

CH08-320143: Assignment #5

Harit Krishan

h.krishan@jacobs-university.de

Jacobs University Bremen — March 9, 2020

1 Fibonacci Numbers

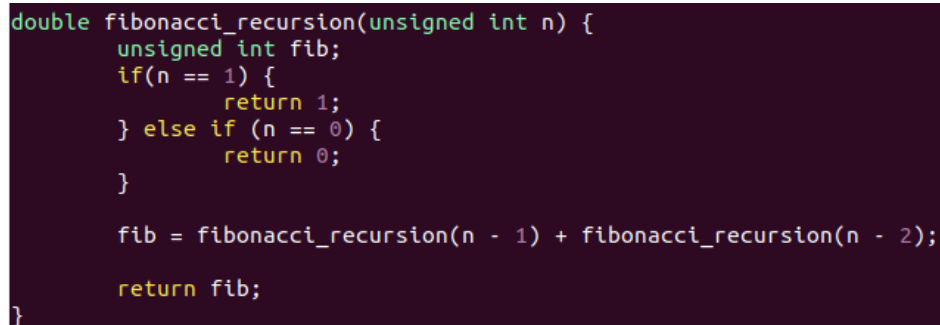
Fibonacci Numbers are a sequence of numbers in which each number is the sum of the two preceding numbers. There are several ways of implementing the Fibonacci computation. We discuss four different algorithms for computing a Fibonacci number - naive approach, bottom-up approach, closed form approach and matrix approach.

1.1 Implementation

The Fibonacci sequence is a sequence in which each term is the sum of the preceding two terms.

1.1.1 Naive Approach

The naive approach is quite straightforward. Below is a screenshot of the C++ implementation of the naive approach to computing the n th Fibonacci number. This method is also called the recursive method because we recursively call a function to compute the value.

A screenshot of a code editor showing a C++ function named 'fibonacci_recursion'. The function takes an 'unsigned int n' as input and returns a 'double'. It uses a recursive approach: if n is 1, it returns 1; if n is 0, it returns 0; otherwise, it returns the sum of 'fibonacci_recursion(n - 1)' and 'fibonacci_recursion(n - 2)'.

```
double fibonacci_recursion(unsigned int n) {
    unsigned int fib;
    if(n == 1) {
        return 1;
    } else if (n == 0) {
        return 0;
    }

    fib = fibonacci_recursion(n - 1) + fibonacci_recursion(n - 2);

    return fib;
}
```

Figure 1: Recursive Algorithm for Computing Fibonacci Number in C++

1.1.2 Closed Form Approach

The closed-form approach uses the laws of exponents to compute the n th Fibonacci number. On the following page we can see the implementation of this approach in C++. We first define a helper function which we use in the main computation. For the algorithm to compile correctly, we include the `<cmath>` header file.

1.1.3 Bottom-Up Approach

The bottom-up approach computes the n th Fibonacci number by creating an array of length n and iteratively fill the array using the axioms that the first term is 0 and the second term is 1. Figure 3 shows the implementation of this algorithm in C++.

```

// Below I define a function for implementing exponents.
double exp_n(double base, double power) {
    if(power != 0) {
        return (base*exp_n(base, power - 1));
    } else {
        return 1;
    }
}

double fibonacci_closed(unsigned int n) {
    double fi1, fi2, fi3, fib;

    fi1 = (1+sqrt(5))/2;
    fi2 = (1-sqrt(5))/2;
    fi3 = ((exp_n(fi1, n)) - (exp_n(fi2, n)));
    fib = (fi3/sqrt(5));

    return fib;
}

```

Figure 2: Closed Form Algorithm for Computing Fibonacci Number in C++

```

double fibonacci_bottom_up(unsigned int n) {
    unsigned int i;
    double fib[n];

    fib[0] = 0.0;
    fib[1] = 1.0;

    for(i = 2; i <= n; i++) {
        fib[i] = fib[i-1] + fib[i-2];
    }

    return fib[n];
}

```

Figure 3: Bottom-Up Algorithm for Computing Fibonacci Number in C++

1.1.4 Matrix Method

For this algorithm, we define two helper functions - multiply() and power(). These functions are incorporated into the main algorithm. The Fibonacci number can be computed using a linear system. Figure 4 shows this implementation in C++.

1.2 Sampling

We sample the different implementations with numbers until we reach a threshold computation time of 0.001s per number. It is important to bound our data to this threshold because these algorithms have varying efficiencies. Ultimately, we bound the number of samples to the number of samples computed using the matrix method given the threshold. This code can be found in the file *fibonacci.cpp*. This program creates data files for run times of each algorithm for each sample. We plot these files using Gnuplot. The Gnuplot script can be found in the file *gnuplot_script*. Figure 5 shows the graph of run times for each algorithm for a rising input.

1.3 Verification

It is important to check if we receive the same Fibonacci number for the same n . We get the same Fibonacci number for all the algorithms except for the closed form approach. In this approach, the produced value is a good *approximation* of n up until a very large n . Therefore, not all the results are same after a very large number of inputs. For the time being, we could only compute 25 Fibonacci numbers using the naive algorithm given the time threshold, hence it makes sense to only compare the first 25 numbers for all the

```

// Next two functions are helper functions for the matrix implementation of
// computing a fibonacci number.
// -> multiply()
// -> power()

void multiply(double F[2][2], double M[2][2]) {
    double x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    double y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    double z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    double w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

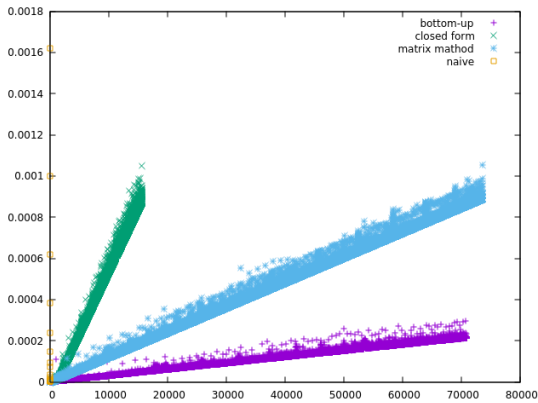
    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

void power(double F[2][2], double n) {
    int i;
    double M[2][2] = {{1,1},{1,0}};
    for(i = 2; i <= n; i++) {
        multiply(F, M);
    }
}

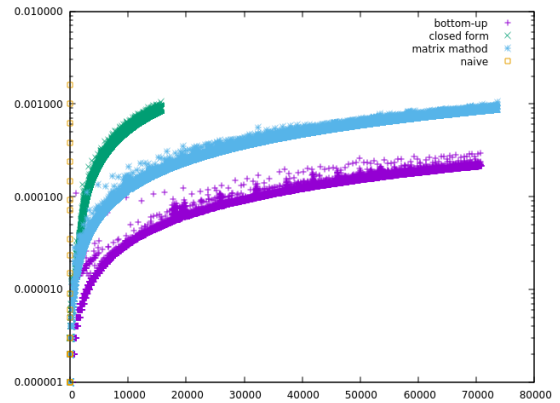
double fibonacci_matrix(unsigned int n) {
    double F[2][2] = {{1,1},{1,0}};
    if(n == 0) {
        return 0;
    }
    power(F, n-1);
    return F[0][0];
}

```

Figure 4: Matrix Method Algorithm for Computing Fibonacci Number in C++



(a) Normal Scale



(b) Logarithmic Scale

Figure 5: My flowers.

algorithms. We write a script which compiles the output files of results from the initial test. This code can be found in the file *read_data.cpp*. This script create a file which shows the computed Fibonacci number against the index n . A snapshot of this output can be seen in Figure 6.

1.4 Results Interpretation

The results of the computations can be seen in Figure 6. The output files for each of the algorithms are in the root directory of this submission. The timings are also recorded in separate files algorithm-wise. In order to interpret the results, refer to Figure 5. We can see through the normal scale that the closed

Index	Naive	Closed	Bottom	Matrix
0	0	0	0	0
1	1	1	1	1
2	1	1	1	1
3	2	2	2	2
4	3	3	3	3
5	5	5	5	5
6	8	8	8	8
7	13	13	13	13
8	21	21	21	21
9	34	34	34	34
10	55	55	55	55
11	89	89	89	89
12	144	144	144	144
13	233	233	233	233
14	377	377	377	377
15	610	610	610	610
16	987	987	987	987
17	1597	1597	1597	1597
18	2584	2584	2584	2584
19	4181	4181	4181	4181
20	6765	6765	6765	6765
21	10946	10946	10946	10946
22	17711	17711	17711	17711
23	28657	28657	28657	28657
24	46368	46368	46368	46368

Figure 6: Console Output of Custom Compiler Script Showing Computed Fibonacci Values. Top row shows column labels.

form, bottom-up, and matrix algorithms have a linear time complexity. Different algorithms have different rates of change of computation time. The bottom-up algorithm seems to have the lowest rate of change of computation time. The naive/recursive algorithm has an exponential time complexity. This implies that the recursive method has the worst time complexity.

We applied linear regression to the respective algorithm-specific data files to compute lines of best fit for the data clusters. We perform all these computations in Gnuplot. The code can be found in the file *gnuplot_script*. Figure 7 shows a graph including the linear regressions in normal scale.

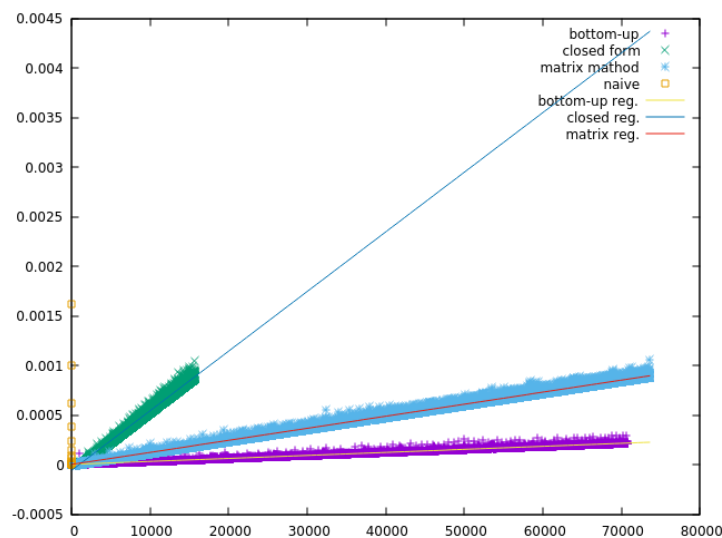


Figure 7: Regressions of Algorithm Run Time Data Sets Graphed Along with Data Sets

Using the linear regression function of Gnuplot, we were able to compute the equations for the run times of the linear time complexity algorithms. Below are the equations.

Closed Form	$6.01597 * 10^{-8} * x - 5.93717 * 10^{-5}$
Bottom-Up	$3.04513 * 10^{-9} * x + 8.42313 * 10^{-7}$
Matrix	$1.21524 * 10^{-8} * x + 1.01186 * 10^{-6}$

2 Divide & Conquer and Solving Recurrences

Divide and conquer approach dictates that you break the problem down into subproblems recursively until subproblem is solvable. Then work back by solving all subproblems and merging your solutions.

2.1 Brute-Force Multiplication Complexity

We must first understand how bitwise multiplication works. This works the same way multiplication sums are solved on paper. We multiply each individual bit of one number with the entire other number and sum the individual products. In computer terms, this would imply bit-shifting by one to the left. For example -

$$5 * 6 = 30$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 101 \\ 1010 \\ \hline 11110 \end{array}$$

$$\therefore (11110)_2 = 30$$

We see above that the successive product terms are bit-shifted to the left by their index values before summation. We assume that this addition, subtraction and bit-shifting is done in linear time. In the multiplication, we make n^2 operations, therefore we get the following -

$$\therefore T(n) = \Theta(n^2)$$

When we do the summation part, that is another n^2 operations. Therefore we can write the following -

$$\Theta(n^2) + \Theta(n^2) = \Theta(2n^2) \Rightarrow \Theta(n^2)$$

2.2 Applying Divide & Conquer

If we take the divide and conquer approach for defining a multiplication algorithm, we would separate the two numbers in half, multiply the halves, then sum them together. We make the assumption that the numbers can be divided in half. In this procedure, we break down the numbers in terms of the sum of bases and then linearly compute the scalar product of the expansions.

$$\begin{aligned} 23 * 12 &= (2 * 10^1 + 3 * 10^0)(1 * 10^1 + 2 * 10^0) \\ &= 2 * 10^2 + 10^1 * (2 * 2 + 3) + 3 * 2 \\ &= 200 + 70 + 6 \\ &= 276 \end{aligned}$$

This same intuition can be applied to binary numbers as well.

$$\begin{aligned}
x &= x_l * 2^{n/2} + x_r \\
y &= y_l * 2^{n/2} + y_r \\
\therefore x * y &= (x_l * 2^{n/2} + x_r) * (y_l * 2^{n/2} + y_r) \\
&= x_l * y_l * 2^n + 2^{n/2}(x_l * y_r + x_r * y_l) + x_r * y_r
\end{aligned}$$

2.3 Deriving Recurrence

Suppose the multiplication function is called with two numbers, x and y . The algorithm would divide x into x_l and x_r . Similarly y would be divided into y_l and y_r recursively.

As we can see, we have performed 4 multiplications. The time complexity is $T(n) = 4 * T(n/2) + \Theta(n)$. However, the following terms have already been computed -

$$x_l * y_r + x_r * y_l = (x_l + x_r) * (y_l + y_r) - x_l * y_l - x_r * y_r$$

Therefore, we can subtract $T(n/2)$ from the earlier derived recurrence relation. The resulting derived recurrence relation is:

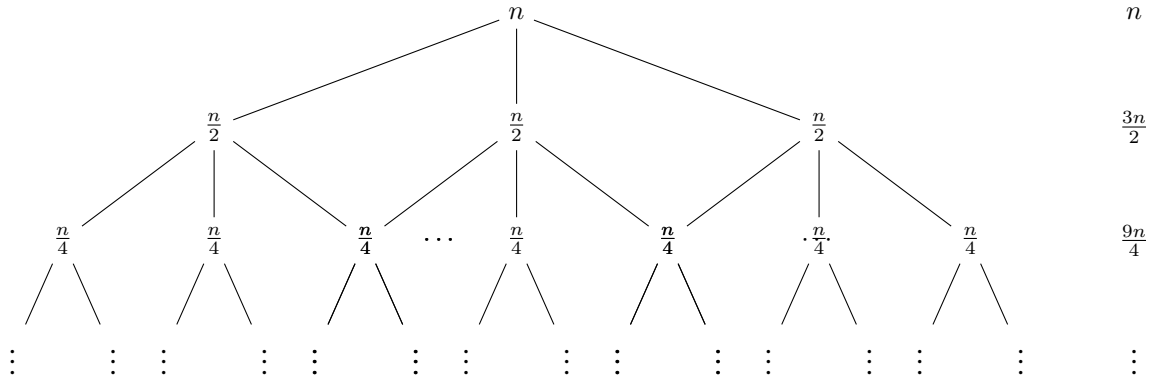
$$T(n) = 3T(n/2) + \Theta(n)$$

2.4 Solving Recurrence

Recurrences can be solved using substitution, recursion tree method and the master method. Here we will use the recursion tree method to solve the earlier derived recurrence relation:

$$T(n) = 3T(n/2) + \Theta(n)$$

If we begin to draw the tree, it would look something like the following. Due to the limited area on a page, we cannot draw the complete recursion tree.



In the first level, the term(s) sum(s) to n . In the second level, the terms sum to $\frac{3n}{2}$. In the third level, the terms sum to $\frac{9n}{4}$. The general formula for the sum of successive levels is the following:

$$\begin{aligned}
\sum_{i=1}^{\infty} \frac{3^i * n}{2^i} &= \frac{1/2(3^h - 1)}{2^{h+1} - 1} * n \\
\therefore &\approx n^{1.58}
\end{aligned}$$

2.5 Validation Using Master Theorem

We apply the Master Theorem to solve the same recurrence relation.

$$\begin{aligned}
T(n) &= 3T(n/2) + \Theta(n) \\
f(n) &= n^a \quad a = 3 \quad b = 2 \\
n^{\log_b a} &= n^{\log_2 3} \approx n^{1.58}
\end{aligned}$$

We know that $\Theta(n)$ is asymptotically smaller or equal to n . Therefore, we make the following computations:

$$\begin{aligned}
\epsilon &= 0.1 \\
n^{1.58-0.1} &= n^{1.48} \\
\lim_{n \rightarrow \infty} \frac{\Theta(n)}{n^{1.48}} &= 0
\end{aligned}$$

We can generalize that $f(n)$ is asymptotically smaller than $g(n)$. Therefore $T(n) = \Theta(n^{\log_2 3})$. This proves the result.