Normal Performance Measurement (POW)

ZHONGWEI YU

Date: 2023-10-8

Chapter 1: Introduction

There are different algorithms that can compute X^N for some positive integer N. Such as:

Algorithm 1: use N−1 multiplications.

Algorithm-2: if N is even, X^N=X^(N/2)×X^(N/2); and if N is odd, X^N =X^[(N−1)/2]×X^[(N−1)/2]×X.

You need to firstly implement Algorithm 1 and an iterative version of Algorithm 2;then analyze the complexities of algorithms, and show the analysis and the conclusion in the report; and you need to test your program, measure the performances of each of them, and compare their performances. It is necessary to make a chart to show their performance.

Note: set X=1.0001 and N = 1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000 ,for comparison.

Chapter 2: Algorithm Specification

Note: function names are bolded.

main function**:**

1. Print the prompt to choose an algorithm.

2. Read the selected algorithm number into variable 'q'.

3. If 'q' equals 1, call the **multiplication** function.

4. If 'q' equals 2, call the **recursion** function.

5. If 'q' equals 3, call the **iteration** function.

6. Otherwise, go back to step 2.

multiplication function:

1. Print the prompt to enter the exponent.

2. Read the exponent 'N' into variable 'N'.

3. Initialize timer variables 'ttime' and 'ticks' to 0. Set a benchmark value 'T' as 1000.

4. Loop 'T' times, performing the following steps each time:

    a. Start the timer.

    b. Define a variable 'res' and initialize it with 'x'.

    c. Iterate from 1 to 'N', multiplying 'x' with 'res' and assigning the result back to 'res'.

    d. Stop the timer and calculate the runtime duration.

    e. Accumulate the current tick count into 'ticks' and the current runtime into 'ttime'.

5. Print the value of 'T'.

6. Print the value of 'ticks'.

7. Print the value of 'ttime'.

recursion function:

1. Print the prompt to enter the exponent.

2. Read the exponent 'N' into variable 'N'.

3. Initialize timer variables 'ttime' and 'ticks' to 0. Set a benchmark value 'T' as 1000.

4. Loop 'T' times, performing the following steps each time:

    a. Start the timer.

    b. Define a variable 'res' and initialize it with 0.

c. If 'N' is odd, call the **ODD** function to compute the result and assign it to 'res'.

If 'N' is even, call the **EVEN** function to compute the result and assign it to 'res'.

d. Stop the timer and calculate the runtime duration.

e. Accumulate the current tick count into 'ticks' and the current runtime into 'ttime'.

5. Print the value of 'T'.

6. Print the value of 'ticks'.

7. Print the value of 'ttime'.

ODD function:

1. If 'n' is less than or equal to 1, return 'x'.

2. Reduce 'n' by 1 and divide it by 2, storing the result into 'n'.

3. Return the result of **ODD** function(n, x) multiplied by the result of **EVEN** function(n, x), then multiplied by 'x'.

EVEN function:

1. If 'n' is less than or equal to 1, return 'x'.

2. Divide 'n' by 2, storing the result into 'n'.

3. Return the result of **EVEN** function(n, x) multiplied by the result of **EVEN** function(n, x).

iteration function:

1. Print the prompt to enter the exponent.

2. Read the exponent 'N' into variable 'N'.

3. Initialize timer variables 'ttime' and 'ticks' to 0. Set a benchmark value 'T' as 1000.

4. Loop 'T' times, performing the following steps each time:

    a. Start the timer.

    b. Define a variable 'res' and initialize it with 0.

    c. Call the **POWER** function to compute the result and assign it to 'res'.

    d. Stop the timer and calculate the runtime duration.

    e. Accumulate the current tick count into 'ticks' and the current runtime into 'ttime'.

5. Print the value of 'T'.

6. Print the value of 'ticks'.

7. Print the value of 'ttime'.

POWER function:

1. Define a variable 'res' and initialize it to 1.

2. While 'n' is greater than 0, perform the following steps:

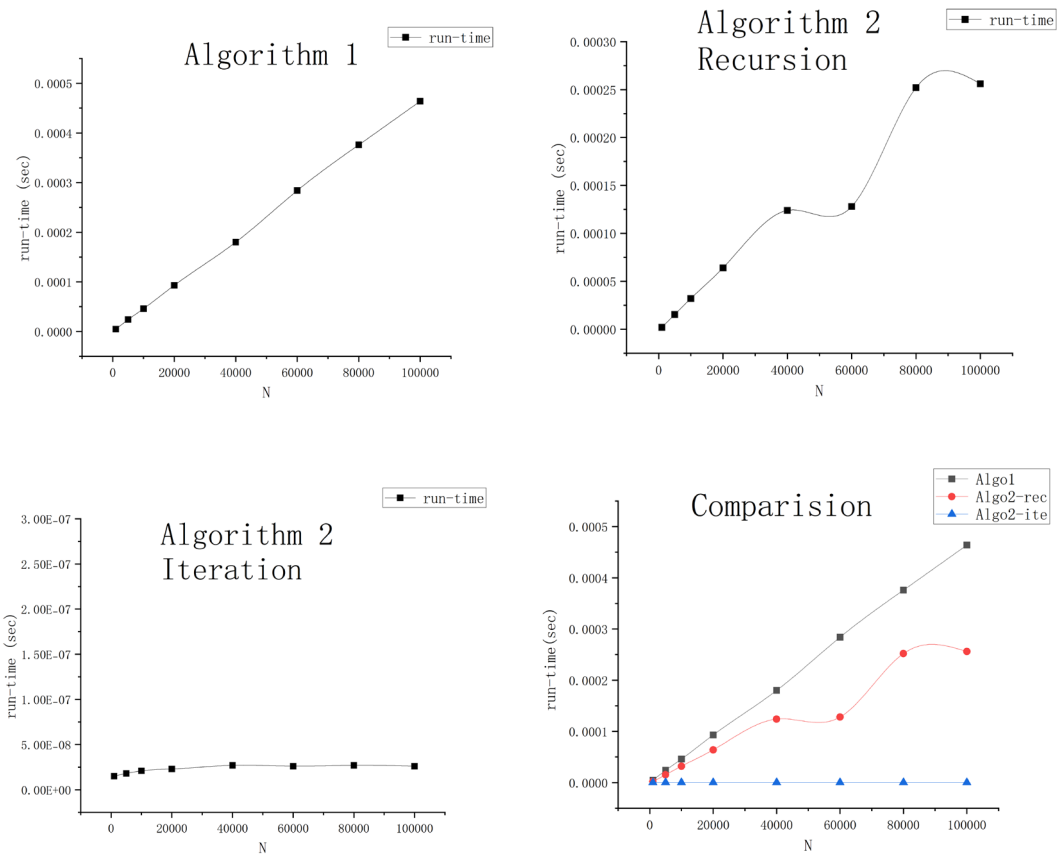
    a. If 'n' is odd, multiply 'res' by 'x'.

b. Multiply 'x' by 'x' and assign the result back to 'x'.

c. Divide 'n' by 2 and assign the result back to 'n'.

3. Return 'res'.Chapter 3: Testing Results

	N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1	Iterations(K)	10000	1000	1000	1000	500	250	250	250
	Ticks	46	24	46	93	90	71	96	116
	Total Time (sec)	0.046	0.024	0.046	0.093	0.09	0.071	0.094	0.116
	Duration(sec)	0.0000046	0.000024	0.000046	0.000093	0.00018	0.000284	0.000376	0.000464
Algorithm 2-res	Iterations(K)	10000	5000	1000	1000	500	500	250	250
	Ticks	19	77	32	64	62	64	63	64
	Total Time (sec)	0.019	0.077	0.032	0.064	0.062	0.064	0.063	0.064
	Duration(sec)	0.0000019	0.0000154	0.000032	0.000064	0.000124	0.000128	0.000252	0.000256
Algorithm 2-ite	Iterations(K)	1000000	1000000	1000000	1000000	100000	1000000	1000000	1000000
	Ticks	15	18	21	23	27	26	27	26
	Total Time (sec)	0.015	0.018	0.021	0.023	0.027	0.026	0.027	0.026
	Duration(sec)	0.00000015	0.00000018	0.00000021	0.00000023	0.00000027	0.00000026	0.00000027	0.00000026



The results are generally in line with expectations(all pass).

Chapter 4: Analysis and Comments

Analysis of the time and space complexities of the algorithms.

1. Algorithm 1

Time complexity: O(N)

This algorithm loops N-1 simple multiplications after each run, so it is easy to get its time complexity as O(N).

Space complexity: O(1)

No extra space related to the size of the input is used in the code, only a small number of variables are used, so the space complexity is O(1).

2. Algorithm 2-recursion

Time complexity: O(logN)

The number of calls to a recursive function is related to the exponent N, and each recursion halves the exponent. So, the total number of recursive calls is $\log_2 N$, by which we can get its time complexity as O(logN).

Space complexity: O(logN)

Recursive functions use stack space to hold the local variables and return addresses of each recursive call. Since the recursion depth is logN, the extra space required is O(logN). Note that the other variables in the code do not increase with the size of the input, so their spatial complexity can be viewed as O(1). To sum up, the spatial complexity of the entire code is O(logN).

3. Algorithm 2-iteration

Time complexity: O(1)

The loop part is executed K times, in which the POWER function is called once for each loop. The number of iterations of the while loop inside the function is related to the exponential N, but since the value of N in each loop will be divided by 2, the total number of iterations does not exceed logN, which can be regarded as constant level. So here is O(1).

Space complexity: O(1)

Other than input variables and a fixed amount of space footprint, the code does not use any additional space. The space footprint of all variables is

fixed and does not increase with the size of the input, so the space complexity can be viewed as O(1).

Comments on further possible improvements:

Algo1:Use bitwise operations to check the parity of N in the while loop instead of using modulus operation, as bitwise operations are generally faster, but unfortunately I didn't know until after I finished the statistics, so I didn't change the code.

Algo2-res: Handle the exponent N with special cases, such as dividing it by 2 when N is even and squaring the base number x, to reduce the number of iterations and improve code speed and memory usage.

Algo2-ite: I don't have an idea about it at the moment.

Appendix: Source Code (in C)

project-1.c

Declaration

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