Project1

Performance Measurement

(POW)

xxx

Date: 2023-10-09

Chapter 1: Introduction

There are at least two different algorithms that can compute X^N for some positive integer N. Algorithm 1 is to use (N−1) multiplications. Algorithm 2 works in the following way: if N is even, $X^N=X^{N/2} \times X^{N/2}$; and if N is odd, $X^N=X^{(N-1)/2}\times X^{(N-1)/2}\times X$. And Algorithm2 can have 2 versions. One is iterative and the other is recursive.

Chapter 2: Algorithm Specification

2.1 Algorithm 1

2.1.1 Description

Algorithm 1 calculates the result of raising a double value x to the power of a long integer n through a simple iterative approach. It runs the calculation k times to measure the average time it takes to perform the operation.

2.1.2 Pseudo-code

Initialize double x = X; long n = N; clock_t time[k+1]; long double result = 1;

```
for(int i = 0; i < k+1; i++) {
    result = 1;
    start = clock();

    for(long j = 0; j < n; j++) {
        result *= x;
    }

    stop = clock();
    duration = stop - start;
    time[i] = duration;
}
```

Calculate the average time (aver) and print it.

Print the result.

2.1.3 Main Data Structures

- x: A double variable representing the base.

- n: A long integer representing the exponent.

- time: An array of clock_t values to store the execution time for each iteration.

- result: A long double variable to store the result of the power operation.

2.2 Algorithm 2

2.2.1 Description

Algorithm 2 calculates the result of raising a double value x to the power of a long integer n using a more efficient approach based on exponentiation by squaring. It also runs the calculation k times to measure the average time it takes to perform the operation.

2.2.2 Pseudo-code:

Initialization is the same as algorithm1.

```
for(int i = 0; i < k+1; i++) {
    reset result, x and n;
    start = clock();

    while(n != 0) {
        if(n % 2) {
            result *= x;
        }
        x = x * x;
        n /= 2;
    }

    stop = clock();
    duration = stop - start;
```

```
    time[i] = duration;
}
```

Calculate the average time (aver) and print it.

Print the result.

2.2.3 Main Data Structures are the same as algorithm1.

2.3 Algorithm 3

2.3.1Description

Algorithm 3 also calculates the result of raising a double value x to the power of a long integer n using a recursive approach. It runs the calculation k times to measure the average time it takes to perform the operation. Additionally, it defines a helper function mypow for the recursive calculation.

2.3.2 Pseudo-code

Initialization is the same as algorithm1.

```
for(int i = 0; i < k+1; i++) {
    result = 1;
    start = clock();

    result = mypow(x, n);

    stop = clock();
    duration = stop - start;
    time[i] = duration;
}
```

Calculate the average time (aver) and print it.

Print the result.

2.3.2.1 Helper Function (mypow)

```
long double mypow(double x, long n) {
    if(n == 0) {
        return 1;
    } else if(n % 2) {
        return mypow(x, (n-1)/2) * mypow(x, (n-1)/2) * x;
    } else {
        return mypow(x, n/2) * mypow(x, n/2);
    }
}
```

2.3.3 Main Data Structures are the same as algorithm1.

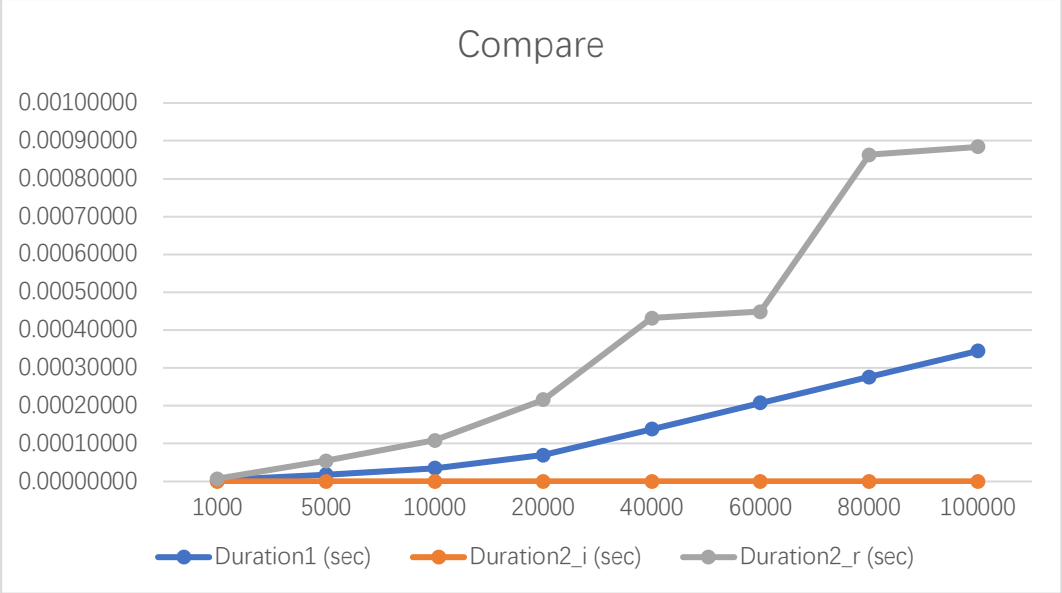
These algorithms use basic data types and arrays to perform calculations and measure execution times.

Chapter 3: Testing Results

Table of test cases.

	N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm1	K	100000	100000	100000	100000	100000	100000	100000	100000
	Ticks	374515	1786137	3467704	6959373	13797064	20753541	27619939	34470586
	Total Time (sec)	0.37451500	1.78613700	3.46770400	6.95937300	13.79706400	20.75354100	27.61993900	34.47058600
	Duration1 (sec)	**0.00000375**	**0.00001786**	**0.00003468**	**0.00006959**	**0.00013797**	**0.00020754**	**0.00027620**	**0.00034471**
Algorithm2 (interative)	K	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
	Ticks	328971	325307	330674	332723	336731	336820	338662	337972
	Total Time (sec)	0.32897100	0.32530700	0.33067400	0.33272300	0.33673100	0.33682000	0.33866200	0.33797200
	Duration2_i (sec)	**0.00000033**	**0.00000033**	**0.00000033**	**0.00000033**	**0.00000034**	**0.00000034**	**0.00000034**	**0.00000034**
Algorithm2 (recursive)	K	100000	100000	100000	100000	100000	100000	100000	100000
	Ticks	751610	5418817	10834460	21626240	43201748	44893607	86311859	88392700
	Total Time (sec)	0.75161000	5.41881700	10.83446000	21.62624000	43.20174800	44.89360700	86.31185900	88.39270000
	Duration2_r (sec)	**0.00000752**	**0.00005419**	**0.00010834**	**0.00021626**	**0.00043202**	**0.00044894**	**0.00086312**	**0.00088393**

The compare plot of three algorithms are displayed below.



Chapter 4: Analysis and Comments

Analysis of the time and space complexities of the algorithms.

4.1 Algorithm 1

4.1.1 Time Complexity

Inside the loop, there's a loop that runs n times. the time complexity of Algorithm 1 is O(n).

4.1.2 Space Complexity

The space complexity is relatively low and the space complexity is O(1), which means the algorithm only needs constant space.

4.2 Algorithm 2

4.2.1 Time Complexity

Algorithm 2 employs exponentiation by squaring, which has a time complexity of O(log(n)).

4.2.2 Space Complexity

Similar to Algorithm 1, the space complexity is O(1).

4.3 Algorithm 3

4.3.1 Time Complexity

Algorithm 3 uses a recursive approach, and its time complexity is influenced by the number of recursive calls. The time complexity depends on the depth of the recursion tree, which is O(log(n)).

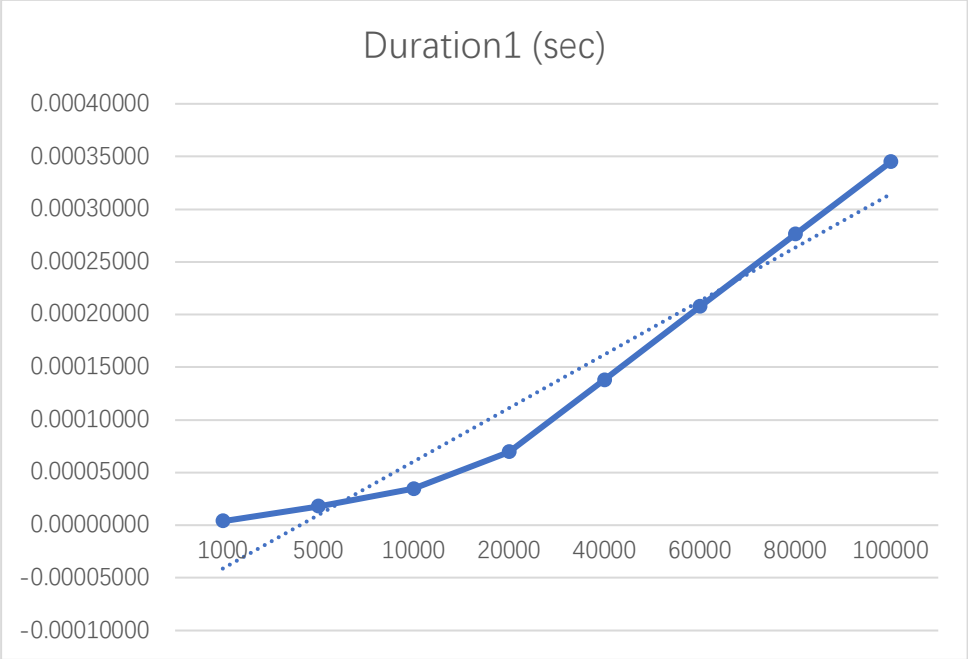
4.3.2 Space Complexity

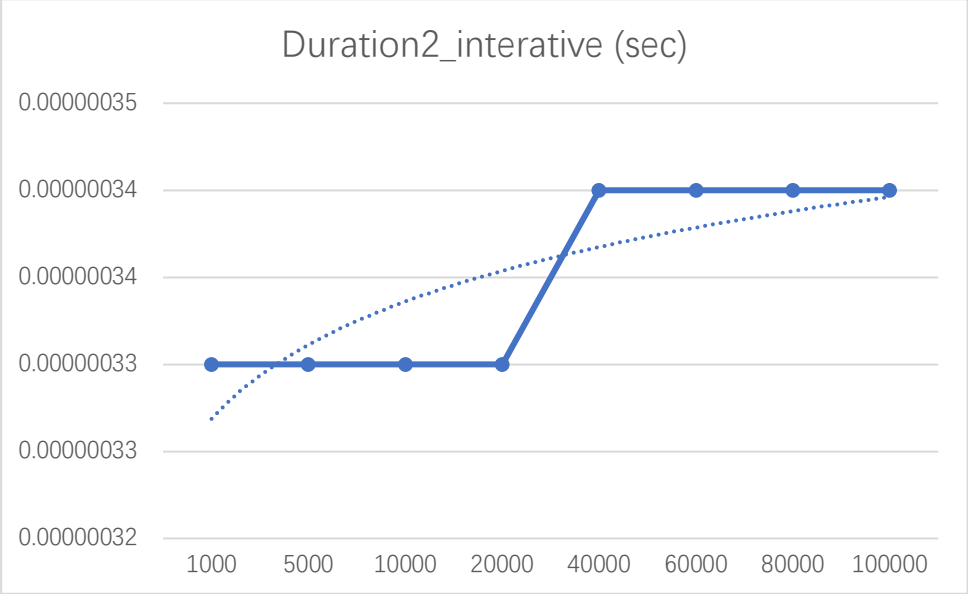
The space complexity of Algorithm 3 is higher due to the recursive calls. The maximum depth of the recursion tree is log(n), leading to a space complexity of O(log(n)) for the function call stack.

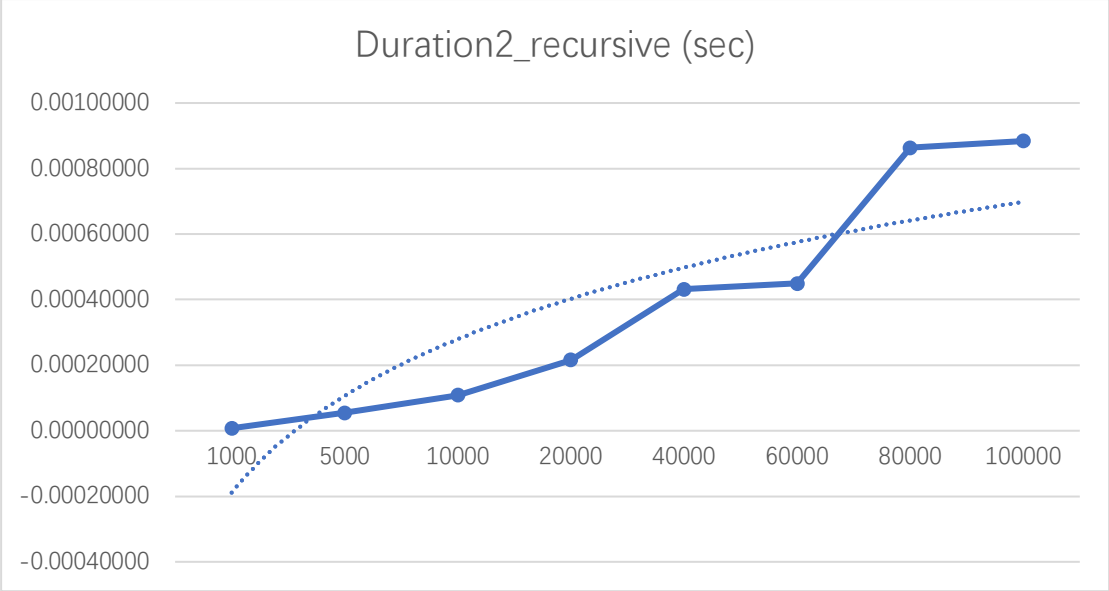
4.4 General Comments

Time complexity and space complexity can't be achieved simultaneously. Low time complexity and space complexity must result in more complex algorithms that need more thinking.

4.5 Detailed plots for each algorithm







Appendix: Source Code (in C)

The source code are in a separate folder with executable file (for mac).

Declaration

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