Project 1: Performance Measurement

Zhang Chengyi

2016-10-06

Contents

1 Introduction

1.1 Concepts Description

1.1.1 Algorithm

Informally, an algorithm is any well-defined computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output. Thus an algorithm is a a set of rules that precisely defines a sequence of operations (which transform the input into the output). [**?**]

1.1.2 Algorithm Analysis

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires, which is usually the computing time. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process. In this project, our main target is to analyze the time complexity in terms of performance measurement.

1.1.3 Growth of Functions

The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. We use such symbols to describe the growth of functions of a certain algorithm:

DEFINITION: $T(n) = O(f(n))$ if there are constants c and n_0 such that $T(n) \leqslant cf(n)$ when $n \geqslant n_0$.
DEFINITION: $T(n) = \Omega(g(n))$ if there are constants c and n_0 such that $T(n) \geqslant cg(n)$ when $n \geqslant n_0$.
DEFINITION: $T(n) = \Theta(h(n))$ if and only if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$.
DEFINITION: $T(n) = o(p(n))$ if $T(n) = O(p(n))$ and $T(n) \neq \Theta(p(n))$.

1.2 Project Description

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, $X^N = X^{N/2} * X^{N/2}$; and if N is odd, $X^N = X^{(N-1)/2} * X^{(N-1)/2} * X$.

Figure 2.11 in your textbook gives the recursive version of this algorithm.
Our 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. As a result, We implemented all three algorithms, and a auto-test program, then integrated them together into a single file. Then we selected a series of K as the iteration number and got a complete test result. The we analyzed the result and drew some conclusions according to it. Finally, we finished this experimental report.

2 Algorithm Specification

Our algorithms are used for doing exponentiation, Algorithm 1 is to use N − 1 multiplications, and Algorithm 2 works in the strategy of 'Divide and Conquer'.

2.1 Algorithm1 : Brute Force

Algorithm 1 is just simply using N - 1 times multiplications.

2.1.1 Implementation in Pseudo Code

Code 1: Brute Force in Pseudo Code

```
function exp(x: base number, n: exponent number)
    sum := 1;
    for i from 1 to n:
        sum := sum * x;
    return sum;
```

2.1.2 Implementation in C

Code 2: Brute Force in C

```c
double algorithm1(double x, int n) {
    double sum = 1;
    for( int i = n; i > 0; i--)
        sum *= x; // Simply multiply N times.
    return sum;
}
```

2.2 Algorithm2 : Exponentiation by Squaring

2.2.1 General

Exponentiating by squaring, which is a general method for fast computation of large positive integer powers of a number, is based on the observation that, for a positive integer n, we have: $x^n = \begin{cases} x\,(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$

Our implementation restrict the exponential $n \in N^*$

2.2.2 Recursive Implementation in Pseudo Code

Code 3: Exponentiation by squaring (Recursive) in Pseudo Code

```
function exp(x: base number, n: exponent number)
    if n = 0 then return 1;
    else if n = 1 then return x;
    else if n is even then return exp(x * x, n / 2);
    else if n is odd then return x * exp(x * x, (n − 1) / 2).
```

2.2.3 Recursive Implementation in C

Code 4: Exponentiation by squaring (Recursive) in C

```c
double algorithm2_recur(double x, int n) {
    if(n == 0)
        return 1;
    if(n == 1)
        return x;
    else if(n%2 == 0){
        double ret = algorithm2_recur(x, n/2);  //Cache the result
        return ret * ret;
    }else{
        double ret = algorithm2_recur(x, (n−1)/2); //Cache the result
        return ret * ret * x;
    }
}
```

2.2.4 Iterative Implementation in Pseudo Code

Code 5: Exponentiation by squaring (Iterative) in Pseudo Code

```
function exp(x: base number, n: exponent number)
    if n = 0 then return 1;
    result := 1;
    while n > 1 do
      if n is odd then
        result := x * result;
      n := n / 2;
      x := x * x;
    return x * result
```

2.2.5 Iterative Implementation in C

Code 6: Exponentiation by squaring (Iterative) in C

```c
double algorithm2_iter(double x, int n) {
    double result = 1.0;
    while(n != 0){
        if(n % 2 != 0) result *= x; //  X ^ (2N + 1) = X * (X ^ 2N)
        x *= x;//X ^ (2N)   = (X ^ 2) ^ N
        n /= 2;
    }
    return result;
}
```

3 Testing Results

3.1 Result Table

	N	1000	5000	10000	20000
Algorithm1	Iteration(K)	300000	60000	30000	15000
	Ticks	1180	1187	1217	1200
	Total Time(sec)	1.18	1.187	1.217	1.2
	Duration(ms)	0.003933	0.019783	0.040567	0.08
Algorithm2 (iterative)	Iteration(K)	30000000	30000000	25000000	25000000
	Ticks	1225	1353	1217	1294
	Total Time(sec)	1.225	1.353	1.217	1.294
	Duration(ms)	0.000041	0.000045	0.000049	0.000052
Algorithm2 (recursive)	Iteration(K)	30000000	30000000	25000000	25000000
	Ticks	2479	3299	3085	3366
	Total Time(sec)	2.479	3.299	3.085	3.366
	Duration(ms)	0.000083	0.00011	0.000123	0.000135
Algorithm1	Iteration(K)	8000	5000	4000	3000
	Ticks	1279	1201	1273	1234
	Total Time(sec)	1.279	1.201	1.273	1.234
	Duration(ms)	0.159875	0.2402	0.31825	0.411333
Algorithm2 (iterative)	Iteration(K)	20000000	20000000	20000000	20000000
	Ticks	1118	1212	1178	1192
	Total Time(sec)	1.118	1.212	1.178	1.192
	Duration(ms)	0.000056	0.000061	0.000059	0.00006
Algorithm2 (recursive)	Iteration(K)	20000000	20000000	20000000	20000000
	Ticks	2744	2828	2908	2948
	Total Time(sec)	2.744	2.828	2.908	2.948
	Duration(ms)	0.000137	0.000141	0.000145	0.000147

Table 1: Test Result Table

3.2 Conclusion

In this test case we set a group of N to test the three algorithms' performance. We record the total time that the functions are called K times, then we caculate the duration. We expect result is that : In algorithm1 the Duration is direct radio funtion of N. In two algorithm2s the Duration is Exponential function of N. And so it is.

4 Analysis and Comments

4.1 Result Graph

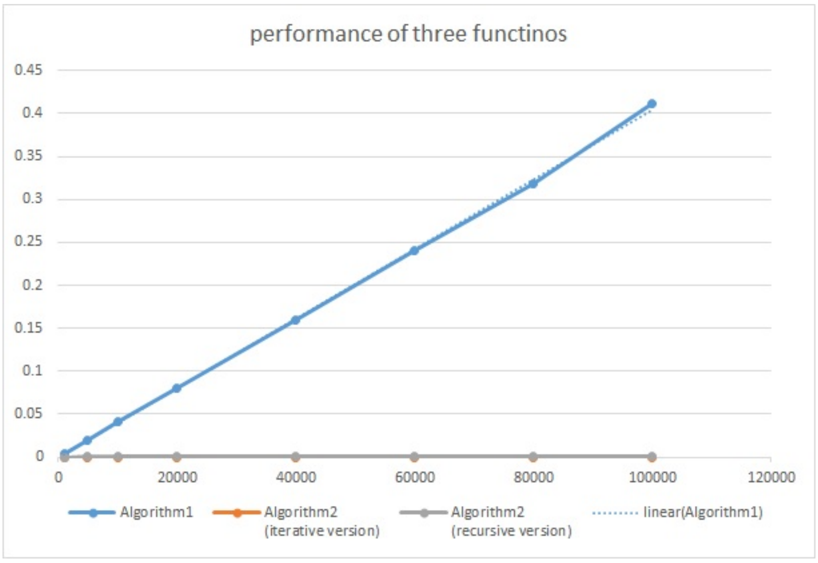

Figure 1: Performance of All Algorithms

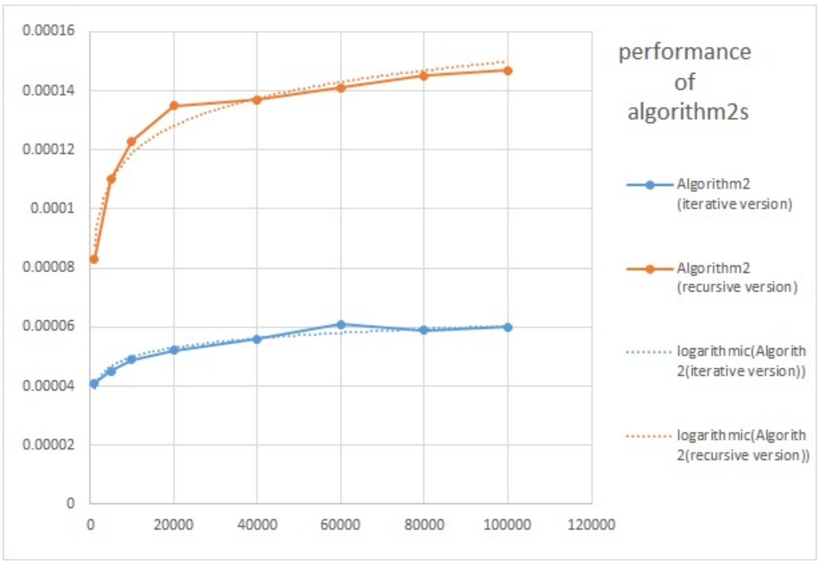

Figure 2: Performance of Algorithm2

We first use theoretical analysis to deduce the growth of function of a certain algorithm, and then do validation in terms of the implementation in C code. Finally, we make comparison between the result from analysis and the result from performance test, in order to make it be accurate.

4.2 Algorithm1 : Brute Force

4.2.1 Theoretical Analysis

Time Complexity Obviously, the growth of algorithm can be calculate as following:

$$T(n) = N - 1 \tag{1}$$

Space Complexity Space complexity is obviously $O(1)$.

4.2.2 C Code Analysis

Line	Operation	the Number of Unit Operation
1	Initialization	1
2	Increment of loop counting variable	2n+2
3	Multiplication	n
4	Return	1
Total		3n+4

Table 2: Brute Force

4.3 Algorithm2 : Exponentiation by Squaring(Recursive)

4.3.1 Theoretical Analysis

Time Complexity By means of the Master Theorem, the growth of algorithm can be calculated as following:

$$T(n) = T(n/2) + 1 \tag{2}$$

Due to $f(n) = \Theta(n^{log1})$, we get:

$$T(n) = O(log(n)) \tag{3}$$

Space Complexity For space complexity, the recursion is performed $O(log(n))$ times, so the space complexities is $O(log(n))$.

4.3.2 C Code Analysis

The C code is similar to the Pseudo Code, and the analysis of C code is similar to Pseudo Code, this part is omitted.

4.4 Algorithm2 : Exponentiation by Squaring(Iterative)

4.4.1 Theoretical Analysis

Time Complexity A brief analysis shows that such an algorithm uses $\lfloor \log_2 n \rfloor$ squaring and at most $\lfloor \log_2 n \rfloor$ multiplications, where $\lfloor\ \rfloor$ denotes the floor function. More precisely, the number of multiplications is one less than the number of ones present in the binary expansion of n. For n greater than about 4 this is computationally more efficient than naively multiplying the base with itself repeatedly.

Space Complexity Space complexity is obviously $O(1)$.

4.4.2 C Code Analysis

Line	Operation	the Number of Unit Operation
1	Initialization	1
2	Loop	at most log(n)
3	Calculation	at most 3 for each time
4	Return	1
Total		3 log(n) + 2

Table 3: Exponentiation by squaring(Iterative)

4.5 Comments

In general, the test result quite matches our expectation. The time that algorithm1 costs is proportional to N. Time that two algorithm2 cost are approximate to logarithmic function. And it's not surprising that algorithm1 is much slower than the two algorithm2.

The reason that algorithm1 is far slower than the other two algorithms is reflected in the analysis of time complexity. When N is big enough, $O(N)$ is, of course, much slower than $O(log(N))$.

The difference between the performances of two versions of algorithm2, we think, is caused by too much stack operation in the recursive version.The recursive version function needs to call itself many times, so the IP needs to record the function pointers and numerous push and pop are done.And that influences the speed.

From the results we notice that when N = 60000, 80000,100000, in algorithm2(the iterative version) the duration time decreases as N increases, which violates the common sense.Maybe we can run the program more times, or replace the K with a larger number, to avoid the coincidence.

5 Appendix

5.1 Source Code

See this subsection on the additional ‘project1.c‘ file.

5.2 Declaration

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

5.3 Duty Assignments

Programmer: Tan Qiye
Tester: Duan Fuzheng
Report Writer: Zhang Chengyi