**Performance Measurement**

**--------Project report 1 of Data Structure**

**Chapter 1: Introduction**

The problem is to compute XN in a more efficient way. The exponent N can be a very large number.

To solve this problem, the most direct way is to use N-1 multiplications. We call it *Algorithm 1*. We simply use FOR LOOP to iterate it for N-1 times. However, it’s also an inefficient way. The time complexity is O(N).

Actually, we can take the strategy of divide-and-conquer. As we know, when searching data from a sequential array, we use binary search to make it faster. This is a typical application of divide-and-conquer algorithm. We continuously use constant time to divide the problem into two sub-problems. So the time complexity is O(log N).

To compute XN, we can also adopt this method. We can divide XN into two parts or three parts. If N is even, XN= XN/2\*XN/2; and if N is odd, XN= X(N-1)/2\*X(N-1)/2\*X. We call it *Algorithm 2.* The time complexity of it is also O(log N). We will implement it in two versions: *iterative version* and *recursion version*.

In *iterative version*, we use the method of bitwise operation to make the program more efficient and more elegant. To calculate the iteration times calls for another LOOP. This leads to iterative version much slower. So we can simply judge when N is 0 in place of this inefficient way.

In *recursion version*, we first call recursive function to calculate M to avoid repeating calculation. If N is odd, we return M\*M\*x. If it’s even, we return M\*M.

**Chapter 2: Algorithm Specification**

Algorithm 1:

The most apparent algorithm. Use a FOR LOOP to do N multiplications( To do N-1 multiplications is also OK.)

Algorithm 2 *iterative version*:

At first, to use a loop to do iteration, we need to know the times of iterations. However, it’s impossible for us to get this value in O(1) time, for the value of N is uncertain. We can’t judge how many times when N is odd and how many times when N is even without a loop.

However, because in every iteration, N is divided by two, we can use N to judge when the loop halts. (If N equals to 0, the loop halts.) Then when N is odd, we just need to multiply x for one more time.

To make the program more elegant and more concise, we use bitwise operation to simplify it. (But it’s not necessary. ) And also to do less operation, we do the multiplication with M only in odd situation. Because the first bit of N in binary code must be 1, there must be at least one time to do assignment operation with M.

Algorithm 2 *recursion version*:

As a recursive function, the first step is to set base case. If N equals to 1, return x. And if N equals to 0, return 1. To judge whether N equals to 0 is just for x==0 situation. Other situation’s base cases are all N equals to 1.

Then according to the recursive formula we analyze in chapter 1, if N is even, XN= XN/2\*XN/2; and if N is odd, XN= X(N-1)/2\*X(N-1)/2\*X. To make the program more efficient, we calculate X first and then plug it into the formula.

Here is the description in pseudo-code of all the three algorithms to solve the problem.

int main()

{

x ← 1.0001

Input exponent N and iterations K

▷ Test algorithm 1

set the start of clock

for i ← 1 to K

sum ← function1(x, N)

stop the clock

ticks ← stop - start

total ← ticks/CLK\_TCK

duration ← total/K

Output the data

▷ Test iterative version of algorithm 2

▷ Test recursion version of algorithm 2

▷ Are similar to algorithm 1

}

function 1

{

M ← 1

for i ← 0 to n-1 M ← M\*x

return M

}

function 2

{

M ← 1;

while(N)

{

if (N is odd) M ← M \* x

▷ If "n" is odd, then M is multiplied by "x"

x ← x \* x

▷ Actually, we only need to multiple an extra x when N is odd

▷ We notice that in this function, the only operation on "M" is to multiply it by "x"(when n is odd), and leave the rest to the next iteration. As we may need to multiply "M" by "x" twice next time, by simply squaring "x" each time.

N ← N/1

}

return M

}

function 3

{

if (N = 1) then return M ← x

if (N = 0) then return M ← 1

▷ set the basic base

else if (N is even) then {

M ← function3(x, N/2)

M ← M \* M;

}

▷ when N is even

else if (N is odd) then {

M ← function3(x, (N-1)/2)

M ← M \* M \* x;

}

▷ when N is odd

}

**Chapter 3: Testing Results**

We 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. We use iteration number K to make the result more accurate. Here is the testing results.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | N | 1000 | 5000 | 10000 | 20000 | 40000 | 60000 | 80000 | 100000 |
| Algorithm  1 | K | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** |
| Ticks | **48183** | **241140** | **484343** | **970097** | **1946105** | **2895734** | **3831137** | **4747306** |
| Total Time | **2647.417582** | **13249.45055** | **26612.25275** | **53302.03297** | **106928.8462** | **159106.26** | **210502.033** | **260840.989** |
| Duration | **0.264742** | **1.324945** | **2.661225** | **5.330203** | **10.692885** | **15.910626** | **21.050203** | **26.084099** |
| Algorithm  2  （iterative  version） | K | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** |
| Ticks | **293** | **360** | **376** | **457** | **496** | **540** | **545** | **550** |
| Total Time | **16.098901** | **19.78022** | **20.659341** | **25.10989** | **27.252747** | **29.67** | **29.945055** | **30.21978** |
| Duration | **0.00161** | **0.001978** | **0.002066** | **0.002511** | **0.002725** | **0.002967** | **0.002995** | **0.003022** |
| Algorithm  2  （recursive  version） | K | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** | **10000** |
| Ticks | **912** | **1426** | **1440** | **1562** | **1779** | **1964** | **2284** | **2493** |
| Total Time | **50.10989** | **78.351648** | **79.120879** | **85.824176** | **97.747253** | **107.9** | **125.494505** | **136.978022** |
| Duration | **0.005011** | **0.007835** | **0.007912** | **0.008582** | **0.009775** | **0.010791** | **0.012549** | **0.013698** |

**Charts：**

Ticks is a integer variable and Duration equals to Ticks/CLK\_TCK. So in order to ensure the accuracy of Duration variable, Ticks should have enough significant digits. However, when N is small, Ticks is also too small to be used to calculate Duration. So here we use a variable K to iterate for more times and adjust it with the vary of N.

To show it in a more intuitive way, we have 2 plots of them. (Please pay attention to vertical axis. The value on the vertical axis in Fig. 1 is much larger than in Fig. 2.)

**Plots：**

Fig. 1

Fig. 2

So here is the conclusion.

**Chapter 4: Analysis and Comments**

Apparently, Iterative and recursive algorithms are much more efficient than algorithm 1 (simple multiply). According to the plot 1, the plot is a line, so the time complexity of is O(N).According to the plot 2, the plot is just like log N function, so the time complexity of is O(log N).

For comparison of function2 and function3, as the chart shown in Fig. 2, Iterative version of algorithm 2 is much more efficient than the recursion version.

1. For the iterative version, the time complexity is only increasing because of the times of the loop, and there is no extra need of space cost. But the question of iterative algorithm is that it’s difficult to understand, and sometimes it’s unlikely to find a good iterative algorithm.
2. For the recursion version, the advantage is that it is easy to understand and it makes the code more readable. But it requires more space complexity because it calls functions. And it may cause stack overflow, too.
3. As a result, we came into the conclusion that the recursion must have the thought of iterative algorithm, but the iterative ones do not have recursion in it sometimes. And we should use iterative algorithm if could find a good way to use it (just like this project) both in the consideration of time and space complexity.

**Appendix: Source Code (in C)**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FileName: DS\_Project\_1.c

Author: 牛奔放、张倬豪、郭爽

Date: 10.4

Description: Compare two different algorithms of computing X^N

Function List:

1. double function1(double x, int N);

2. double function2(double x, int N);

3. double function3(double x, int N);

Three ways of performing X^N

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define CLK\_TCK 18.2 /\*In case that const is not in the library\*/

clock\_t start, stop; /\* clock\_t is a built-in type for processor time (ticks) \*/

double function1(double x, int N);

double function2(double x, int N);

double function3(double x, int N);

int main()

{

double ticks;

double duration, total; /\* records the run time (seconds) of a function \*/

int K, N, m, i;

double x = 1.0001, sum = 0;

printf("Enter N\n");

scanf("%d", &N);/\*input N\*/

printf("Enter K\n");

scanf("%d", &K);/\*input K\*/

start = clock(); /\* records the ticks at the beginning of the function call \*/

printf("input a number\n1 for simple multiply\n2 for iterative version\n3 for recusivee version\n");

scanf("%d",&m);

switch(m) /\* run your function here for K times\*/

{

case 1: for(i=1;i<=K;i++)

sum=function1(x,N);break;

case 2: for(i=1;i<=K;i++)

sum=function2(x,N);break;

case 3: for(i=1;i<=K;i++)

sum=function3(x,N);break;

default: printf("Illegal Number!"); break;

}

stop = clock(); /\* records the ticks at the end of the function call \*/

ticks = stop - start;

total = (double)ticks/CLK\_TCK;

duration = total/K;

printf("sum = %lf\n",sum);

printf("ticks = %lf\ntotaltime = %lf\nduration = %lf\n", ticks, total, duration);

return 1;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: // function1

Description: // simple multiply of X^N

Calls: // none

Called By: // main

Input: // double x, int N

Output: // none

Return: // the result of multiply

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double function1(double x, int N)

{

int i;

double M = 1; /\*give M an initial number\*/

for(i = 0 ; i < N; i++)

M = M \* x;

return M;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: // function2

Description: // iterative algorithm of X^N

Calls: // none

Called By: // main

Input: // double x, int N

Output: // none

Return: // the result of multiply

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double function2(double x, int N)

{

double M = 1;

while(N)

{

if(N & 1) M \*= x; /\* If "n" is odd, then M must be multiplied by "x". Here is the only assignment operation with M, because we know the first bit of N in binary code must be 1. \*/

x \*= x; /\*Actually, we noticed that we only need to multiple an extra x when N is odd\*/

N >>= 1; /\*That is, N = N/2\*/

}

return M;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: // function3

Description: // recursive algorithm of X^N

Calls: // none

Called By: // main

Input: // double x, int N

Output: // none

Return: // the result of multiply

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double function3(double x, int N)

{

double M;

if(N == 1)

M = x;

else if(N == 0)

M = 1; /\* Base cases\*/

else if(N & 1) /\*When N is odd\*/

{

M = function3(x, (N-1)/2); /\*To minimize the running time,we should calculate the function as less as possible\*/

M = M \* M \* x; /\* N is odd, so X^N = X^((N-1)/2) \* X^((N-1)/2) \* X \*/

}

else /\*When N is even\*/

{

M = function3(x, N/2); /\* N is even, so X^N = X^(N/2) \* X^(N/2) \*/

M = M \* M;

}

return M;

}

**Declaration**

***We hereby declare that all the work done in this project titled "ds\_project1\_report" is of our independent effort as a group.***

**Duty Assignments:**

**Programmer: 牛奔放**

**Tester: 张倬豪**

**Report Writer: 郭爽**