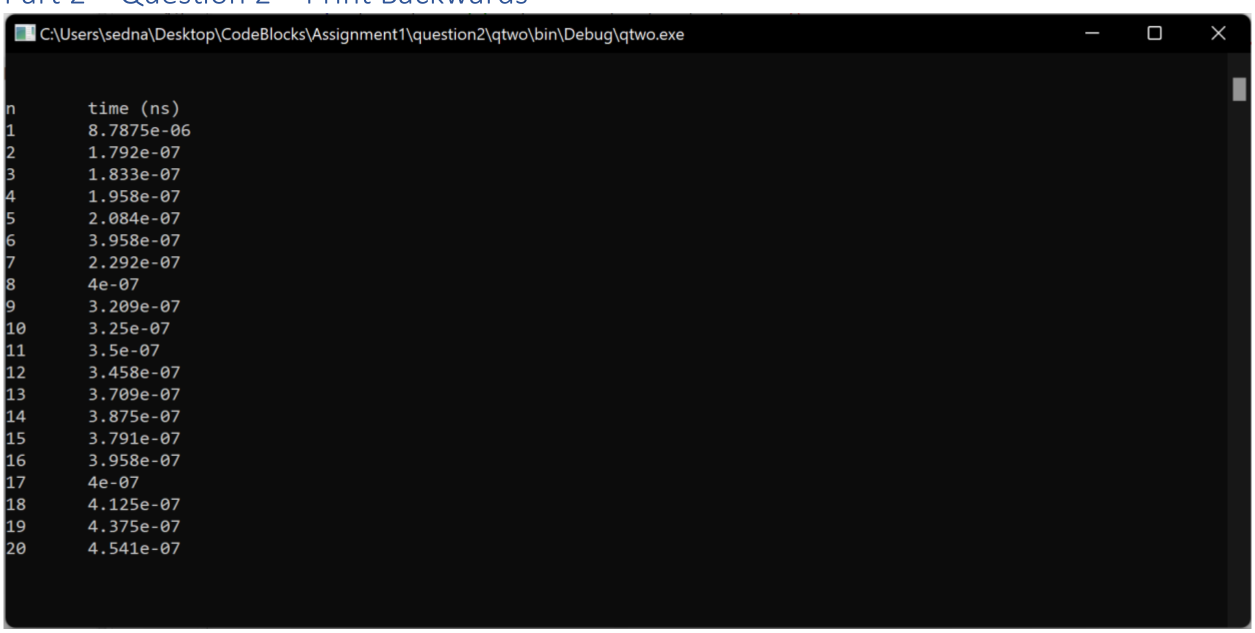
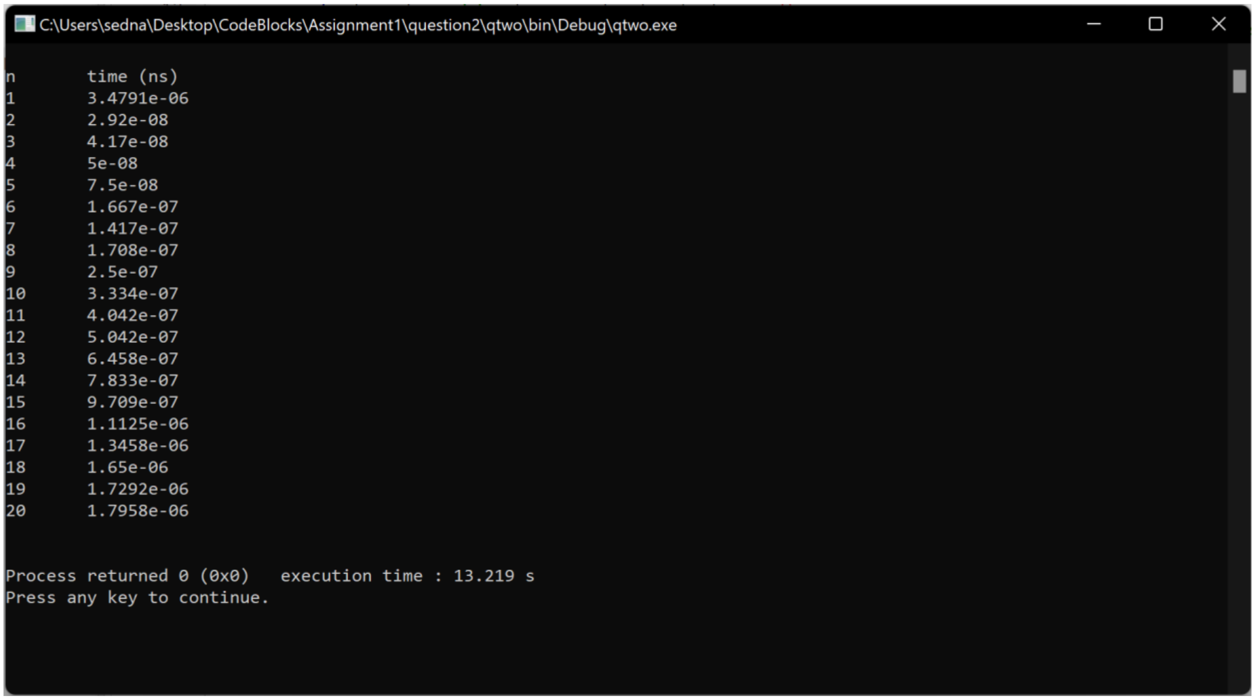



*Figure 9. Output of benchmark.cpp of part 2 – Print Backwards*

As you can see from *Figure 9*, *print_backwards()* has **O(N)** type of algorithmic complexity.

## Part 2 – Question 3 – Nth Prime




```
n       time (ns)
1       3.4791e-06
2       2.92e-08
3       4.17e-08
4       5e-08
5       7.5e-08
6       1.667e-07
7       1.417e-07
8       1.708e-07
9       2.5e-07
10      3.334e-07
11      4.042e-07
12      5.042e-07
13      6.458e-07
14      7.833e-07
15      9.709e-07
16      1.1125e-06
17      1.3458e-06
18      1.65e-06
19      1.7292e-06
20      1.7958e-06


Process returned 0 (0x0)   execution time : 13.219 s
Press any key to continue.
```

*Figure 10. Output of benchmark.cpp of part 2 – Nth Prime*

As you can see from *Figure 10*, *nth_prime()* has **O(N$^2$)** type of algorithmic complexity. This is same as my assumption.