Report - Assignment 2

Camilla Kristiansen Birkelund IDATT2101 2. September 2022

Assignment description

There were two parts to this assignment. The first part was to create a program that could solve the following equation by using recursion:

$$x^n = \begin{cases} 1 \; when \; n = 0 \\ x * x^{n-1} \; when \; n > 0 \end{cases}$$

The second part of the assignment was to create a program for the following equation using recursion, and to compare the difference in runtime to the first program:

$$x^n = \begin{cases} 1 \; when \; n = 0 \\ x * (x^2)^{\frac{n-1}{2}} \; when \; n \; is \; an \; odd \; number \\ (x^2)^{\frac{n}{2}} \; when \; n \; is \; an \; even \; number \end{cases}$$

Implementation

First algorithm

The first part of the assignment was solved by creating an algorithm to solve the equation using recursion. The algorithm simply returns 1 if n equals zero, and $x * x^{n-1}$ if n equals one or higher. There is currently no exception handling concerning values of n that are lower than zero, which means the program will crash if n values lower than zero are attempted.

```
//Finding the value of x^n.
double findxPown(double x, int n) {
    if (n == 0) {
        return 1.;
    }

    return x * findxPown(x, n - 1);
}
```

Figure 1: The algorithm used to solve part one of the assignment.

To find the time it takes to run the program, the most accurate solution was to see how many times it is able to run in one second, and then divide one second by the amount of times it was able to run. This was implemented using a while-loop to count the amount of completed calculations.

```
auto a_second = std::chrono::duration<int>(1);
auto start = std::chrono::high_resolution_clock::now();
int counter = 0;

//Finding the number of times the findxPown algorithm can be run in a second.
while (std::chrono::high_resolution_clock::now() - start < a_second) {
    findxPown(1.01, 10000);
    counter ++;
}

//Finding the average runtime of the algorithm.
auto freq = std::chrono::duration<double>(1./counter);
auto timeUsed = std::chrono::duration_cast<std::chrono::nanoseconds>(freq);
```

Figure 2: The code used to measure time in part one of the assignment.

Second algorithm

A big portion of the same code was used to implement the second part of the assignment. The algorithm was modified to fit the second equation, with conditions to determine whether n was an odd or even number, or if n was equal to zero. There was not any exception handling in this program either, which would cause the program to crash for negative values of n.

```
//Finding the value of x^n.
double findxPown(double x, int n) {
    //Returns one if n=0.
    if (n == 0) {
        return 1.;
    }

    //Checking if n is an odd number, returns the following if it is.
    if (n % 2 != 0) {
        return x * findxPown(x*x, (n-1) / 2);
    }

    //Returns the following if n is an even number.
    return findxPown(x*x, n / 2);
}
```

Figure 3: The algorithm used to solve the equation in part two of the assignment.

The same logic was applied to measure the time used to run the program as in the first part of the assignment.

Testing

To test that the code had the correct semantics and syntax, two test values were used to control that the calculations gave the expected results.

Test values	2^{12}	3^{14}
Expected result	4096	4782969
First algorithm	4096	4782969
Second algorithm	4096	4782969

Table 1: Results from testing the calculations in part one and two of the assignment.

After getting the expected results for the test values, different values were used to test the difference in the time used by each program. The registered time will help determine how the time difference is related between the two implementations.

x = 1.0001, n-values:	10	100	1000	10000	100000
Time used for the first algorithm	102 ns	552 ns	5290 ns	52009 ns	654878 ns
Time used for the second algorithm	72 ns	91 ns	108 ns	124 ns	140 ns
Time used for the pow(x,n) method	53 ns	53 ns	53 ns	54 ns	54 ns

Table 2: Results from measuring the time for different values of x^n.

Results

From the registered values in table 2 it is possible to observe some connections between the different measurements. The difference in time used by each equation's algorithm can be explained by their differing time complexities. The first algorithm took ten times longer when

n was ten times bigger, which means it has a complexity of T(n) $\in \Theta$(n). The time used in the second algorithm increases at a much lower rate. In this algorithm, every new recursion will be ran as n/2, which has a complexity of $T(n) = T(n/2) + 1$. The master method is necessary to analyze the complexity of this, because it fits into the equation $T(n) = a * T(n/b) + c * n^k$. Our original equation has a = 1, b = 2, c = 1 and k = 0. This means that $b^k = a$, which gives a result of

$T(n) \in \Theta(n^k * log\, n) \Leftrightarrow \Theta(1 * log\, n) \Leftrightarrow \Theta(log\, n)$.

Practically, this means that the time difference will become bigger for the two algorithms the larger the n value is. This is also demonstrated in table 2.

When using the pow(x, n)-method, which is a built-in method to calculate xn, the time used increases very slightly for increasing values of n. This is also a logarithmic growth, but the algorithm is much faster than the second equation.