Implementation and comparison of three exponentiation algorithms

Xu Shengze

Date: 2021-10-12

Chapter 1: Introduction

This question is to use different algorithms to complete the calculation of X^N. The topic requires us to implement algorithm 1 and an iterative version of algorithm 2, and analyze the time complexity of different algorithms. Considering the given value of X, change the value of N to test and compare the performance of different algorithms. In order to achieve this, we use the standard library `time.h` of `C`.

In order to overcome the problem that the function runs too fast and the accuracy is not enough, we can use the method of calling the function multiple times to make the result more accurate. This requires that we first need to calculate the appropriate number of calls K, and then divide the total time by K to get the time of a single run of the function. The topic requires us to use the given data to obtain the test results of each algorithm, and to draw the corresponding function image to analyze and evaluate the different algorithms.

Chapter 2: Algorithm Specification

2.1 Algorithm 1

The idea of this algorithm is very simple, and it is also the simplest and direct method. It is easy to write the program by directly multiplying $N-1$ times. The main pseudocode of the function is as follows.

```
result ← 1
for i ← 0 to n
    do result ← result*x
```

This function uses the `for` loop to multiply $N-1$ times to get the result. Although this method is very easy to implement, it is obviously not efficient, and the time complexity is obviously $O(N)$.

2.2 Algorithm 2

2.2.1 Recursive function

Since the above algorithm is very inefficient when N is very large, we can use dichotomy to speed up the calculation of X^N. If N is an even number, then the calculation of X^N is split into $X^{N/2} \times X^{N/2}$; if N is an odd number, then the calculation of X^N is split into $X^{(N-1)/2} \times X^{(N-1)/2} \times X$.

There are two ways to implement this algorithm, one is recursion and the other is iteration. The recursive form of this algorithm has been given in the book, but we still give the pseudocode of the recursive function here.

```
if n=0
    then return 1
if n=1
    then return x
if n%2=0
    then return Recursive(x*x,n/2)
    else return Recursive(x*x,(n-1)/2)*x
```

If it is a relatively simple situation, that is, when the index is 0 or 1, the function directly returns the calculated value. If the exponent is the remaining number, in order to return the correct value without increasing the time complexity, we call the recursive function itself in a reasonable way and obtain the return value through corresponding calculations.

The complete code of the recursive function will be given in the appendix. Obviously, the maximum number of multiplications required here is $2log(N)$, and the time complexity is $log(N)$.

2.2.2 Iterative function

In order to reduce repeated calculations, we use a non-recursive form, namely iteration, to implement the calculation process of the dichotomy. The following is the pseudocode of the iterative algorithm.

```
result ← 1
if n=0
    then return 1
while n>0
    do if n%2=1
        then result ← result*x
    x ← x*x
    n ← n/2
```

This function is slightly more complicated than the recursive method, and the loop ends when the exponent is equal to 0. The loop condition of this function is `n>0`, and each loop will go through $n/2$. Obviously, the time complexity of this iterative function is still $O(logN)$, which can be multiplied by $O(LogN)$ realizes the calculation of X^N within the number of times, and reduces repeated calculations compared to recursion.

This iterative method is related to the binary expression of the exponent, and is equivalent to the effect of the program represented by the following pseudo-code, with subtle differences in the calculation process.

```
while n>0
    do if n%2=0
        then x ← x*x
             n ← n/2
        else result ← result*x
             n ← n-1
```

2.3 Main program

In the main program, we call each function written above, and use the standard library `time.h` of `C` to get the time required for the function to run, and calculate the values that need to be filled in the form accordingly. The most critical part of the main program is to determine the value of K in each case, so as to obtain a reasonable and more accurate time required to call a written function. It is necessary to satisfy that K is at least greater than 10. This part The pseudocode is as follows.

```
for i ← 0 to 8
    do K ← 1
    repeat
    K ← K*10
    start ← clock()
    for j ← 0 to K
        do result ← function()
    stop ← clock()
    until stop-start<10
```

It should be noted that the K used in the following test is the minimum value that satisfies the conditions, but in order to make the data more accurate, the value of K can be increased appropriately.

Chapter 3: Testing Results

3.1 Operation result

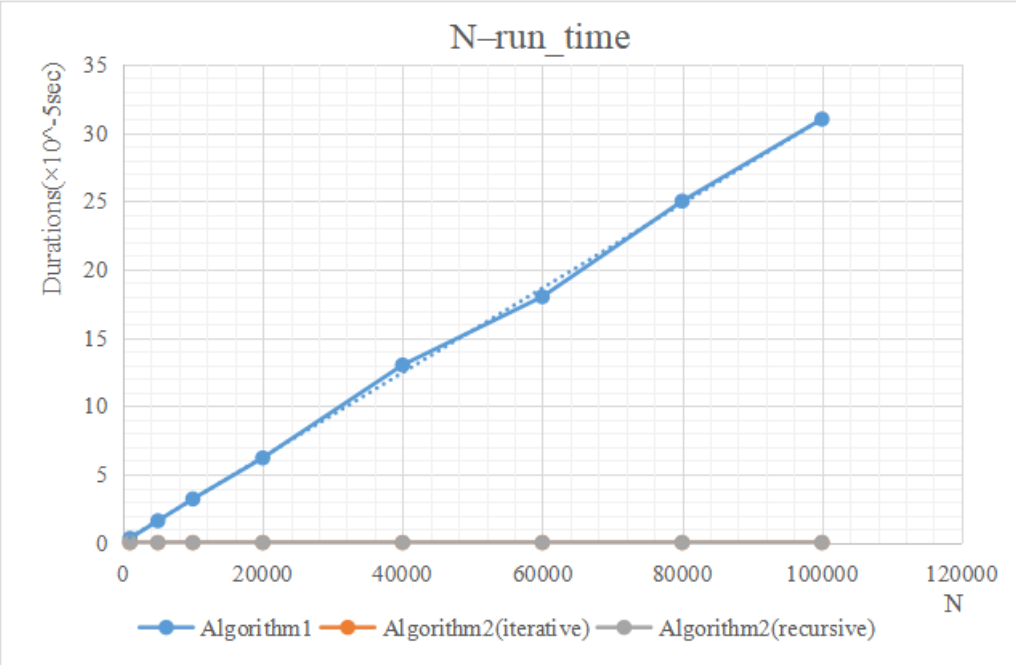
Run the program, get the data and fill in the table, get the following results.

	N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm1	Iterations (K)	10^4	10^3	10^3	10^3	10^2	10^2	10^2	10^2
	Ticks	31	16	32	62	13	18	25	31
	Total Time (sec)	0.031	0.016	0.032	0.062	0.013	0.018	0.025	0.031
	Duration (sec)	3.1×10^{-6}	1.6×10^{-5}	3.2×10^{-5}	6.2×10^{-5}	1.3×10^{-4}	1.8×10^{-4}	2.5×10^{-4}	3.1×10^{-4}
Algorithm2 (iterative version)	Iterations (K)	10^6	10^6	10^6	10^6	10^6	10^6	10^6	10^6
	Ticks	29	36	38	41	43	45	46	48
	Total Time (sec)	0.029	0.036	0.038	0.041	0.043	0.045	0.046	0.048
	Duration (sec)	2.9×10^{-8}	3.6×10^{-8}	3.8×10^{-8}	4.1×10^{-8}	4.3×10^{-8}	4.5×10^{-8}	4.6×10^{-8}	4.8×10^{-8}
Algorithm2 (recursive version)	Iterations (K)	10^6	10^6	10^6	10^6	10^6	10^6	10^6	10^6
	Ticks	36	50	53	59	64	65	67	68
	Total Time (sec)	0.036	0.050	0.053	0.059	0.064	0.065	0.067	0.068
	Duration (sec)	3.6×10^{-8}	5.0×10^{-8}	5.3×10^{-8}	5.9×10^{-8}	6.4×10^{-8}	6.5×10^{-8}	6.7×10^{-8}	6.8×10^{-8}

It can be seen from the table that the time required for the two implementation methods of the algorithm 2 is far less than the time required for the algorithm 1, which shows that the efficiency of the algorithm 2 is much higher than the algorithm 1.

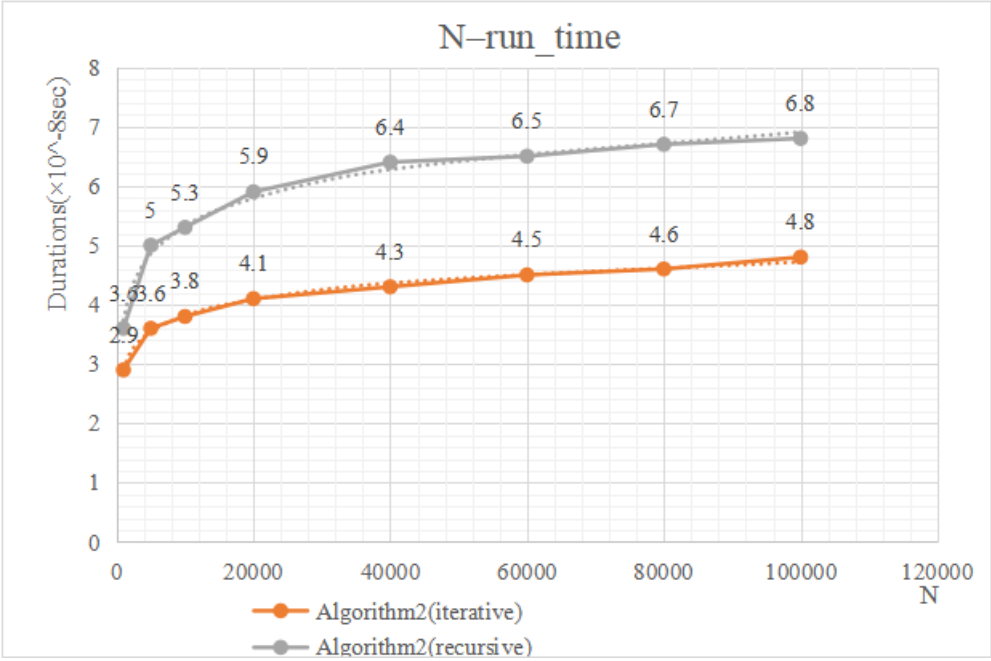
3.2 Algorithm comparison

We draw the time curve required for each operation of the three functions under different N values, as shown in the figure below.



It can be seen that the time required for the algorithm 1 is much higher than the two forms of the algorithm 2, and it increases almost linearly with the increase of N. The dotted line is the trend line, which is in line with our expectations and also in line with The fact that the time complexity of the algorithm 1 is $O(N)$.

In order to compare the two implementations of the algorithm 2 more intuitively, we draw two time curves separately, as shown in the figure below.



The gray and orange curves are the time required for recursion and iteration respectively. It can be clearly seen that the time required for iteration is less than that of recursion. This is because in the process of iterative calculations, a lot of repeated calculations are reduced, which further saves time, but the time complexity of the two is the same, both are $O(logN)$, which is why the changing trends of the two time curves are almost the same.

The dashed line in the figure is an asymptote fitted according to the data points of the two time curves, and the fitting model is a logarithmic function. From the perspective of the fitting effect, the actual time curve required for the function to run is very close to the fitted logarithmic function curve, which has reached our expectation, which also conforms to the fact that the time complexity is $O(logN)$.

Chapter 4: Analysis and Comments

4.1 The complexity

4.1.1 Time complexity

For the algorithm 1, each call to the function will perform N loops, and the complexity of each loop is $O(1)$, so the time complexity of this algorithm is $O(N)$.

For the two forms of the algorithm 2, the time complexity is both $O(logN)$. For the recursive form, the problem is divided into half each time the calculation is performed. The program runs with $O(logN)$. A small change to the program will not change its time complexity. The key is $return(x*x,n/2)$ and $return(x*x,n/2)*x$. For the iterative form, it actually converts N into a 2 hexadecimal system, determines whether it is 1 from low to high, and performs corresponding operations. The time complexity is also $O(logN)$.

4.1.2 Space complexity

For the algorithm 1, its space complexity is obviously $O(1)$.

For the algorithm 2, the space complexity of its two implementation forms are different. For the recursive form, the recursion depth is $O(log_2N)$, and the space of each recursion is constant, so the space complexity is $O(log_2N)$. For the iterative form, its space is always a constant level, so its space complexity is $O(1)$.

4.2 Algorithm improvement

Both the recursive and iterative forms of the algorithm 2 used by $Project$ can complete the calculation in $O(logN)$ time. Compared with the former, the latter reduces a lot of repeated calculations and is already very efficient, but There is still room for improvement.

As mentioned above, the iterative function we used essentially uses the binary form of N to retrieve in turn whether the low to high digits are 1 or 0. In the program, we use to determine whether N is divided by 2 To achieve the remainder. In fact, we can filter out the low-order 0 first, and then perform calculations. For example, $144=128+16$, its binary form is 10010000, and there are a lot of low-order 0. After filtering these 0, it can improve the efficiency of the program to a certain extent.

Appendix: Source Code

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

double Directmultiply(double x,int n);//function of Algorithm 1
double Iterative(double x,int n);//function of Algorithm 2 in iterative form
double Recursive(double x,int n);//function of Algorithm 2 in recursive form

int main(){
    clock_t start,stop;
    double total,duration;
    int i,j,K=1;//i and j are the variables used in the loop, and K is the
iteration number
    double x=1.0001,result;//result is the return value of the function
    int n[8]={1000,5000,10000,20000,40000,60000,80000,100000};//store the index
N into an array for easy traversal
//  double a=Directmultiply(2,3);
//  double b=Recursive(3.1,2);
```

```
//  double c=Iterative(2.5,2);
//  printf("%f %f %f",a,b,c);
    printf("Algorithm 1\n");
    for(i=0;i<8;i++){
        K=1;//set the initial value of K to 1
        do{//enter the loop,record the time required for K calls
            K*=10;
            start=clock();//record start time
            for(j=0;j<K;j++){
                result=Directmultiply(x,n[i]);//call function
            }
            stop=clock();//record end time
        }while(stop-start<10);//end the loop when tick is greater than 10
        total=((double)(stop-start))/CLK_TCK;//calculate the total time
        duration=total/K;//calculate single time
        printf("N=%d,K=%d,Ticks=%d,total time=%.1e,duration=%.1e\n",n[i],K,stop-
start,total,duration);
    }
    printf("\n");
    printf("Algorithm 2-iterative version\n");
    for(i=0;i<8;i++){
        K=1;//set the initial value of K to 1
        do{//enter the loop,record the time required for K calls
            K*=10;
            start=clock();//record start time
            for(j=0;j<K;j++){
                result=Iterative(x,n[i]);//call function
            }
            stop=clock();//record end time
        }while(stop-start<10);//end the loop when tick is greater than 10
        total=((double)(stop-start))/CLK_TCK;//calculate the total time
        duration=total/K;//calculate single time
        printf("N=%d,K=%d,Ticks=%d,total time=%.1e,duration=%.1e\n",n[i],K,stop-
start,total,duration);
    }
    printf("\n");
    printf("Algorithm 2-recursive version\n");
    for(i=0;i<8;i++){
        K=1;//set the initial value of K to 1
        do{//enter the loop,record the time required for K calls
            K*=10;
            start=clock();//record start time
            for(j=0;j<K;j++){
                result=Recursive(x,n[i]);//call function
            }
            stop=clock();//record end time
        }while(stop-start<10);//end the loop when tick is greater than 10
        total=((double)(stop-start))/CLK_TCK;//calculate the total time
        duration=total/K;//calculate single time
        printf("N=%d,K=%d,Ticks=%d,total time=%.1e,duration=%.1e\n",n[i],K,stop-
start,total,duration);
    }
}

double Directmultiply(double x,int n){
    int i;
    double result=1;//define variables for output
    if(n==0){
```

```
        return 1;//if the index is 0, return 1 directly
    }
    else{
        for(i=0;i<n;i++){//enter the loop
            result*=x;//multiply x each time
        }
    }
    return result;
}

double Iterative(double x,int n){
    double result=1;//define variables for output
    if(n==0){
        return 1;//if the index is 0, return 1 directly
    }
    while(n>0){//when n>0, continue the while loop
        if(n%2==1){//if n is odd, multiply result by x
            result*=x;
        }
        x*=x;//square x
        n/=2;//divide n by 2, automatically round
    }
    return result;//return calculation result
}

double Recursive(double x,int n){
    if(n==0){
        return 1;//if the index is 0, return 1 directly
    }
    if(n==1){
        return x;//if the index is 0, return x directly
    }
    //if n is not trivial, divide the problem in half and call the recursive
function
    if(n%2==0){
        return Recursive(x*x,n/2);//if n is an even number, split x^n into
x^(n/2)*x^(n/2)
    }
    else{
        return Recursive(x*x,(n-1)/2)*x;//if n is odd, split in the same way
    }
}
```

Declaration

I hereby declare that all the work done in this project is of my independent effort.