# FDS Project1 Report
# Performance Measurement of POW

Name

2023.9.28

# 1 Introduction

## 1.1 Background

The formula $X^N$ is frequently used in all kinds of fields and algorithms. However, the calculation of $X^N$ is time-consuming, especially when $N$ is large. Therefore, we need to find a way to accelerate the calculation of $X^N$. In this project we compare and analyze three distinct algorithms: Multiples, Iteration, and Recursion, which are used for calculating $X^N$, with a focus on their runtime performance. This analysis can provide insights and conclusions for selecting the most suitable algorithm for specific scenarios and optimizing program performance.

## 1.2 Problem Statement

To compare the performance of these algorithms, we keep $X$ fixed and vary the value of $N$ as input for the calculations. We utilize the clock function from the time library to measure the execution time of each algorithm and take multiple measurements to obtain an average for more accurate results. According to our existing knowledge, the Multiples method has a time complexity of $O(N)$, the Iteration and the Recursion method has a time complexity of $O(log(N))$. However, actual runtime can also be influenced by other factors such as hardware performance and compiler optimization.

# 2 Algorithm Specification

First, to make sure that the algorithms are implemented correctly, we compare the result with that of the built-in function `pow()` in `math.h`. Whenever the algorithm's result goes out of the threshold of error, we print an error message. In order to record the execution time of each algorithm, we need to use the clock function from the time library. The clock function returns the number of clock ticks elapsed since the program was launched. Here in `time.h`, `CLOCKS_PER_SEC` is a constant 1000. We declare $K_i$ as the number of loops (or repetitions) for each algorithm, and $Ticks$ as the difference between the clock ticks before and after the algorithm is executed. So the total time duration of the algorithm is $\frac{Ticks}{CLOCKS\_PER\_SEC}$. And the time duration of a single process of the algorithm can be calculated by

$$\frac{Ticks}{CLOCKS\_PER\_SEC \times K_i}$$

The pseudo-code is shown as below:

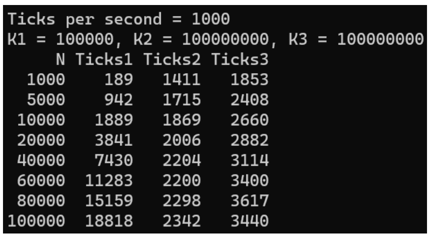
```
Define number of loops for each algorithm:
K1 = 100000, K2 = 100000000, K3 = 100000000
Define the threshold of error: dE = 0.000001
Define three functions for calculating X^N:
Function Multiples(X, N):
    Loop i = 0 to N-1: Result = Reslut * X
    Return result

Function Iteration(X, N):
    Loop while N is greater than 0:
        If N is odd: Result = Reslut * X
        X = X * X, N = N / 2
    Return result
```

```
Function Recursion(X, N):
    If N equals 1: Return X
    Else, if N is odd: Return Recursion(X * X, N / 2) * X
    Else: Return Recursion(X * X, N / 2)

Main function:
Assign X and N
Declare a 2D array Ticks[][] for recording execution time

Loop i from 0 to 7:
    Record current tick in 'start'
    Loop j = 0 to K1-1:
        Call Multiples(X, N[i])
    Record current tick in 'stop'
    if(absolute(Multiples(X, N[i]) - pow(X, N[i])) > dE): Print "Error!"
    Assign (stop - start) to Ticks[0][i]
Do the same for Iteration and Recursion and store the results.
```

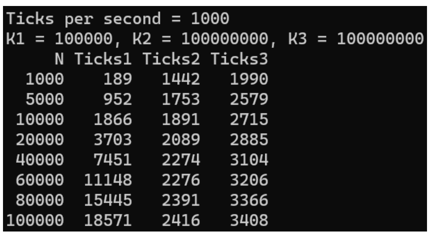
# 3 Testing Result

Table 1: The Runtime of Three Algorithms

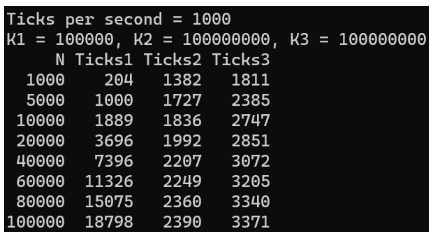
| | N | 1000 | 5000 | 10000 | 20000 | 40000 | 60000 | 80000 | 100000 |
|---|---|---|---|---|---|---|---|---|---|
| Algorithm1 Multiples | Iteration(K) | $10^5$ | $10^5$ | $10^5$ | $10^5$ | $10^5$ | $10^5$ | $10^5$ | $10^5$ |
| | Ticks | 190 | 924 | 1835 | 3653 | 7349 | 11038 | 14527 | 18318 |
| | Total time(sec) | 0.190 | 0.924 | 1.835 | 3.653 | 7.349 | 11.038 | 14.527 | 18.318 |
| | Duration($10^{-8}$sec) | 190 | 924 | 1835 | 3653 | 7349 | 11038 | 14527 | 18318 |
| Algorithm2 Iteration | Iteration(K) | $10^8$ | $10^8$ | $10^8$ | $10^8$ | $10^8$ | $10^8$ | $10^8$ | $10^8$ |
| | Ticks | 1433 | 1725 | 1917 | 2075 | 2250 | 2296 | 2400 | 2417 |
| | Total time(sec) | 1.433 | 1.725 | 1.917 | 2.075 | 2.250 | 2.296 | 2.400 | 2.417 |
| | Duration($10^{-8}$sec) | 1.433 | 1.725 | 1.917 | 2.075 | 2.250 | 2.296 | 2.400 | 2.417 |
| Algorithm3 Recursion | Iteration(K) | $10^8$ | $10^8$ | $10^8$ | $10^8$ | $10^8$ | $10^8$ | $10^8$ | $10^8$ |
| | Ticks | 1885 | 2454 | 2696 | 2935 | 3132 | 3199 | 3365 | 3441 |
| | Total time(sec) | 1.885 | 2.454 | 2.696 | 2.935 | 3.132 | 3.199 | 3.365 | 3.441 |
| | Duration($10^{-8}$sec) | 1.885 | 2.454 | 2.696 | 2.935 | 3.132 | 3.199 | 3.365 | 3.441 |



```
Ticks per second = 1000
K1 = 100000, K2 = 100000000, K3 = 100000000
     N Ticks1 Ticks2 Ticks3
  1000    189   1411   1853
  5000    942   1715   2408
 10000   1889   1869   2660
 20000   3841   2006   2882
 40000   7430   2204   3114
 60000  11283   2200   3400
 80000  15159   2298   3617
100000  18818   2342   3440
```



```
Ticks per second = 1000
K1 = 100000, K2 = 100000000, K3 = 100000000
     N Ticks1 Ticks2 Ticks3
  1000    189   1442   1990
  5000    952   1753   2579
 10000   1866   1891   2715
 20000   3703   2089   2885
 40000   7451   2274   3104
 60000  11148   2276   3206
 80000  15445   2391   3366
100000  18571   2416   3408
```



```
Ticks per second = 1000
K1 = 100000, K2 = 100000000, K3 = 100000000
     N Ticks1 Ticks2 Ticks3
  1000    204   1382   1811
  5000   1000   1727   2385
 10000   1889   1836   2747
 20000   3696   1992   2851
 40000   7396   2207   3072
 60000  11326   2249   3205
 80000  15075   2360   3340
100000  18798   2390   3371
```

Figure 1, 2, 3: Some of the Test Results on the Terminal

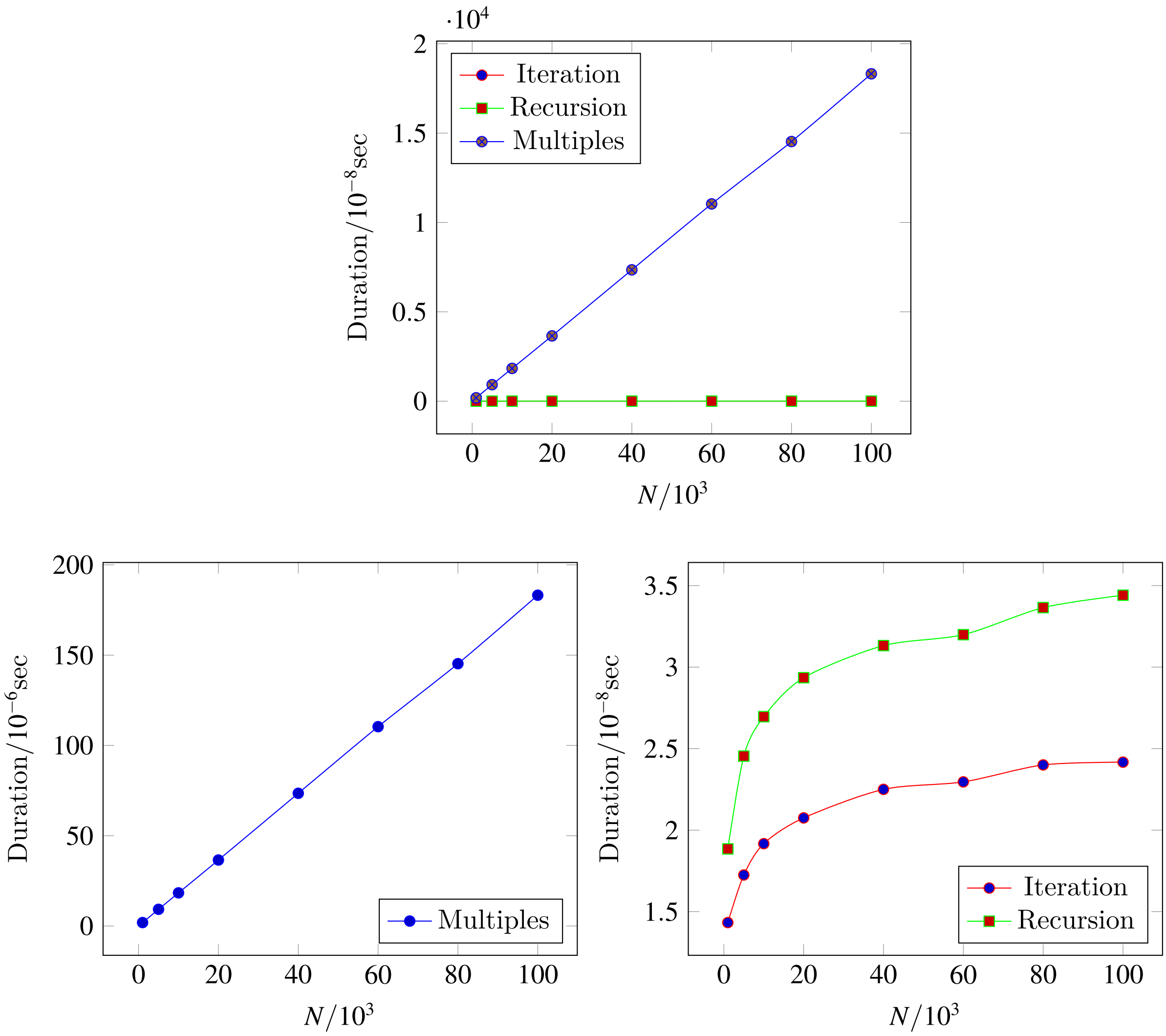



Figure 4, 5, 6: The Growth of Runtime with N

**Note**

The unit of time is $10^{-6}$sec for Multiples and $10^{-8}$sec for Iteration and Recursion. So if we force the two figures to be in the same coordinate system like Figure 1, the Iteration and Recursion's curve will be much flatter than Multiples', which is not intuitive and of nonsense.

# 4 Analysis and Comments

## 4.1 The Theoretical Time and Space Complexity of Three Algorithms

1. **Multiples**

   **Time Complexity:** $O(N)$

   This function uses a simple loop to perform $N$ multiplications on $X$. Each iteration requires one multiplication operation, so as N increases, the number of multiplication operations also increases linearly. Therefore, the time complexity of the Multiples function is $O(N)$.

   **Space Complexity:** $O(1)$

   This function only uses one (or two) variable to calculate and store the result, so the space complexity is $O(1)$.

2. **Iteration**

   **Time Complexity:** $O(log(N))$

First, represent $N$ in binary form. Then, start traversing from the rightmost bit of the binary representation. If the current bit is 1, multiply the result by $X$; if the current bit is 0, square $X$. In each iteration, $X$ is squared regardless of whether the current bit is 0 or 1. Hence, the number of iterations is equal to the number of 1s in the binary representation of $N$. Since the number of 1s in the binary representation of $N$ is at most $\log_2 N$, the time complexity of the Iteration function is $O(\log_2 N)$.

**Space Complexity:** $O(1)$

This function only uses less than 3 variables to calculate and store the result, so the space complexity is $O(1)$. So it's easy to figure out from the figure and the data that the Iteration is almost the most efficient algorithm among the three.

3. **Recursion**

   **Time Complexity:** $O(log(N))$

   The recursion stops when $N$ equals 1. In each recursive call, one square operation and one multiplication operation are performed. The overall recursion depth depends on the value of $N$. Let's assume the recursion depth is $d$. The problem size is halved in each recursive call. Therefore, $d$ satisfies $2^d = N$, i.e., $d = log_2 N$. In each recursive level, one square operation and one multiplication operation are performed. Thus, the time complexity of the Recursion function is proportional to the recursion depth, which is $O(\log_2 N)$.

   **Space Complexity:** $O(log(N))$

   The recursion depth is $O(\log_2 N)$, and for each recursive call, one or two variables are used in the stack frame. So the space complexity is $O(\log_2 N)$. So as $N$ increases, the growth of the space complexity cause the function to spend more time on the stack operation, i.e. the function call and return, and the stack's load and store.

## 4.2 Test Result Analysis basing on the Theoretical Time Complexity

1. Multiples

   The figure of Multiples (Figure 5) is a straight line, approximately $T(M)(10^{-6}s) = 1.8525N(10^2) - 0.0675$ by fitting process, which is consistent with the theoretical time complexity of $O(N)$.

2. Iteration and Recursion

   The figure of Iteration and Recursion (Figure 6) is a curve. To be more specific, the function of Iteration is $T(I)(10^{-8}s) = 0.087\log_2 N(10^2) + 1.249$ and the function of Recursion id $T(R)(10^{-8}s) = 0.177\log_2 N(10^2) + 1.108$ by fitting process, which is consistent with the theoretical time complexity of $O(log(N))$. **However, I'm not sure if this fitting process is accurate enough, because the data is limited and the base of the logarithm can't be determined. In order to present the result better, Here I just assume that the base is 2.** So as $N$ increases, the growth of these two algorithms' runtime will significantly slow down for orders of magnitude, almost $10^2$ times when $N$ increases to $10^5$. As the data, the figure and the fitting equations' coifficient show, we find that though the curves' trend is similar, the Iteration is always faster than Recursion.

## 4.3 The Influencing Factors in the Actual Runtime

1. Hardware Performance

   I've run the program several times on my PC and the results are different with a largest difference of approximately 4 times. So I choose one of those results which seem to be more close to the general trend as the data in Table 1.

2. The Test data $N$ and The Number of Loops $K$
   In Iteration and Recursion's result from the same group of test, there're sometimes one or two data that are not consistent with the general trend. For example, in figure 2, in the test of $N = 60000$, the time duration is apparently shorter than the general trend. Though $N$ seems to be large enough, there're still some unexpected results. I also find that the larger $K$ is, the longer a single loop might take.

3. The Way the Algorithm is Implemented
   As is known to all, the bit-operation is always faster than the arithmetic operation. So I always use `N/2` instead of `N>>1` in order to create a more *fair* environment for the three algorithms. If I use `N>>1`, the Iteration and Recursion will be much faster than Multiples.

4. Other Possible Reasons
   I've test the program on **Dev-C++** and **VSCode**. It turns out that the program always runs faster on **VSCode** with **MinGW** compiler than on **Dev-C++** with **gcc** compiler. I'm not sure if it was because the compiler counts or it was just because that **VSCode** asked for more CPU resources.

## Appendix: Source Code in C

```c
#include<stdio.h>
#include<time.h>
#include<math.h>

// three loop numbers for each function
#define K1 100000 // for function1 Multiples
#define K2 100000000 // for function2 Iteration
#define K3 100000000 // for function3 Recursion
#define dE 0.000001 // the threshold of error

// clock time record
clock_t start, stop, Ticks[3][8];

// three functions to calculate the X^N
double Multiples(double X, int N);
double Iteration(double X, int N);
double Recursion(double X, int N);

int main(){
    double X = 1.0001; // the base of X
    int N[8] = {1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000};
        // the power of X
    int i, j; // loop variables
```

```
    // calculate the time for each function
    for(i = 0; i < 8; i++){
        start = clock(); // start time
        for(j = 0; j < K1; j++) Multiples(X, N[i]);
        stop = clock(); // stop time
        if(abs(Multiples(X, N[i]) - pow(X, N[i])) > dE) printf("Error!\n"
           ); // check the result is correct or not
        Ticks[0][i] = stop - start; // calculate the time ticks
    }
    for(i = 0; i < 8; i++){ // do the same for Iteration and Recursion
        start = clock();
        for(j = 0; j < K2; j++) Iteration(X, N[i]);
        stop = clock();
        if(abs(Iteration(X, N[i]) - pow(X, N[i])) > dE) printf("Error!\n"
           );
        Ticks[1][i] = stop - start;
    }
    for(i = 0; i < 8; i++){ // do the same for Iteration and Recursion
        start = clock();
        for(j = 0; j < K3; j++) Recursion(X, N[i]);
        stop = clock();
        if(abs(Recursion(X, N[i]) - pow(X, N[i])) > dE) printf("Error!\n"
           );
        Ticks[2][i] = stop - start;
    }

    // print the result
    printf("Ticks per second = %d\n", CLOCKS_PER_SEC); // print the clock
         ticks per second
    printf("K1 = %d, K2 = %d, K3 = %d\n", K1, K2, K3); // print the loop
       numbers
    printf("     N Ticks1 Ticks2 Ticks3\n"); // print the table head
    for(i = 0; i < 8; i++){
        printf("%6d %6ld %6ld %6ld\n", N[i], Ticks[0][i], Ticks[1][i],
           Ticks[2][i]); // print the table
    }
    return 0;
}

//function1 multiplications
double Multiples(double X, int N){
    int i;
    double result = 1;
```

```
    for(i = 0; i < N; i++){
        result *= X;    // X^N = X * X * X * ... * X
    }
    return result;
}

//function2 iteration
double Iteration(double X, int N){
    double result = 1;
    while(N > 0){
        if(N % 2 == 1){
            result *= X; //if N is odd, X^N = X * X^(N-1)
        }
        X *= X;     // X^N = X^(N/2) * X^(N/2)
        N /= 2;
    }
    return result;
}

//funtion3 recursion
double Recursion(double X, int N){
    if(N == 0) return 1; // X^0 = 1 (N >= 0
    if(N == 1) return X; // recusion out condition
    else if(N % 2) return Recursion(X * X, N / 2) * X; //if N is odd, X^N
        = X * X^(N-1)
    else return Recursion(X * X, N / 2); // X^N = X^(N/2) * X^(N/2)
}
```

## Delaration

I hereby declare that all the work done in this project titled "Performance Measurement of POW" is of my independent effort.