Project 1 : Performance Measurement(POW)

September 24, 2019

Chapter1 Introduction

Problem Description

There are at least two ways to compute X^N for some positive integer N.
Algorithm 1 : Simple interation
It uses N-1 multiplications,which is the most common way.
Algorithm 2 : Divide And Conquer
It works as follows:

$$X^N = X^{N/2} \times X^{N/2} \quad \text{(N is even)}$$
$$X^N = X^{(N-1)/2} \times X^{(N-1)/2} \times X \quad \text{(N is odd)}$$

It is an optimization of algorithm 1, which is more efficient.

This project aims at analyzing the complexities of these two algorithms,so we have to test the actual running time and make charts to compare them visually. What's more, algorithm 2 can be implemented by a recursive or an iterative way. In this case, we will also compare them in complexity.

Chapter2 Algorithm Specification

algorithm_1(X, N)

```
if N == 0
    return result
else
    for N → 1
        result = result * X
return result
```

---

algorithm_2_iterative(X, N)

```
parity[32] = 0                         // use to record the parity
i = 0                                  // index
result = 1
while N > 0                            // record the parity
    if N mod 2 == 0
        parity[i] = 2
    else
        parity[i] = 1
    i++
    n = n/2
i--
for i → 0                              // move the one more count
    if parity[i] == 2
        result = result * result
    elseif parity[i] == 1
        result = result * result * X
return result
```

---

algorithm_2_recursion(X, N)

```
if N == 0
    return 1
if N == 1
    return X
if N mod 2 == 0
    return algorithm-2-recursion(X * X, N/2)
else return aglorithm-2-recursion(X * X, N/2) * X
```

This function calculates the running time of one execution.

```
runtime_calculation(duration, Times)
return duration/Times                    // calculate the true result
```

The procedure to measure the performance of function in function main.

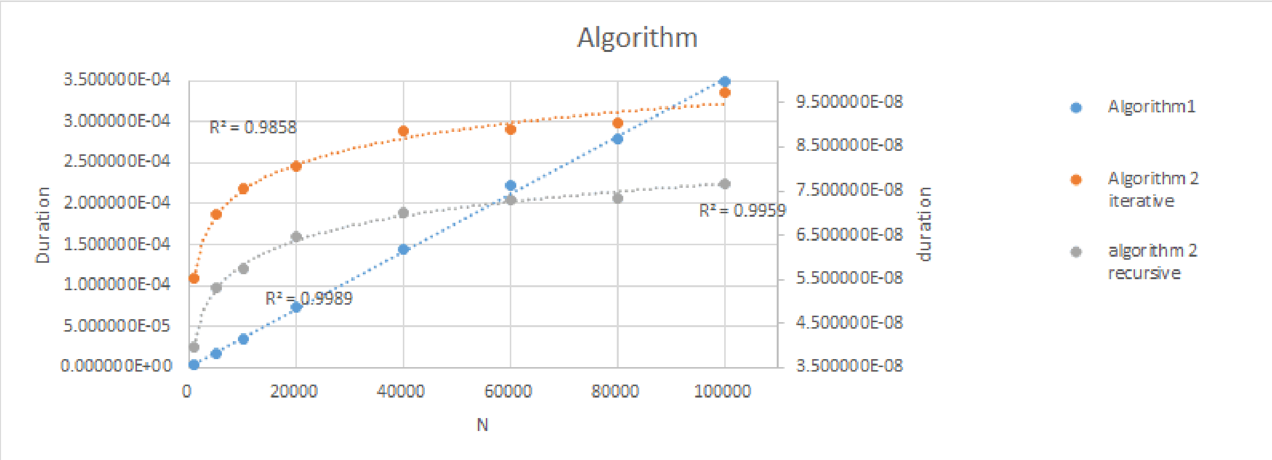
```
MAIN()
                                         // we just represent part of this function
i = 0
start = clock()                          // records the ticks at the end of the function
                // perform K times to ensure that the running time is measurable
while i < Times
    function(X, N)
    result = 1
    i++
stop = clock()
duration = ((double)(stop-start))/CLK_TCK
                        // CLK_TCK is a built_in constant = ticks per second
runtime_calculation(duration, Times)
                                         // calculate the running time of the function
```

Chapter3 Testing Results

	N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm1	Iterations(K)	58000	12000	6000	3000	1500	1000	750	600
	Ticks	202	208	215	219	215	220	209	210
	Total Time(sec)	0.202	0.208	0.215	0.219	0.215	0.220	0.209	0.210
	Duration(sec)	3.482759E-06	1.733333E-05	3.583333E-05	7.300000E-05	1.433333E-04	2.220000E-04	2.786667E-04	3.500000E-04
Algorithm2 (iterative version)	Iterations(K)	3500000	2500000	2300000	2100000	2000000	1900000	1850000	1840000
	Ticks	193	174	174	169	177	169	167	179
	Total Time(sec)	0.193	0.174	0.174	0.169	0.177	0.169	0.167	0.179
	Duration(sec)	5.514286E-08	6.960000E-08	7.565217E-08	8.047619E-08	8.850000E-08	8.894737E-08	9.027027E-08	9.728261E-08
Algorithm2 (recursive version)	Iterations(K)	3500000	2500000	2300000	2100000	2000000	1900000	1850000	1840000
	Ticks	139	133	132	136	140	139	136	141
	Total Time(sec)	0.139	0.133	0.132	0.136	0.140	0.139	0.136	0.141
	Duration(sec)	3.971429E-08	5.320000E-08	5.739130E-08	6.476190E-08	7.000000E-08	7.315789E-08	7.351351E-08	7.663043E-08

The purpose of the test is to obtain the single running time of the three algorithms when the power exponent (N) is different, and analyse the test results. According to the time complexity of each algorithm, the exponential N of algorithm 1 should have a linear relationship with the Duration of a single run, and the N and Duration of algorithm 2 should have a logarithm relationship. When the test was repeated more than ten times, the two kind of relationships between Dutations and Ns were in line with expectations.

Chapter4 Analysis and Comments



Analysis

Algorithm1 dose a cycle of n-1 times, each operation is once (result *= x), so the time complexity is O(n). Algorithm 2 uses a idea similar to dichotomy: if the exponent is even, compute result = result * result; If the exponent is odd, compute result = result * result * x. It takes logN operations from N to 1. So the time complexity is O(logN).According to the table and figure, it can be seen that algorithm 1 is much slower than algorithm 2: when N=1000, algorithm 1 is two orders of magnitude slower than algorithm 2, and when N grows to 10000, algorithm 1 is four orders of magnitude slower than algorithm 2.Meanwhile,the exponent of algorithm 1 is approximately linear with the single running time. The exponent of algorithm 2 and its single running time approximately changes in logarithm relation, but the recursive implementation is slightly faster than the iterative implementation (probably because the iterative implementation method builds an array to store the result of dividing each exponent by two, and then corresponding operations are performed according to the array). The space complexity of algorithm1 is O(1),and the space complexity of algorithm2(both iterative and recursive)is O(logN)

Comments

In C, "bit operations" are available and faster: if the exponent N is even, the last digit in its binary representation must be 0; If N is odd, the last digit of its binary representation must be 1. If we "&" it with the binary of one, then we get the last digit of N.The result 0 shows that N is even, and 1 indicates that N is odd. So the judgement of odd or even number can be replaced by a bit operation.Similarly, we can just shift the binary representation of N by one bit to the right ($N \gg 1$)to get half of it, just as shown in the picture:

```
int fastPower(int base, int exponent) {
    int sum = 1;
    while (exponent != 0) {
        if ((exponent & 1) != 0) {
            sum *= base;
        }
        exponent = expnonent >> 1;  // 对指数进行移位
        base *= base;               // 让base的次幂以2的倍数增长
    }
    return 0;
}
```

(Code comes from the Internet.)

Appendix: Source Code (in C)

```c
#define _CRT_SECURE_NO_WARNINGS//avoid the warn of scanf
#include<stdio.h>
#include<time.h>
#include<math.h>

//#define Times 2100000//number of times(ticks) the function runs

clock_t start, stop;//clock_t is a built_in type for processor time
double duration;//records the run time(seconds) of a function
double result = 1;

double algorithm_1(double x, int n);
double algorithm_2_iterative(double x, int n);
double algorithm_2_recursion(double x, int n);
double runtime_calculation(double duration,unsigned long Times);//caculate the run time of one execution

int main(void)
{
    while(1){
        int i=0;//Cycle control variable
        double x;
        int n;
        unsigned long Times;
        //the same as X,N in the problem

        //scan data
        printf("Please enter the power first, then the index, separated by spaces in the middle.\n");
        scanf("%lf", &x);
        scanf("%d", &n);
        printf("Please enter how may times should this algorithm run.");
        scanf("%d", &Times);
        //run algorithm_1
        start = clock();//records the ticks at the end of the function
        while (i < Times)//perform K times to ensure that running time is measurable
        {
            algorithm_1(x, n);
            result = 1;
            i++;
        }
        stop = clock();
        duration = ((double)(stop - start)) / CLK_TCK;//CLK_TCK is a built_in constant = ticks per second
        //print the run time of algorithm_1 and the result
        printf("one tick is : %d\n", CLK_TCK);
        printf("The total ticks : %d\n", stop-start);
        printf("The total running time is : %lf\n", duration);
        printf("The running time of one execution of algorithm_1 is : %e\n", runtime_calculation(duration,Times));
        printf("The result of %f^%d is: %f\n\n", x, n, algorithm_1(x, n));

```

```
        //run algorithm_2_iterative
        i = 0;
        start = clock();//records the ticks at the end of the function
        while (i < Times)//perform 100 times to ensure that running time is measurable
        {
            algorithm_2_iterative(x, n);
            i++;
        }
        stop = clock();
        duration = ((double)(stop - start)) / CLK_TCK;//CLK_TCK is a built_in constant = ticks per second
        //print the run time of algorithm_1 and the result
        printf("The total ticks : %d\n", stop-start);
        printf("The total running time is : %lf\n", duration);
        printf("The running time of one execution of algorithm_2_iterative is : %e\n", runtime_calculation(duration,Times));
        printf("The result of %f^%d is: %f\n\n", x, n, algorithm_2_iterative(x, n));

        //run algorithm_2_recursion
        i = 0;
        start = clock();//records the ticks at the end of the function
        while (i < Times)//perform 100 times to ensure that running time is measurable
        {
            algorithm_2_recursion(x, n);
            i++;
        }
        stop = clock();
        duration = ((double)(stop - start)) / CLK_TCK;//CLK_TCK is a built_in constant = ticks per second
        //print the run time of algorithm_1 and the result
        printf("The total ticks : %d\n", stop-start);
        printf("The total running time is : %lf\n", duration);
        printf("The running time of one execution of algorithm_2_recursion is : %e\n", runtime_calculation(duration,Times));
        printf("The result of %f^%d is: %f\n", x, n, algorithm_2_recursion(x, n));
    }
    return 0;
}

```

```
double algorithm_1(double x, int n)
{
    if(n == 0){
        return result;
    }
    else{
        for(;n>0;n--){
            result *= x;
        }
    }
    return result;
}

```

```
//Iterative solution
double algorithm_2_iterative(double x, int n)
{
    int parity[32] = { 0 };//use to record the parity
    int i = 0;//index
    double result = 1;
    //record the parity
    while (n>0)
    {
        if (n % 2 == 0)
            parity[i] = 2;
        else
            parity[i] = 1;
        i++;
        n /= 2;
    }
    i--;//move the one more count
    for (; i >= 0; i--)
    {
        if (parity[i] == 2)
        {
            result = result * result;
        }
        else if (parity[i] == 1)
        {
            result = result * result * x;
        }
    }
    return result;
}

double algorithm_2_recursion(double x, int n)
{
    if (n == 0)
    {
        return 1;
    }
    if (n == 1)
    {
        return x;
    }
    if (n % 2 == 0)
    {
        return algorithm_2_recursion(x*x, n / 2);
    }
    else
    {
        return algorithm_2_recursion(x*x, n / 2)*x;
    }
}

double runtime_calculation(double duration,unsigned long Times)//caculate the run time of one execution
{
    return duration / Times;//caculate the true result
}
```

Declaration

We hereby declare that all the work done in this project titled " Project 1 : Performance Measurement(POW)" is of our independent effort as a group.