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.

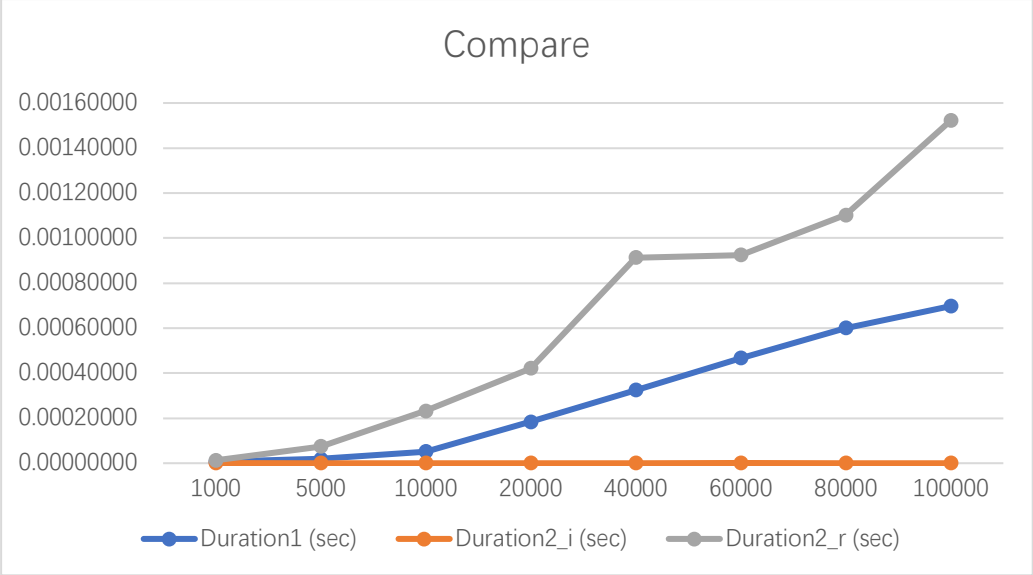
These algorithms use basic data types and arrays to perform calculations and measure execution times.

Chapter 3: Testing Results

Table of test cases.

	N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1	K	50	50	50	50	50	50	50	50
	Ticks	339	950	2578	9221	16238	23345	30065	34893
	Total Time (sec)	0.00033900	0.00095000	0.00257800	0.00922100	0.01623800	0.02334500	0.03006500	0.03489300
	Duration1 (sec)	**0.00000678**	**0.00001900**	**0.00005156**	**0.00018442**	**0.00032476**	**0.00046690**	**0.00060130**	**0.00069786**
Algorithm 2 (interative)	K	100	100	100	100	100	100	100	100
	Ticks	69	76	89	83	92	103	99	101
	Total Time (sec)	0.00006900	0.00007600	0.00008900	0.00008300	0.00009200	0.00010300	0.00009900	0.00010100
	Duration2_i (sec)	**0.00000069**	**0.00000076**	**0.00000089**	**0.00000083**	**0.00000092**	**0.00000103**	**0.00000099**	**0.00000101**
Algorithm 2 (recursive)	K	30	30	30	30	30	30	30	30
	Ticks	386	2253	6987	12648	27390	27737	33065	45679
	Total Time (sec)	0.00038600	0.00225300	0.00698700	0.01264800	0.02739000	0.02773700	0.03306500	0.04567900
	Duration2_r (sec)	**0.00001287**	**0.00007510**	**0.00023290**	**0.00042160**	**0.00091300**	**0.00092457**	**0.00110217**	**0.00152263**

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)).

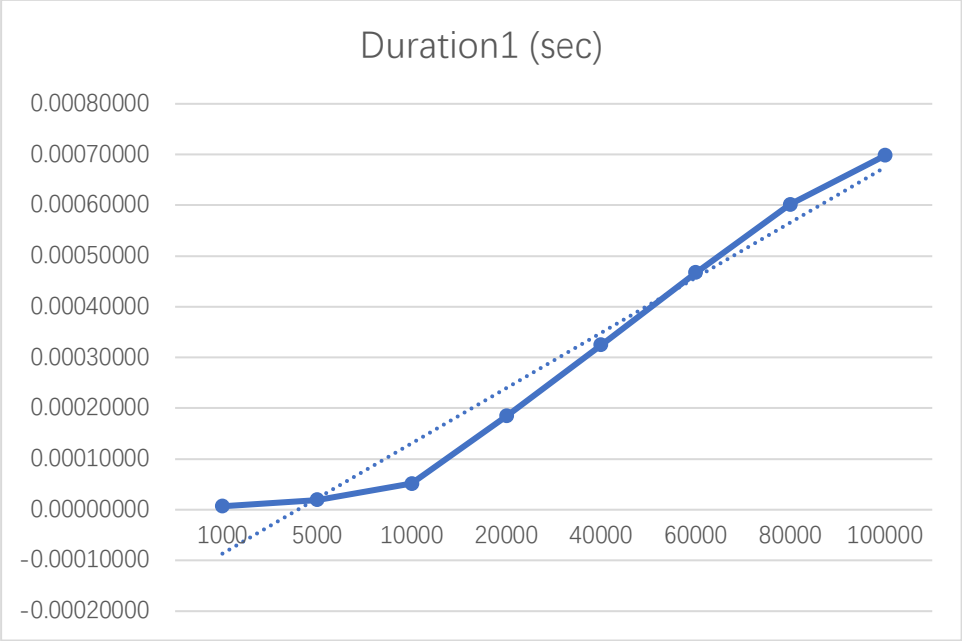
4.3.2 Space Complexity

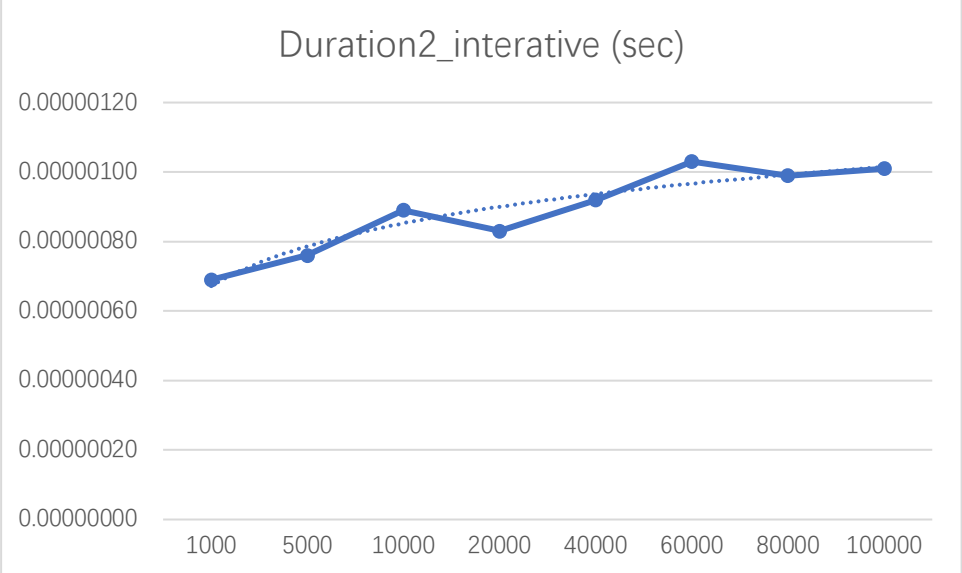
The space complexity of Algorithm 3 is higher due to the recursive calls. The maximum depth of the recursion tree is log(n), leading to a space complexity of O(log(n)) for the function call stack.

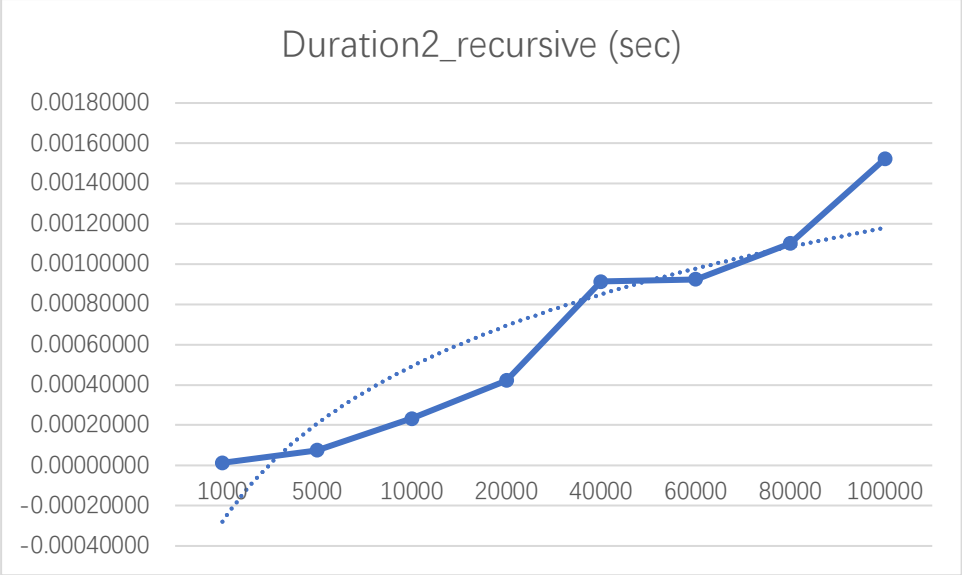
4.4 General Comments

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.