Lab1: Performance Measurement (POW)

Name: 姜雨童

Date: 2023-10-08

Chapter 1: Introduction

A power function is a common mathematical operation to compute the exponential power of a number. In computer science, it is very important to implement an efficient power function algorithm, because it has a wide range of applications in many fields, such as numerical computation, cryptography, etc.

Now 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, $X^N = X^{(N-1)/2} \times X^{(N-1)/2} \times X$. (Figure 2.11 in textbook gives the recursive version of this algorithm.)

And in this project, the **tasks** are:

(1) Implement Algorithm 1 and an iterative version of Algorithm 2;

(2) Analyze the complexities of the two algorithms;

(3) Measure and compare the performances of Algorithm 1 and the iterative and recursive implementations of Algorithm 2 for X=1.0001 and N = 1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000.

(4) The performances of the three functions must be plotted in the same N–run_time coordinate system for illustration.

Chapter 2: Algorithm Specification

My program can be divided into the main program and four functions, three of which involve different algorithms required for this lab, and the remaining function realizes the judgment of whether it is even. (The function algorithm to determine whether it is an even number is simple, and it is omitted here.)

The **main program** consists of setting the values of x and n, calling and running three functions (using different algorithms and each function running K times), and the testing program. Its pseudo-code looks like this:

```
function main():
```

```
1.      K = some_value
2.      n_values = [1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000]
3.      x = 1.0001
```

```
4.
5.      print("k =", K)
6.
7.      for n in n_values:
8.          print("n =", n)
```

```
        # Run function 1: 1-multiplication
```

```
9.          start = clock()
10.         for j in range(K):
11.             pow = func1_multiplication(x, n)
12.         stop = clock()
13.         total_time = (stop - start) / CLOCK_PER_SEC
14.         duration = total_time / K
15.         print("1-multiplication: Ticks =", stop - start, ", Total Time =",
    total_time, "seconds, Duration =", duration, "seconds.")
```

```
        # Run function 2: 2-recursion
```

```
16.         start = clock()
17.         for j in range(K):
18.             pow = func2_recursion(x, n)
19.         stop = clock()
20.         total_time = (stop - start) / CLOCK_PER_SEC
21.         duration = total_time / K
22.         print("2-recursion     : Ticks =", stop - start, ", Total Time =", total_time,
    "seconds, Duration =", duration, "seconds.")
```

```
        # Run function 3: 3-iteration
```

```
23.         start = clock()
24.         for j in range(K):
25.             pow = func3_iteration(x, n)
26.         stop = clock()
27.         total_time = (stop - start) / CLOCK_PER_SEC
28.         duration = total_time / K
29.         print("3-iteration     : Ticks =", stop - start, ", Total Time =", total_time,
    "seconds, Duration =", duration, "seconds.")
30.
31.     return 0
```

The first function uses **algorithm 1** using n-1 multiplications), and

its pseudo-code looks like this:

```
function func1_multiplication(x, n):
```

```
1.      pow = x
2.      for i from 1 to n – 1:    # Repeat n-1 times
3.          pow = pow * x
4.      return pow
```

The second function uses algorithm the **recursive version of algorithm 2** (performing repeated squaring operations until recursion is over), and its pseudo-code looks like this:

```
1. function func2_recursion(x, n):
```

```
2.     if n == 0:
3.         return 1.0
4.     if n == 1:
5.         return x
6.     if is_even(n):  # Check if n is even
7.         temp = func2_recursion(x * x, n / 2)
8.         return temp * temp
9.     else:  # n is odd
10.        temp = func2_recursion(x * x, (n - 1) / 2)
11.        return x * temp * temp
```

The third function uses algorithm the **iterative version of algorithm 2** (using division and remainder to keep track of whether n is odd or even when iteration occurs and performing repeated squaring operations.), and its pseudo-code looks like this:

```
1. function func3_iteration(x, n):
```

```
2.     counter = 0                      # Record how many times the program should run
3.     remainder = [0] * 100             # Record each time current n is even or odd
4.     pow = x
5.
6.     if n == 0:
7.         return 1.0
8.
9.     while n != 1:
10.        counter += 1
11.        remainder[counter] = n % 2     # Increment counter and record the remainder
12.        n //= 2
13.
14.    while counter != 0:
```

```
        if remainder[counter] == 1:
            pow = pow * pow * x         # Current n is odd
        else:
            pow = pow * pow             # Current n is even
        counter -= 1

    return pow
```

Chapter 3: Testing Results

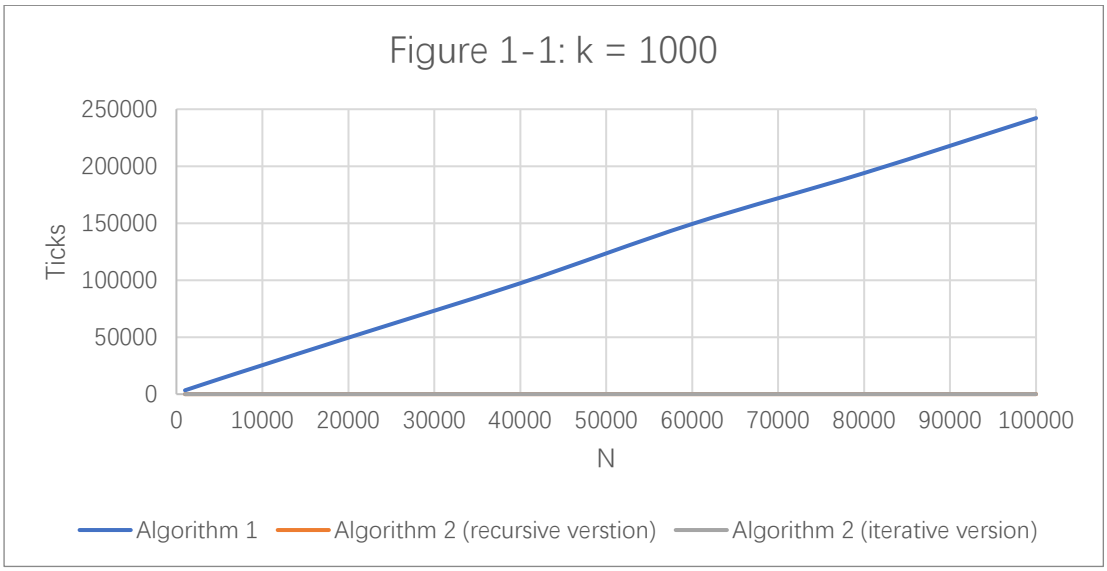
This case **is to** measure and compare the performances of Algorithm 1 and the iterative and recursive implementations of Algorithm 2 for X=1.0001 and N = 1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000. At the same time, we should compare their time complexity and space complexity, and compare them with the actual performance.

The **expected result** is that the running time of algorithm 1 should be much greater than the running time of the two versions of algorithm 2 (because the time complexity of algorithm 1 is much greater than that of algorithm 2). The running time of the two versions corresponding to algorithm 2 is not much different. In addition, when K (the number of times the function is repeatedly run, in order to reduce the error that becomes significant because the running time is too short) is multiplied, the running time of the corresponding function (Duration) should also be roughly multiplied by the corresponding multiple.

The **actual behavior of program** is that the running time of algorithm 1 is much greater than the running time of the two versions of algorithm 2, which is not much different. And when K is multiplied, the running time of the corresponding function is roughly multiplied by the same multiple (10 times). The tables and figures have been posted at the end of this chapter of the report.

Analysis of tables and figures shows that the program behavior is basically in line with expectations

Table 1: the actual data of project (k = 1000)					
\	N	1000	5000	10000	20000
Algorithm 1	Iterations (K)	1000			
	Ticks	3388	13319	25442	49568
	Total Time (sec)	0.003388	0.013319	0.025442	0.049568
	Duration (sec)	0.0000033880	0.0000133190	0.0000254420	0.0000495680
Algorithm 2 (recursive verstion)	Iterations (K)	1000			
	Ticks	40	52	53	55
	Total Time (sec)	0.000040	0.000052	0.000053	0.000055
	Duration (sec)	0.0000000400	0.0000000520	0.0000000530	0.0000000550
Algorithm 2 (iterative version)	Iterations (K)	1000			
	Ticks	35	54	46	48
	Total Time (sec)	0.000035	0.000054	0.000046	0.000048
	Duration (sec)	0.0000000350	0.0000000540	0.0000000460	0.0000000480
\	N	40000	60000	80000	100000
Algorithm 1	Iterations (K)	1000			
	Ticks	97332	149338	194004	242173
	Total Time (sec)	0.097332	0.149338	0.194004	0.242173
	Duration (sec)	0.0000973320	0.0001493380	0.0001940040	0.0002421730
Algorithm 2 (recursive verstion)	Iterations (K)	1000			
	Ticks	60	68	65	66
	Total Time (sec)	0.000060	0.000068	0.000065	0.000066
	Duration (sec)	0.0000000600	0.0000000680	0.0000000650	0.0000000660
Algorithm 2 (iterative version)	Iterations (K)	1000			
	Ticks	53	53	56	61
	Total Time (sec)	0.000053	0.000053	0.000056	0.000061
	Duration (sec)	0.0000000530	0.0000000530	0.0000000560	0.0000000610



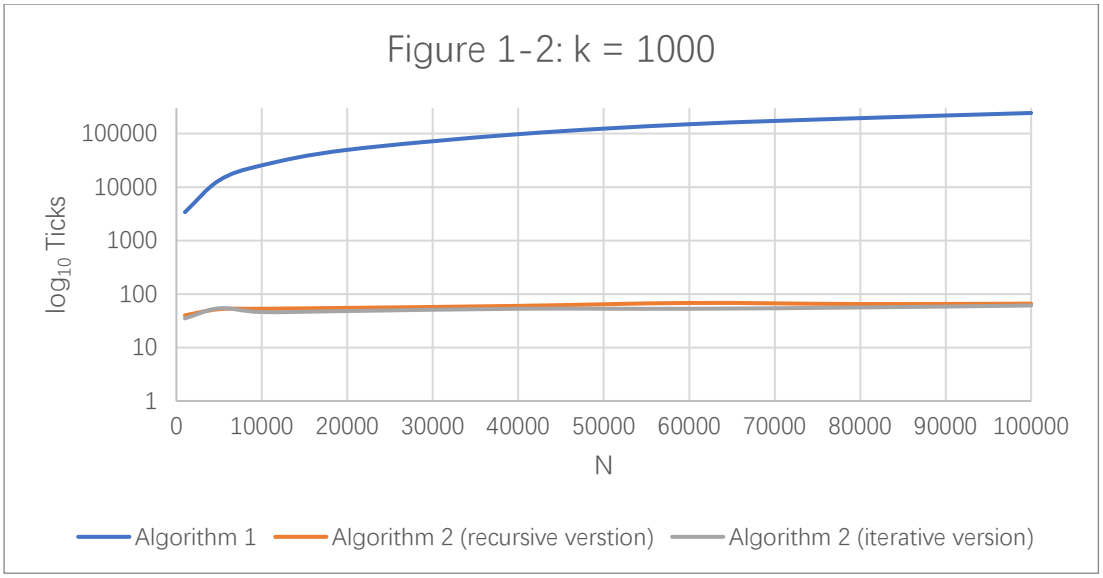
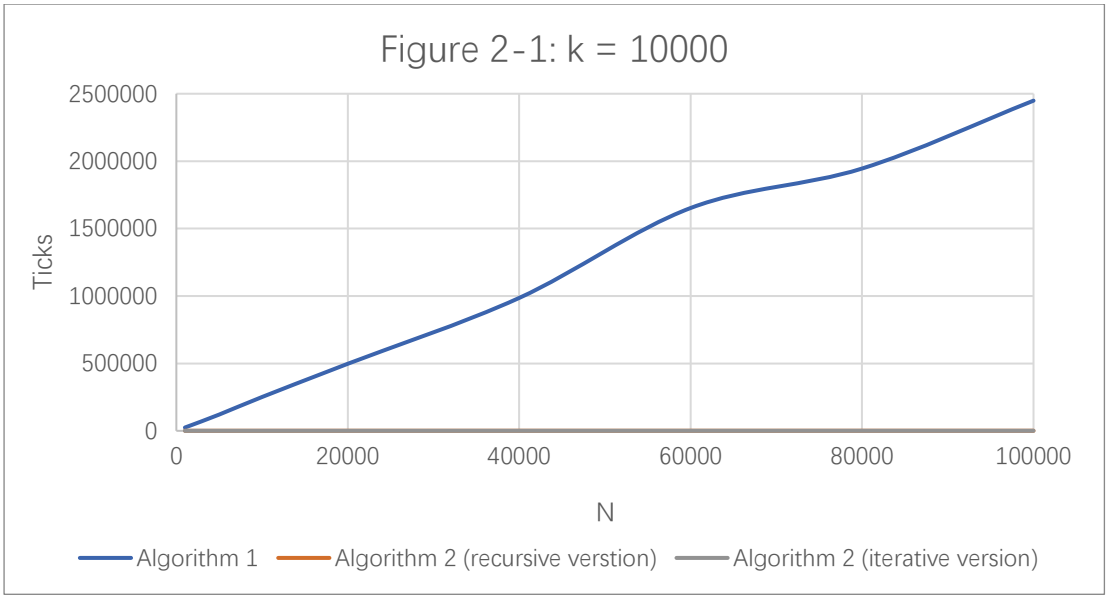
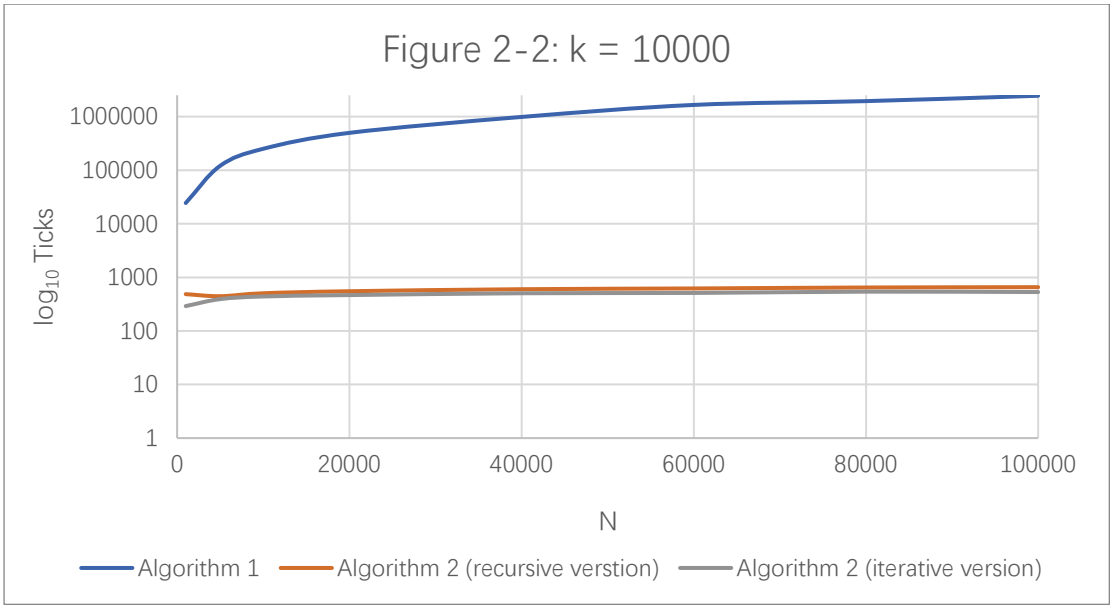


Table 2: the actual data of project (k = 10000)					
\	N	1000	5000	10000	20000
Algorithm 1	Iterations (K)	10000			
	Ticks	24409	120763	250948	497349
	Total Time (sec)	0.024409	0.120763	0.250948	0.497349
	Duration (sec)	0.0000024409	0.0000120763	0.0000250948	0.0000497349
Algorithm 2 (recursive verstion)	Iterations (K)	10000			
	Ticks	486	445	505	551
	Total Time (sec)	0.000486	0.000445	0.000505	0.000551
	Duration (sec)	0.0000000486	0.0000000445	0.0000000505	0.0000000551
Algorithm 2 (iterative version)	Iterations (K)	10000			
	Ticks	291	393	440	467
	Total Time (sec)	0.000291	0.000393	0.000440	0.000467
	Duration (sec)	0.0000000291	0.0000000393	0.0000000440	0.0000000467
\	N	40000	60000	80000	100000
Algorithm 1	Iterations (K)	10000			
	Ticks	985708	1652618	1945334	2448607
	Total Time (sec)	0.985708	1.652618	1.945334	2.448607
	Duration (sec)	0.0000985708	0.0001652618	0.0001945334	0.0002448607
Algorithm 2 (recursive verstion)	Iterations (K)	10000			
	Ticks	601	621	647	656
	Total Time (sec)	0.000601	0.000621	0.000647	0.000656
	Duration (sec)	0.0000000601	0.0000000621	0.0000000647	0.0000000656
Algorithm 2 (iterative version)	Iterations (K)	10000			
	Ticks	505	512	541	532
	Total Time (sec)	0.000505	0.000512	0.000541	0.000532
	Duration (sec)	0.0000000505	0.0000000512	0.0000000541	0.0000000532





Chapter 4: Analysis and Comments:

Analysis of the time and space complexities of the algorithms:

- Algorithm 1: The loop runs from 1 to (n-1), so the **time complexity is O(n)**. The space used is constant (The space used when entering data is not counted), so the **space complexity is O(1)**.
- Algorithm 2 (recursive version): In each recursive call, n is divided by 2 (when n is even), so the **time complexity is O(log n).** The **space complexity** is determined by the maximum depth of the recursion, and **is O(log n).**

- Algorithm 2 (iterative version): Similar to recursion version, the **time complexity is O(log n)**. The space used is constant (a fixed-size array and some variables), so the **space complexity is O(1)**.

Comments on further possible improvements:

The recursive and iterative versions of Algorithm 2 have a significantly better time complexity than Algorithm 1 for large values of n, as they use a divide-and-conquer approach that reduces the number of multiplications required.

However, there exists something to improve. For iterative version of algorithm 2, we can use bitwise operations (such as n & 1) to check if n is even or odd, instead of using division and remainder operations.

Appendix: Source Code (in C)

```c
#include <stdio.h>
#include <math.h>
#include <time.h>
#define K 1000       /* the program runs K times */
clock_t start, stop;
double duration, total_time;

/* function declaration: */
double func1_multiplication(double x, int n);
/* algorithm 1: use N - 1 multiplications */
double func2_recursion(double x, int n);
/* algorithm 2-1: recursive version of algorithm 2, from Figure 2.11 in textbook */
double func3_iteration(double x, int n);
/* algorithm 2-2: iterative version of algorithm 2 */
int even(int n);
/* function: judge whether n is an even, if it's return 1, else return 0 */

int main(void)
{
    int i, j;
    int n[8] = {1000,5000,10000,20000,40000,60000,80000,100000};
    /* different n for test (total eight values of n)*/
    double x = 1.0001;      /* base number */
```

```
    double pow;             /* the result */

    printf("k = %d\n",K);   /* print the value of K */
    for(i = 0;i < 8;i++)    /* change the value of n */
    {
        /* run function 1 */
        start = clock();
        /* records the ticks at the beginning of the function call */
        for(j = 0;j < K;j++)/* repeat the function calls for K times */
            pow = func1_multiplication(x, n[i]);
        stop = clock();
        /* records the ticks at the end of the function call */
        total_time = ((double)(stop - start))/CLOCKS_PER_SEC;
        /* CLOCK_PER_SEC is a built-in constant = ticks per second */
        duration = total_time / K;
        /* duration is the actual running time of this algorithm */
        printf("n = %6d, 1-multiplication: Ticks = %5d, Total Time = %2f seconds,
Duration = %.10f seconds.\n",n[i],(int)(stop-start),total_time,duration);
        /* print the kind of algorithm, ticks, total time and duration */

        /* run function 2: recursive version */
        start = clock();
        for(j = 0;j < K;j++)
            pow = func2_recursion(x, n[i]);
        stop = clock();
        total_time = ((double)(stop - start))/CLOCKS_PER_SEC;
        duration = total_time / K;
        printf("n = %6d, 2-recursion     : Ticks = %5d, Total Time = %2f seconds,
Duration = %.10f seconds.\n",n[i],(int)(stop-start),total_time,duration);

        /* run function 3: iterative version */
        start = clock();
        for(j = 0;j < K;j++)
            pow = func3_iteration(x, n[i]);
        stop = clock();
        total_time = ((double)(stop - start))/CLOCKS_PER_SEC;
        duration = total_time / K;
        printf("n = %6d, 3-iteration     : Ticks = %5d, Total Time = %2f seconds,
Duration = %.10f seconds.\n",n[i],(int)(stop-start),total_time,duration);
    }
    return 0;
}
```

```
63.
```

```
64. /* algorithm 1: use N - 1 multiplications */
```

```
65. double func1_multiplication(double x, int n)
66. {
67.     int i;
68.     double pow = x;
69.     for(i = 1;i < n;i++)    /* n-1 times */
70.         pow *= x;
71.     return pow;
72. }
73.
```

```
74. /* algorithm 2-1: recursive version of algorithm 2, from Figure 2.11 in textbook*/
```

```
75. double func2_recursion(double x, int n)
76. {
77.     if(n == 0)
78.         return 1.0;
79.     if(n == 1)
80.         return x;
81.     if(even(n))
82.         return ( func2_recursion(x*x, n/2 ));       /* the case that n is an even */
83.     else
84.         return ( func2_recursion(x*x, n/2) * x );   /* the case that n is an odd */
85. }
86.
```

```
87. /* algorithm 2-2: iterative version of algorithm 2 */
```

```
88. double func3_iteration(double x, int n)
89. {
90.     int counter = 0;        /* record how many times the program should run */
91.     int remainder[100];     /* record each time current n is even or odd */
92.     double pow = x;
93.
94.     if(n == 0)
95.         return 1.0;
96.     while(n != 1)
97.     {
98.         remainder[++counter] = n % 2;   /* counter increments and record the
remainder */
99.         n /= 2;
100.    }
101.    while(counter != 0)
102.    {
103.        if(remainder[counter])
```

```
104.             pow = pow * pow * x;       /* current n is an odd */
105.         else
106.             pow = pow * pow;           /* current n is an even */
107.         counter--;
108.     }
109.     return pow;
110. }
111.
```

```
112. /* function: judge whether n is an even, if it's return 1, else return 0 */
```

```
113. int even(int n)
114. {
115.     if(n % 2 == 0) return 1;
116.     return 0;
117. }
```

Declaration

I hereby declare that all the work done in this project titled " Lab1: Performance Measurement (POW)" is of my independent effort.