Performance Measurement

Fundamentals of Data Structures

Laboratory Projects 1

(曾天)[1], (孟展豪)[2], and Chen Hanyi (陈翰逸)[3]

[1]College of Computer Science and Technology, Zhejiang University
[2]College of Computer Science and Technology, Zhejiang University
[3]College of Computer Science and Technology, Zhejiang University

September 30, 2016

Contents

Chapter 1

Introduction

Problem description and (if any) background of the algorithms.

Chapter 2

Algorithm Specification

Description (pseudo-code preferred) of all the algorithms involved for solving the problem, including specifications of main data structures.

2.1 Algorithm 1

$$X^N = \begin{cases} 0, & X = 0 \wedge N \neq 0 \\ 1, & X \neq 0 \wedge N = 0 \\ X, & N = 1 \\ (X^{\frac{N}{2}})^2, & N \text{ is an even number} \\ (X^{\frac{N-1}{2}})^2 \times X, & N \text{ is an odd number} \end{cases}$$

Algorithm 1 The algorithm 1 to calculate power

```
if some condition is true then
    do some processing
else if some other condition is true then
    do some different processing
else
    do the default actions
end if
```

2.2 Algorithm 2 recursive version

2.3 Algorithm 2 iterative version

To calculate X^N, we first express the N of X^N in binary number system,

$$X^{N_{(10)}} = X^{N_{(2)}} = X^{n_k n_{k-1} \ldots n_2 n_1{}_{(2)}} = X^{\sum_{i=1}^{k} n_i \times 2^i} = \prod_{i=1}^{k} X^{n_i \times 2^i}$$

and then if we have the value of $X^{2^i}(1 \leq i \leq k)$, we can calculate X^N in $O(k)$ time.

According to following equation, values of $X^1, X^2, X^4, ..., X^{2^{k-1}}, X^{2^k}$ could be get by multiplying $k-1$ times.

$$X^{2^k} = (X^{2^{k-1}})^2$$

Besides, $2^k \le N$, so $k \le log_2 N$.

Finally, we get an algorithm which can calculate X^N in $O(logN)$ time.

Chapter 3

Testing Results

3.1 Algorithm 1

O(N) time

Table 3.1: test result of the Algorithm 1

N	Iterations (K)	Ticks	Total Time (sec)	Duration (sec)
1000	1000000	4896065	4.896065	4.896065×10^{-6}
5000	200000	4917766	4.917766	2.458883×10^{-5}
10000	100000	4896786	4.896786	4.896786×10^{-5}
20000	50000	4812429	4.812429	9.624858×10^{-5}
40000	25000	4823615	4.823615	1.929446×10^{-4}
60000	16666	4827711	4.827711	2.896742×10^{-4}
80000	12500	4826477	4.826477	3.861182×10^{-4}
100000	10000	4799912	4.799912	4.799912×10^{-4}

3.2 Algorithm 2 (recursive version)

O(logN) time

Table 3.2: test result of the Algorithm 2 (recursive version)

N	Iterations (K)	Ticks	Total Time (sec)	Duration (sec)
1000	10000000	1502949	1.502949	1.502949×10^{-7}
5000	10000000	1858551	1.858551	1.858551×10^{-7}
10000	10000000	1932978	1.932978	1.932978×10^{-7}
20000	10000000	2104549	2.104549	2.104549×10^{-7}
40000	10000000	2228389	2.228389	2.228389×10^{-7}
60000	10000000	2674874	2.674874	2.674874×10^{-7}
80000	10000000	2384372	2.384372	2.384372×10^{-7}
100000	10000000	2543837	2.543837	2.543837×10^{-7}

3.3 Algorithm 2 (iterative version)

O(logN) time

Table 3.3: test result of the Algorithm 2 (iterative version)

N	Iterations (K)	Ticks	Total Time (sec)	Duration (sec)
1000	10000000	609220	0.609220	6.092200×10^{-8}
5000	10000000	1035374	1.035374	1.035374×10^{-7}
10000	10000000	1164042	1.164042	1.164042×10^{-7}
20000	10000000	1260366	1.260366	1.260366×10^{-7}
40000	10000000	1323494	1.323494	1.323494×10^{-7}
60000	10000000	1078078	1.078078	1.078078×10^{-7}
80000	10000000	1397193	1.397193	1.397193×10^{-7}
100000	10000000	1411758	1.411758	1.411758×10^{-7}

Chapter 4

Analysis and Comments

4.1 Algorithm 1

4.1.1 Time complexity

The for loop in line ????? of is executed N times.

The function multiplies N times and each multiply operation takes constant time, so the time complexity of this algorithm is $O(N)$.

4.1.2 Space complexity

The whole function only use 4 variables, so the space complexity of this algorithm is $O(1)$.

4.2 Algorithm 2 recursive version

4.2.1 Time complexity

$O(logN)$

4.2.2 Space complexity

example :
Stack: 1000 -> 500 -> 250 -> 125 -> 62 -> 31 -> 15 -> 7 -> 3 -> 1

$O(logN)$

4.3 Algorithm 2 iterative version

4.3.1 Time complexity

$O(logN)$

4.3.2 Space complexity

$O(1)$

Appendices

Appendix A

Source Code (in C)

A.1 Main program

Listing A.1: main.c

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

/**
 * define the function to calculate power.
 * @param double x, int n
 * @require x is a real number,
 *          n is an integer and n >= 0
 * @return double x ^ n
 */
double calculatePower(double x, int n);

int main(int argc, char *argv[])
{
    const double x = 1.0001;
    int iterations, n, i;

    sscanf(argv[1], "%d", &iterations);
    sscanf(argv[2], "%d", &n);

    /** measure the performance of the function */
    clock_t start = clock(), stop;

    /** run the function for "iterations" times */
    for (i = 0; i < iterations; i ++) {
        calculatePower(x, n);
    }

    stop = clock();

    int ticks = stop - start;
    double total_time = (double)ticks / CLOCKS_PER_SEC;
    double duration = total_time / iterations;
    printf("iterations = %d, ticks = %d, total_time = %lf, duration = %e\n",
        iterations, ticks, total_time, duration);
    return 0;
}
```

Listing A.2: algorithm 1

```c
#include "main.c"

/**
 * the alogrithm 1 to calculate power
 * @param double x, int n
 * @require x is a real number,
 *          n is an integer and n >= 0
 * @return double x ^ n
```

```c
 */
double calculatePower(double x, int n)
{
    double result = 1.0;
    int i;

    /** the for loop runs and multiplys n (contains 0) times. */
    for (i = 0; i < n; i ++) {
        result *= x;
    }
    return result;
}
```

Listing A.3: algorithm 2 (iterative version)

```c
#include "main.c"

/**
 * the iterative version of the alogrithm 2 to calculate power
 * @param double x, int n
 * @require x is a real number,
 *          n is an integer and n >= 0
 * @return double x ^ n
 */
double calculatePower(double x, int n)
{
    double sq = x, result = 1.0;
    int i;
    /**
     * traverse all bits of n.
     * the for loop runs and multiplys log_2(n) times.
     */
    for (i = 0; (1 << i) <= n; i ++) {
        /**
         * if the i th bit of n is '1',
         * then multiplys result by x ^ (2 ^ i).
         */
        if ((1 << i) & n) {
            /** sq = x ^ (2 ^ i) */
            result *= sq;
        }
        /** x ^ (2 ^ (i + 1)) = (x ^ (2 ^ i)) ^ 2 */
        sq *= sq;
    }
    return result;
}
```

Listing A.4: algorithm 2 (recursive version)

```c
#include "main.c"

/**
 * the recursive version of the alogrithm 2 to calculate power
 * @param double x, int n
 * @require x is a real number,
 *          n is an integer and n >= 0
 * @return double x ^ n
 */
double calculatePower(double x, int n)
{
    if (n == 0) {
        /** x ^ 0 = 1 */
        return 1.0;
    }
    if (n == 1) {
        return x;
    /**
     * to determinate wether n is an odd number
     * n & 1 = n % 2
     */
    } else if (n & 1) {
        /**
```

```
         * n >> 1 = n / 2
         */
        double sq = calculatePower(x, n >> 1);

        /**
         * x ^ n = (x ^ ((n - 1) / 2)) ^ 2 * x
         * (x is an odd number)
         */
        return sq * sq * x;
    } else {
        double sq = calculatePower(x, n >> 1);

        /**
         * x ^ n = (x ^ (n / 2)) ^ 2
         * (x is an even number)
         */
        return sq * sq;
    }
}
```

A.2 Analysis tools

Listing A.5: Analysis tools for the algorithms

```
#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;

int main(void)
{
    /** 8 different values of n */
    const int n[8] = {1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000};

    /** the file name of programs */
    const string prog_file_name[3] = {"algo_1", "algo_2_recursive", "algo_2_iterative"};

    for (int i = 0; i < 3; i++) {
        cout << prog_file_name[i] << " : " << endl;
        for (int j = 0; j < 8; j++) {
            cout << "n = " << n[j] << "," << endl;

            int iterations;
                if (i == 0) {
                    /**
                     * To make the total time which the program cost in the same range (several seconds),
                     * the value of iterations is set to be (iterations / n[j]).
                     *
                     * command : ./prog iterations n
                     */
                    iterations = 1000000000 / n[j];
                } else {
                        iterations = 10000000;
                }
            system(("./" + prog_file_name[i] + " " + to_string(iterations) + " " + to_string(n[j])).c_str());
            cout << endl;
        }
        cout << endl << endl;
    }
    return 0;
}
```

A.3 Test tools

Listing A.6: main.c(to show the result of the 3 functions)

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

/**
 * define the function to calculate power.
 * @param double x, int n
 * @require x is a real number,
 *          n is an integer and n >= 0
 * @return double x ^ n
 */
double calculatePower(double x, int n);

int main(void)
{
        double x;
        int n;
        scanf("%lf%d", &x, &n);
        /**
         * to view the result of "calculatePower"
         */
        printf("%f\n", calculatePower(x, n));
        return 0;
}
```

Appendix B

Declaration and Duty Assignments

Declaration

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

Duty Assignments

Programmer: XXX
Tester: XXX
Report Writer: XXX