Normal Performance Measurement Project 1

Date: 2021-10-02

Chapter 1: Introduction

In this project known for efficient exponentiation, we are supposed to compare two different algorithms when calculating X^N.

For the first one, just keep multiplying X for N-1 times, and the time complexity of this algorithm is O(N).

For the second one, trying consider N in two ways: even and odd. Provided that N is an odd number, $X^N=X^{(N-1)/2}\times X^{(N-1)/2}\times X$. If not, $X^N=X^{N/2}\times X^{N*2}$. A perfect chance for recursive algorithm.And its time complexity is O(logN).

So, by testing both algorithms we are able to find out which one is more efficient for solving this problem.

Chapter 2: Algorithm Specification

Algorithm 1:(Multiplying)

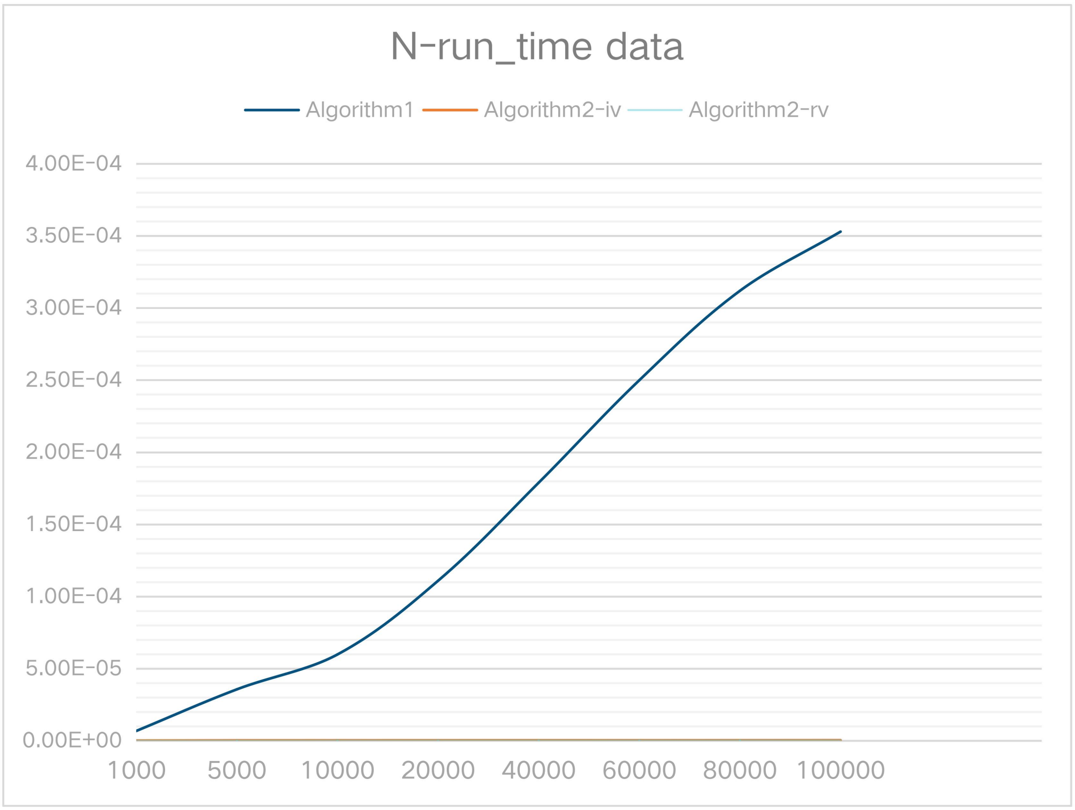
```
Function Multiplying(N:integer)
{
    var N,i;
    x=1.0001;
    S=1;
    for i:=0 to n do
        S=S*x;
    Output S;
}
```

Algorithm 2:(Iteration)

```
Function Pow(X, N)
{
    S=1;
    while(N>0){
        if(N%2==1){
            S*=X;
        }
        X*=X;
        N=N/2;
    }
    return S;
}
```

Chapter 3: Testing Results

	N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1	**Iterations(K)**	100	100	100	100	100	100	100	100
	Ticks	685	3571	5994	11109	17874	24955	31217	35308
	Total time(sec)	6.85E-04	3.57E-03	5.99E-03	1.11E-02	1.79E-02	2.50E-02	3.12E-02	3.53E-02
	Duration(sec)	6.85E-06	3.57E-05	5.99E-05	1.11E-04	1.79E-04	2.50E-04	3.12E-04	3.53E-04
Algorithm 2 (iterative version)	**Iterations(K)**	1000	1000	1000	1000	1000	1000	1000	1000
	Ticks	191	332	364	385	412	414	425	461
	Total time(sec)	1.91E-04	3.32E-04	3.64E-04	3.85E-04	4.12E-04	4.14E-04	4.25E-04	4.61E-04
	Duration(sec)	1.91E-07	3.32E-07	3.64E-07	3.85E-07	4.12E-07	4.14E-07	4.25E-07	4.61E-07
Algorithm 3 (recursive version)	**Iterations(K)**	10000	10000	10000	10000	10000	10000	10000	10000
	Ticks	1568	2412	2976	3205	3372	3766	3991	4097
	Total time(sec)	1.57E-03	2.41E-03	2.98E-03	3.21E-03	3.37E-03	3.77E-03	3.99E-03	4.10E-03
	Duration(sec)	1.57E-07	2.41E-07	2.98E-07	3.21E-07	3.37E-07	3.77E-07	3.99E-07	4.10E-07



By collecting all data and making charts, the 2 algorithms(including 3 examples) and their efficiency are thoroughly explained respectively.

Both algorithms share the same **purpose**: calculating X^N . When doing calculation in a small range of numbers, choosing better algorithms is not a vital problem. As for particularly large data, however, a wiser algorithm can really save a lot of time. By comparing above data I expect to **find a more useful algorithm** to solve the problem, and I suppose that the latter one is better.

Both (or all three) programs run well. For the first algorithm, just timing X for N-1 times. For the second one, which is based on a method similar to binary search, if N is an odd number, $X^N=X^{(N-1)/2}\times X^{(N-1)/2}\times X$. If not, $X^N=X^{N/2}\times X^{N*2}$. In an iterative way, X gets multiplying repeatedly while N reduces by half. In an recursive way, follow the odd and even rules and the function is developed. Meanwhile, to get a more precise duration of each program, every function has been repeated for K times(from 100 to 10000, depended on the selected algorithm), then doing division to get an average running time.

In this program, where we get an N from 1000 to 100000, the difference is huge. For algorithm 1(doing X=1.0001 self-multiplying for N-1 times), clearly the run time of the procedure surges as N increasing. Algorithm 2, however, is much more faster than algorithm one, no matter the iterative version or recursive version.

Actually when testing my codes I've confronted with **some problems**, which have already **gotten passed**. Firstly, different data types matter a lot when a program is running. At first I chose too limited data, and the result showed no difference when increasing numbers. Then I referred to the book, corrected my code and did again, problem solved. It also proved that a too small K also made bad results on the accuracy of test. Secondly, in the beginning I was really worried about my program: each time I couldn’t get the same number even when using the same N and K. Then I found out that indeed there existed some randomness based on background running programs in my PC. I tested many times to minimize the problem.

Overall, the testing results are gratifying.

Chapter 4: Analysis and Comments

	Space complexity	Time complexity
Algorithm 1	O(1)	O(N)
Algorithm 2-iterative	O(1)	O(logN)
Algorithm 2-recursive	O(logN)	O(logN)

Analysis for Algorithm 1(Multiplying for N-1 times)

【Space complexity】

When doing multiplying, the change of N won’t effect the space being used. So O(1).

【Time complexity】

```
{for（i=0;i<N;i++）
    Function();}
```

The function will be run for N times. Therefore the time complexity is O(N).

Analysis for Algorithm 2(Iteration)

【Space complexity】

```
Function Pow(X, N)
{
        S=1;
        while(N>0){
                if(N%2==1){
                        S*=X;
                }
                X*=X;
                N=N/2;
        }
        return S;
}
```

It works like algorithm 1, and the space complexity is O(1).

【Time complexity】

To calculate the time complexity of an iterative algorithm:

$T(N)=T(N/2)*T(N/2)$, N is even
$T(N)=T(N-1)*N$, N is odd

$T(N)=T(N/2)*T(N/2)$;
......
$T(2)=T(1)*T(1)$;
k times Iteration
$T(N)=T(1)^{2k}$, so the time complexity is O(logN).

Actually we can see that algorithm 2-iterative version is developed on the recursive version. Of course, it has made some improvements. For the recursion, space complexity is sacrificed to some extent in order to limit the time complexity. However, the iterative algorithm can keep both in a reasonable range.

As for the further possible improvements, I deem that some changes can happen when X is not just a number. In fact there are so many examples when doing fast exponentiation, especially matrix. Maybe the algorithm in this project can be similarly used in matrix computation, such as the Fibonacci number.

Appendix: Source Code (in C)

In order to make sure the whole docx is easy to read and check, I just picked the function part of my code. Some redundant parts of code has been removed.

Algorithm 1

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

clock_t start,stop;
double duration;

int main(){
    int N,i,j;
    int K=100;//repeat
    double s=1;//variation
    scanf("%d",&N);//input N.
    start = clock();
    for(j=0;j<K;j++)//control the repeating times
        for(i=0;i<N;i++)
            s=s*1.0001;//doing exponentiation for N-1 times.
    stop = clock();//time record for the function
    duration =
((double)(stop-start))/CLOCKS_PER_SEC;//another version of
CLK_TCK
    printf("%ld\n%f",stop-start,duration);
    return 0;

}
```

Algorithm 2-iterative

```
#include <stdio.h>
#include <time.h>
clock_t start,stop;
double duration;
// the iterative version of algorithm 2.
typedef long long ll;
ll Pow(double x,int N){
    ll s=1;
    while(N>0){
        if(N%2==1){
            s*=x;
        }
        x*=x;//x self-multiplying.
        N=N/2;//N decreases by half
    }
    return s;
}
int main(){
    int N,K=1000,i;
    double s=1;

    scanf("%d",&N);
    start = clock();
    for(i=0;i<K;i++)//repeat K times.
        s=Pow(1.0001,N);
    stop = clock();
    duration = 
((double)(stop-start))/CLOCKS_PER_SEC;//another version of 
CLK_TCK.
    printf("%ld\n%e",stop-start,duration);//output.
    return 0;
}
```

Algorithm 2-recursive

```
#include <stdio.h>
#include <time.h>
clock_t start,stop;
double duration;
//the recursive version of algorithm 2.
double Pow(double x, int N)
{
    if(N==0)
        return 1;//ending part for recursion if N is even 
finally.
    if (N==1)
        return x;//ending part for recursion if N is odd 
finally.
    if (N%2==0)
        return Pow(x*x,N/2);//if N is even
    else
```

```
        return Pow(x*x, N/2)*x;//if N is odd
    }


int main(){
    int N,i,s;
    int K=10000;//repeat 10000 times
    scanf("%d",&N);
    start = clock();
    for (i=0; i<K; i++)
        s=Pow(1.0001,N);//function operating.
    stop = clock();
    duration =
((double)(stop-start))/CLOCKS_PER_SEC;//another version of
CLK_TCK.
    printf("%ld\n%f\n",stop-start,duration);
    return 0;
}
```

Declaration

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