Performance Measurement (POW)

Date：2019-09-24

Chapter 1. Introduction

The problem shows two different algorithms to calculate X^N :

- Algorithm 1 : multiply X for (N-1) times straightly.
- Algorithm 2 : divide X^N into $X^{\frac{N}{2}} * X^{\frac{N}{2}}$ (if N is even) or $X^{\frac{N-1}{2}} * X^{\frac{N-1}{2}} * X$ (if N is odd).

And there are two different ways to implement algorithm 2. One is the recursive method, and the other is the iterative method. Our job is to use C to make a program to test the efficiency of these two algorithms.

Chapter 2. Algorithm Specification

1. Algorithm 1:

- Key function : mul(x,n)
- Pseudo-code:

```
Func mul( x , n )
    ans=1
    for i=0 upto n-1
        ans=ans*x
    return ans
```

2. Algorithm 2(recursive):

- Key function : pow(x,n)
- Pseudo-code:

```
Func pow( x , n )
    if n==0
        return 1
    if n==1
        return x
    if n%2==0
        return pow(x*x, n/2)
    else
        return pow(x*x, n/2)*x
```

3. Algorithm 2 (iterative):

- Key function : ite(x,n)
- Pseudo-code:

```
Func ite( x , n )
    if n==0
        return 1
    i=0
    m=n
    ans=x
    while m>0
        s[i] = m
        m=m/2
        i=i+1
    i=i-1
    while i>0
        if s[i-1]==2*s[i]
```

```
            ans=ans*ans
        else
            ans=ans*ans*x
        i=i-1
    return ans
```

Chapter 3. Testing Results

The testing results are as follows in the table.

	N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1	Iterations (K)	500000	500000	200000	200000	200000	100000	100000	100000
	Ticks	1874	8680	7842	15134	27823	20413	26573	33665
	Total time (sec)	1.874	8.68	7.842	15.134	27.823	20.413	26.573	33.665
	Duration (sec)	0.000003748	0.00001736	0.00003921	0.00007567	0.000139115	0.00020413	0.00026573	0.00033665
Algorithm 2 (iterative version)	Iterations (K)	500000	500000	200000	200000	200000	100000	100000	100000
	Ticks	33	42	19	20	21	11	11	11
	Total time (sec)	0.033	0.042	0.019	0.02	0.021	0.011	0.011	0.011
	Duration (sec)	0.000000066	0.000000084	0.000000095	0.0000001	0.000000105	0.00000011	0.00000011	0.00000011
Algorithm 2 (recursive version)	Iterations (K)	500000	500000	500000	200000	200000	100000	100000	100000
	Ticks	29	42	16	17	21	9	11	15
	Total time (sec)	0.029	0.042	0.016	0.017	0.021	0.009	0.011	0.015
	Duration (sec)	0.000000058	0.000000084	0.00000008	0.000000085	0.000000105	0.00000009	0.00000011	0.00000015

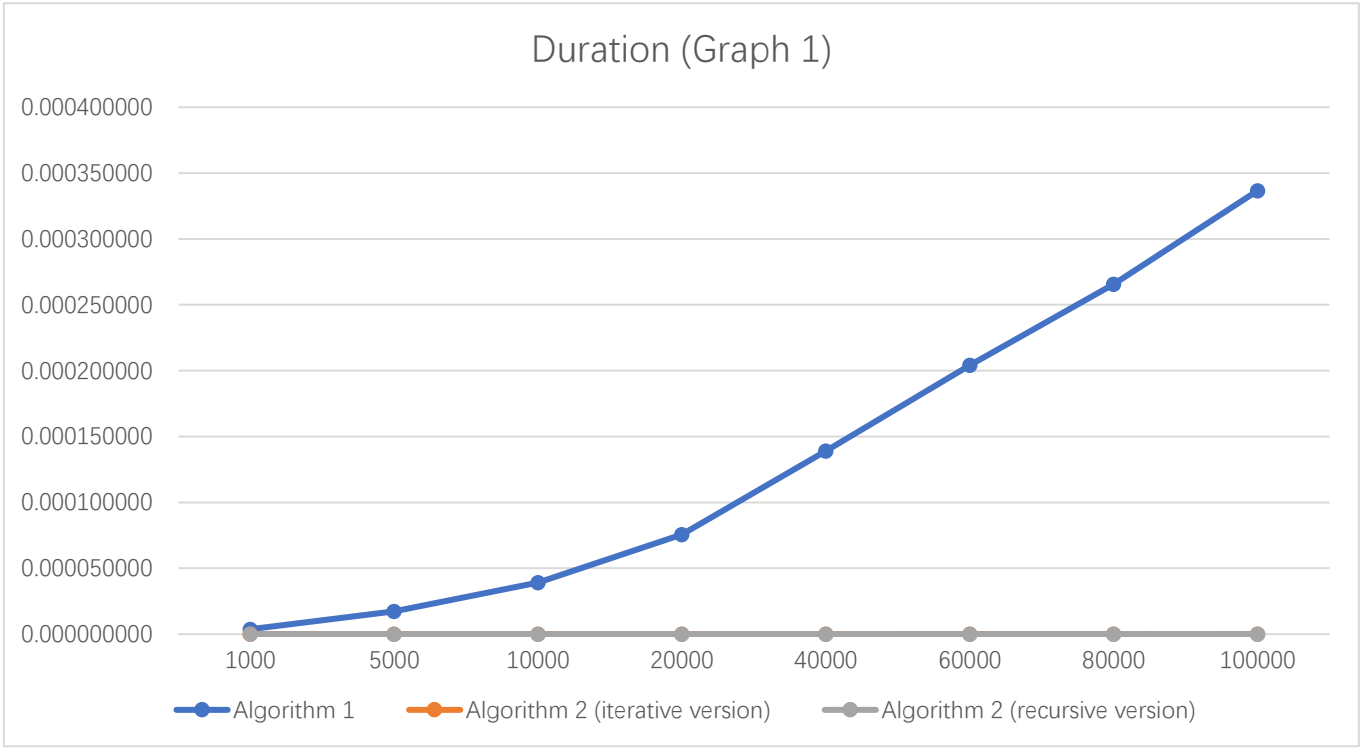
In the table, we list the performance of three different functions. The Ticks represent the running time of functions recorded by the program. The Total time is the total running time of the functions expressed in seconds, which is computed from the ticks. The duration the the running time of the functions to run once, which is computed by dividing the total time with the running times.

As for K, we use different K to cater to different N. When N is small, K must be large enough to reduce the error. When N is large, we need to reduce K so that the time won't be too long to tolerate.

As you can see from the table, algorithm 1 takes much more time than algorithm 2 respectively. And as the input N increases, the gap of time taken between the two algorithms augments.

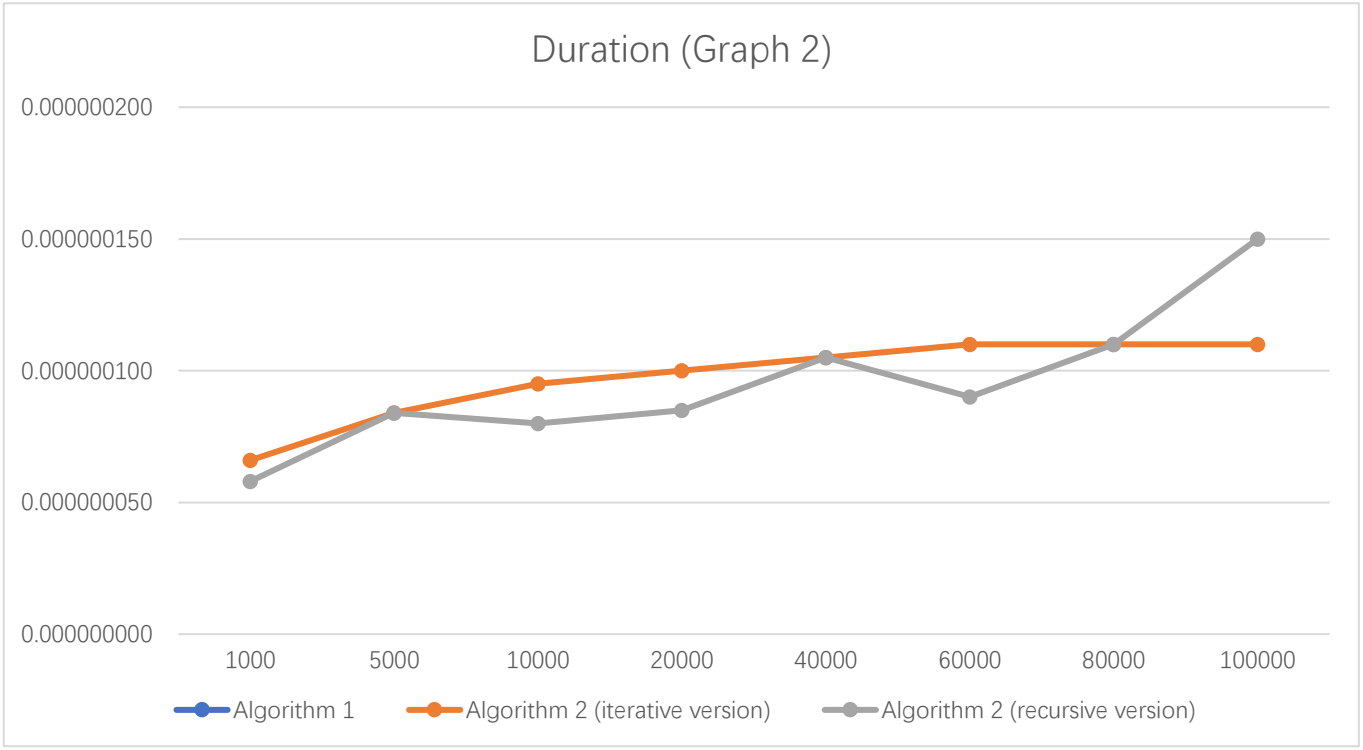
Chapter 4. Analysis and Comments

In order to make the data more intuitive, we can plot the duration of the three functions in a graph as follows.



By algorithm analysis we know that the time complexity of algorithm 1 is O(N) while the time complexity of algorithm 2 is O(log N). Consequently, algorithm1 grows faster. From the graph we can see that the duration of algorithm 1 grows linearly approximately as the input N grows. In other words, the duration of algorithm 1 is proportional to N. This is just the result and evidence that Algorithm 1 is O(N).

As for algorithm 2, its grows much slower. From the graph we can see that compares to algorithm 1, the duration of algorithm 2 stays very close to zero. In order to see the changes more clearly, we can change the coordinates.



The duration of the iterative and recursive version of algorithm 2 is almost the same, but still they have a little difference. We think this is related to the characteristics of C. The shape of iterative version's curve is just like the shape of a

logarithmic function. This is the result and evidence that the time complexity algorithm 2 is O(log N).

As for the space complexity, in algorithm 1 we don't need extra space to store intermedia results, thus the space complexity of algorithm 1 is O(1). In the recursive version of algorithm 2, we need to store the intermedia value in the stack each step, so the space complexity is O(log N). This is the same with the iterative version. We need an array to store the intermedia value, thus the space complexity is also O(log N).

From the analysis we know that algorithm 2 runs faster than algorithm 1. But on the other hand, algorithm 2 takes more space. Usually time complexity is more important than space complexity in practice, so algorithm 2 is the better one we can choose in practice.

As for further improvements, O(log N) is the best time complexity, so the binary algorithm is the best. No more improvement can be made.

Appendix：Source Code

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

clock_t start, stop;                    //clock_t is a built-in type for processor time(ticks)
double total_time, duration;            //records the run time(seconds) of a function

double mul(double x, int n)             //Implement Algorithm 1
{
    int i;
    double ans = 1;
    for (i = 0; i< n; i++) ans*= x;             //Use N-1 multiplications to get X^N
    return ans;
}

double pow(double x, int n)                     //Implement an recursive version of Algorithm 2
{
    if (n == 0) return 1;                       //Boundary condition
    if (n == 1) return x;                       //Boundary condition
    if (n%2 == 0)
        return pow(x*x, n/2);                   //If N is even, X^N = X^(N/2)*X^(N/2)
    else
        return pow(x*x, n/2)*x;                 //If N is odd, X^N = X^((N-1)/2)*X^((N-1)/2)*X
}

double ite(double x, int n)                     //Implement an iterative version of Algorithm 2
{
    if (n == 0) return 1;                       //Boundary condition
    int s[100], i = 0, m = n;
    double ans = x;
    while (m> 0)                                //S is an array that stores the remainder of n
                                                //successive division by 2 for final iteration
    {
        s[i] = m;
        m/= 2;
        i++;
```

```
    }
    i--;
    while (i> 0)                                //This is part of the iteration
    {
        if (s[i-1] == 2* s[i]){                 //It's equal to 'N is even'.
            ans = ans* ans;
        }
        else{                                   //It's equal to 'N is odd'.
            ans = ans* ans* x;
        }
        i--;                                    //With S, ans can go from X to X^N
    }
    return ans;
}

int main()
{
    int n, K;
    double x, ans;
    scanf("%lf %d %d", &x, &n, &K);             //Read in x, n, K according to the question

    start = clock();                            //records the ticks at the beginning of the function
call
    for (int i = 0; i < K; i++)
    {
        ans = mul(x, n);                        //Implement Algorithm 1
    }
    stop = clock();                             //records the ticks at the end of the function call
    printf("Algorithm 1:\n");
    printf("Ticks: %d\n", stop - start);
    total_time= ((double)(stop- start))/CLK_TCK;           //CLK_TCK is a built-in constant = ticks
                                                           //per second
    duration = total_time / K;
    printf("Total time: %.10lf\n",total_time);
    printf("Duration: %.10lf\n",duration);             //Output run time    T(N) = O(N)

    start = clock();                            //Same as above
    for (int i = 0; i < K; i++)
    {
        ans = pow(x, n);                        //Implement an recursive version of Algorithm 2
    }
    stop = clock();
    printf("Algorithm 2(recursive):\n");
    total_time= ((double)(stop- start))/CLK_TCK;
    duration = total_time / K;
    printf("Ticks: %d\n", stop - start);
    printf("Total time: %.10lf\n",total_time);
    printf("Duration: %.10lf\n",duration);         //T(N) = O(log(N))
```

```
    start = clock();
    for (int i = 0; i < K; i++)
    {
        ans = ite(x, n);                          //Implement an iterative version of Algorithm 2
    }
    stop = clock();
    printf("Algorithm 2(iterative):\n");
    total_time= ((double)(stop- start))/CLK_TCK;
    duration = total_time / K;
    printf("Ticks: %d\n", stop - start);
    printf("Total time: %.10lf\n",total_time);
    printf("Duration: %.10lf\n",duration);         //T(N) = O(log(N))

    system("pause");
    return 0;
}
```

Declaration:

We hereby declare that all the work done in this project titled "Performance Measurement (POW)" is our independent effort as a group.