# Different Ways to Compute X^N

# Team 8

# Date: 2019-9-22

Chapter1: Introduction

There are a few ways to compute . We can use N-1 multiplications. We can also do it in the following way: if N is even = × ; and if N is odd, = × ×X. The latter algorithm also has more than one version. There are iterative version and recursion version. We use these different ways to compute and use function clock to measure the performances of these algorithm. In this way, we can compare their time complexities and decide which way is the best.

Chapter2: Algorithm Specification

**We write three algorithm to solve the problem.**

**Algorithm1(N-1 multiplications):**

result ‏‏‏‏🡨 x;

for t 🡨 0 to t<the power of x - 1

result 🡨 result\*x;

return result;

**Algorithm2(iterative):**

result 🡨 1

while the power of x>0

if(the power of x%2==1)

then result 🡨 result\*x

the power of x 🡨 the power of x / 2

x 🡨 x\*x

return result

**Algorithm3(recursion):**

Algo\_2\_rec(number x; the power of x)

if (the power of x == 0) then return 1 if (the power of x == 1) then return x if (the power of x % 2 == 0) then return Algo\_2\_rec(x\*x, the power of x / 2) else return Algo\_2\_rec(x\*x, the power of x/ 2)\*x

Test algorithm:

start = clock()

run the function

stop = clock()

time = (stop – start) /CLK\_TCK

Chapter3: Testing Results

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | N | 1000 | 5000 | 10000 | 20000 | 40000 | 60000 | 80000 | 100000 |
| Algorithm1 | Iterations(K) |  |  |  |  |  |  |  |  |
| Ticks | 27 | 11 | 23 | 46 | 92 | 15 | 18 | 23 |
| Total Time(sec) | 0.027 | 0.011 | 0.023 | 0.046 | 0.092 | 0.015 | 0.18 | 0.23 |
| Durations(sec) |  |  |  |  |  |  |  |  |
| Algorithm2  （iterative version） | Iterations(K) |  |  |  |  |  |  |  |  |
| Ticks | 43 | 46 | 48 | 51 | 53 | 56 | 57 | 59 |
| Total Time(sec) | 0.043 | 0.046 | 0.048 | 0.051 | 0.053 | 0.056 | 0.057 | 0.059 |
| Durations(sec) |  |  |  |  |  |  |  |  |
| Algorithm2  （recursive version） | Iterations(K) |  |  |  |  |  |  |  |  |
| Ticks | 15 | 19 | 21 | 22 | 24 | 25 | 26 | 27 |
| Total Time(sec) | 0.015 | 0.019 | 0.021 | 0.022 | 0.024 | 0.025 | 0.026 | 0.027 |
| Durations(sec) |  |  |  |  |  |  |  |  |

Chart 1 runtime statistics

**But when fitting the data, we find that it is not appropriate to draw these three function images in ordinary coordinates at the same time (Chart 2), so we use logarithmic coordinates (Chart 3).**

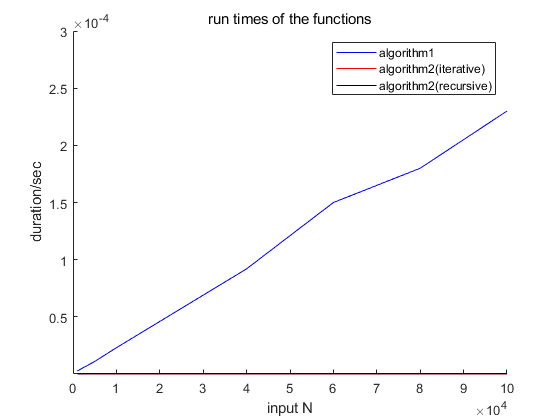
****

Chart2 ordinary analysis of function running time

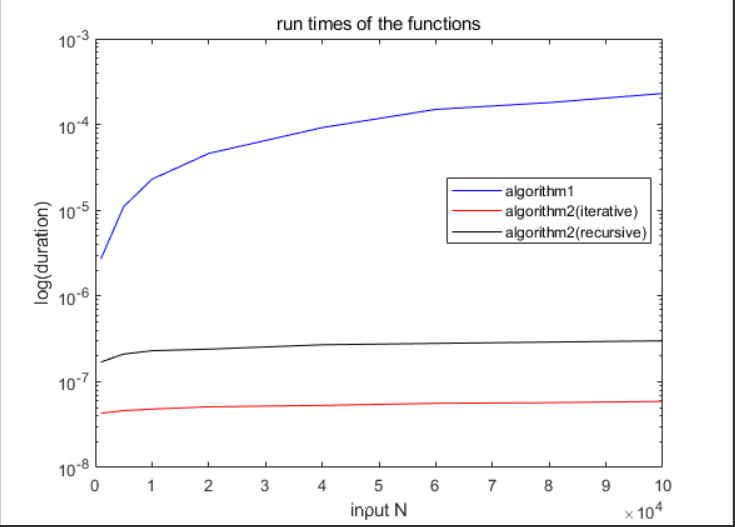


Chart3 logarithmic analysis of function running time

Chapter4: Analysis and Comments

4.1 Time Complexity Analysis

As is shown in the plot, the iterative version of the Algorithm2 is the most efficient, while the Algorithm 1 is the least efficient, and the differences between the two is quite significant. As you can see in the plot1 (the normal coordinate), we can hardly see the black and the red line, which stand for the two versions of the Algorithm2, since the quantity of them are too small, that’s why we draw another plot in logarithmic coordinates.

For Algorithm1, the time complexity is , since the we multiply the base X for N times. Though this algorithm is straightforward and easy-to-understand, it takes far more time than the other algorithms.

As for Algorithm 2 and 3, each time N is reduced by half, resulting in cutting the problem in half, which minimizing the running time significantly. So the time complexity of Algorithm 2 and 3 are both . We can also see that when the input N is getting large, these two algorithms have a stable growth in running time.

Moreover, different versions of a same algorithm vary greatly. As is shown in table, the iterative version is 10 times faster than the recursive version of Algorithm2. That is because recursive algorithm calls itself repeatedly, while the iterative algorithm only needs to do the iteration.

4.2 Space Complexity Analysis

The space complexity of Algorithm 1 is , but the time complexity of Algorithm 2 and 3 are . Although Algorithm 2 and 3 cost more space than Algorithm 1, the operational efficiency is improved. So sometimes we sacrifice space for time. Also, we can see that the efficiency and the functional calls are the disadvantage of recursion.

Appendix: Source Code

#include<stdio.h>

#include<time.h>

clock\_t start, stop; // record the time when a function start/stop

double duration; // record the run time (seconds) of a function

void testAlgorithm1(double X, double N); //these are test functions

void testAlgorithm2\_rec(double X, double N);

void testAlgorithm2\_ite(double X, double N);

double Algo\_1(double x, int n); //3 algorithms to implement POW

double Algo\_2\_rec(double x, int n);

double Algo\_2\_ite(double x, int n);

int main(void)

{

const double X = 1.0001; //X is the base number (1.0001 according to the instruction)

int N = 0; // N is the exponent, which can be an arbitrary value

printf("Please input the value of N：");

scanf("%d", &N);

printf("\n");

// in each test, we first input the iteration K, then the duration will be output

testAlgorithm1(X, N);

testAlgorithm2\_ite(X, N);

testAlgorithm2\_rec(X, N);

return 0;

}

double Algo\_1(double x, int n) { // Algorithm 1 is to use N−1 multiplications.

double result = x;

for (int i = 1; i < n; i++) {

result \*= x; // N−1 multiplications

}

return result;

}

double Algo\_2\_rec(double x, int n) { //the recursive version, Figure 2.11 in the textbook

if (n == 0)

return 1;

if (n == 1)

return x;

if (n % 2 == 0) // n is odd

return Algo\_2\_rec(x \* x, n / 2);

else // n is even

return Algo\_2\_rec(x \* x, n / 2) \* x;

}

double Algo\_2\_ite(double x, int n) { // the iterative version

double result = 1;

while (n > 0) {

if (n % 2 == 1) {

result \*= x;

}

n = n / 2; // reduce n to half of its original value

x = x \* x; // each x to its original square.

}

return result;

}

void testAlgorithm1(double X, double N)

{

{

int k = 0; //iteration nmber

printf("Please input the iteration number of Algorithm1:");

scanf("%d", &k);

start = clock(); // start timing

while (k--)

Algo\_1(X, N);

stop = clock();

// stop timing

duration = ((double)(stop - start)) / CLK\_TCK;

printf("The duration of Algorithm1 (k times) is %f s\n\n", duration);

}

}

void testAlgorithm2\_ite(double X, double N)

{

{

int k = 0; //iteration nmber

printf("Please input the iteration number of Algorithm2 (iterative version):");

scanf("%d", &k);

start = clock(); // start timing

while (k--)

Algo\_2\_ite(X, N);

stop = clock();

// stop timing

duration = ((double)(stop - start)) / CLK\_TCK;

printf("The duration of Algorithm2\_ite (k times) is %f s\n\n", duration);

}

}

void testAlgorithm2\_rec(double X, double N)

{

{

int k = 0; //iteration nmber

printf("Please input the iteration number of Algorithm2 (recursive version):");

scanf("%d", &k);

start = clock(); // start timing

while (k--)

Algo\_2\_rec(X, N);

stop = clock();

// stop timing

duration = ((double)(stop - start)) / CLK\_TCK;

printf("The duration of Algorithm2\_rec (k times) is %f s\n\n", duration);

}

}

Declaration

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

**Duty Assignment:**

Programmer:张佳文

Tester:王睿

Report writer: 杨云皓