Fibonacci Sequence as Matrices: An Efficient Strategy to Count (1, 2)-Compositions of an Integer

Asif Hummam Rais — 13517099
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
asif@students.itb.ac.id

***Abstract*—Fibonacci numbers have peculiar relationship with combinatorics, i.e. they represent the sum of "shallow" diagonals in Pascal's triangle. A well-known Fibonacci-number-finding formula involves the exponentiation of φ, the golden ratio, which is represented as a floating point in computers, thus affecting the accuracy. The (1, 2)-composition of an integer *n* in combinatorics is defined as a way of writing *n* as the sum of 1's and 2's that could be solved by the sum of a combination series. This paper discusses a better approach in finding 1-2 composition of an integer *n-1* using matrix exponentiation algorithm to find *n*th Fibonacci number.**

***Keywords*—composition, divide and conquer algorithm, Fibonacci sequence, induction, recursion.**

I. Introduction

Composition is one of the branches of combinatorics that is defined as a way of writing an integer *n* as the sum of a sequence of strictly positive integers. It is proven that any positive integer *n* has 2^{n-1} distinct compositions with the sequence consisting of numbers ranging from 1 to *n*.

Unfortunately, it is much trickier to find the composition of a positive integer with the sequence consisting only of 1's and 2's — namely the (1, 2)-composition. A little approach would be finding the combinations of zero 2's, one 2's, and so forth, canonically representing the sum of "shallow" diagonals in Pascal's triangle.

On the other hand, Fibonacci numbers are strongly related to the golden ratio, φ, or 1.6180339887…, that strangely has interesting connection to the world we live in. The sum of shallow diagonals in Pascal's triangle also has a connection to Fibonacci sequence, which will be discussed later in this paper.

Much to our dismay, the well-known Fibonacci-number-finding mathematical formula has the golden ratio in it — represented as a floating point in computers. With computers having a limit to their maximum memory allocation, φ can't be represented by a floating point to the extent of an exact representation, as the golden ratio is an irrational number. Instead, to avoid the O(*n*) naïve approach in finding either the *n*th Fibonacci number or a dynamic programming approach in finding the number of (1, 2)-compositions, another approach of finding *n*th Fibonacci number via matrix multiplication to count the number of (1, 2)-compositions works better with the algorithm complexity of O(log *n*).

II. Theory

A. Combinations and Compositions

A combination is the way in selecting a number of elements from a set, in such that the order of element does not matter. For example, between three colors: red, green, blue, the combinations of two from said set are red-green, red-blue, and green-blue. Formally, a *k*-combination of a set *S* is the subset of *k* distinct elements of *S*. The number of *k*-combinations is equal to binomial coefficient and can be written mathematically as

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

and has the identity

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$$

A *k*-multicombination defines a way in selecting a number of elements from a set with different kind of elements, where selecting duplicates count as different multicombination, i.e. the number of elements from each kind of elements can be assumed to be infinite, but disregarding different orderings (e.g. {1,1,2} = {2, 1, 1}). For example, if you have three types of donuts ($n = 3$) to choose from and you want exactly two donuts ($k = 2$), the number of 2-multicombinations of 3 is 6. Formally the number of such *k*-multicombinations is denoted by

$$\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$$

A composition of integer *n* defines a way of writing *n* as the sum of positive integers. As composition differ by arrangement similar to permutation, two sequences with different order of numbers with same set of numbers define a different composition but are considered to be the same partition. The distinct compositions of any integer are finite, with negative integers having zero composition and 0 having one composition (i.e. the empty sequence). For example, the integer 4 has five compositions: 1+1+1+1, 1+1+2, 1+2+1, 2+1+1, and 2+2.

A *weak composition* of *n* is defined similar to composition, with the restricting numbers that make the sum of their sequence is *n* is non-negative. That is, the weak composition of an integer could be consisting of zero elements. Incidentally, the number of weak compositions of an integer is infinite. Succeeding zeroes at the end of a weak composition sequence is usually not considered, thus a weak composition can implicitly be defined as having infinite number of zeroes at the end of the sequence.

Formally, an *A(-restricted)-composition* of n defines a way of writing n as a sequence of numbers consisting of the elements of A.[1] Henceforth, a (1, 2)-composition of n is the composition of n with the sequence consisting only of x where $x \in (1, 2)$. For example, the integer 5 has eight (1, 2)-compositions: 1+1+1+1+1, 1+1+1+2, 1+1+2+1, 1+2+1+1, 2+1+1+1, 1+2+2, 2+1+2, and 2+2+1.

By convention, 0 has the number of compositions 1: the empty composition. Negative numbers have zero composition. For any n where $n \geq 1$, there are 2^{n-1} distinct compositions of n. The proof is quite straightforward:

Assume there is a line of n numbers of 1's where between two adjacent 1's is a box that can be placed with either a plus sign or a comma.

$$\left(\underbrace{1 \square 1 \square 1 \square 1 \ldots \square 1 \square 1}_{n}\right)$$

For example, if n is 6, one of the possible compositions is 1+2+3, that can be represented in the box-comma instance as (1, 1+1, 1+1+1). As there are n-1 number of boxes and the possible values of each box is either one of the 2, the number of compositions of n is 2^{n-1}.

B. Algorithmic Complexity

In computer science, an algorithm is defined as a set of operations or rules for the computer to execute in solving a problem. The efficiency of an algorithm depends on its complexity, that is synonymous to the metric of an algorithm process. Algorithmic complexity spans to time complexity and space complexity.

Time complexity is the amount of time needed by the algorithm to complete the set of rules in respect to the time-factor parameters, commonly estimated by counting how many elementary operations are being performed. Time complexity is usually depicted as either worst-case complexity, i.e. the worst possible time that is affected by a certain time-factor parameter input, and the average-case complexity, which is the average of time taken on a given size parameter input. Time complexity can be expressed explicitly as T(n), the exact operations being performed for a specific parameter n, or the big-O notation, O(n), that focuses on the asymptotic behavior of the complexity, i.e. when n is nearing infinity. Henceforth, a $T((n+3)^2 + 7n)$ complexity would only be depicted as $O(n^2)$ by the big-O.

Space complexity, on the other hand, is the amount of space or memory allocation needed by the algorithm to perform the operations. It can be expressed explicitly as S(n) or asymptoticly as O(n), similar to time complexity.

C. Recursion and Induction

Recursion is a phenomenon when a thing being defined is applied to its own definition by any form. In computer science or generally mathematics, a function or object will formally be classified as having recursive behavior if it has 1) a simple base case or a terminating case where it does not exhibit a recursion, i.e. the function call stops and returns a value when the parameter is within the base case, and 2) set of rules, in a way that the function will call itself by any form, where the parameter is getting closer to the base case.

A classic example of recursive behavior in a computer program is the factorial function, which is defined and written in C++ as

```
int Factorial (int n) {
    if (n == 0) return 1;
    else return n * Factorial (n - 1);
}
```

Above piece of code satisfies the recursion formal definition, as it possesses the base case that is defined when n is equal to 0 and the set of rules that calls the function itself (*Factorial*) with the parameter getting closer to the base case (n-1), thus is classified as exhibiting recursive behavior.

A common algorithm strategy namely divide-and-conquer also usually exhibits recursive behavior. The idea is to simplify the problem and divide it into varying subproblems, then solve the smaller problems using a recursive rule by calling itself. The base case is defined when the problem has become small enough to be handled the fastest, hence the algorithm will stop dividing the problem into subproblems and instead return a value.

For example, an efficient integer exponentiation algorithm applies divide-and-conquer, recursive strategy to yield an effective time complexity of O(log n) where n is the exponent, written in C++ as follows

```
int Power (int x, int n) {
    if (n == 0) return 1;
    if (n == 1) return x;
    int sub = Power (x, n/2);
    return sub * sub * Power (x, n%2);
}
```

Moreover, by recursive behavior analysis, we can prove a property $P(n)$ holds for every number n, if a finite number of n has been proven as the base case and the recursive behavior proves the next n for $P(n)$. This proof technique is called the mathematical induction, where the first step requires the base case proving that the property holds for a certain number, and the induction step proving that if the property holds for one natural number n, then it holds for $n+1$, practically proving that it works for all n where n is greater or equal to the base case.

Metaphorically speaking, induction is proving that we can climb the ladder as high as we can, as long as we can climb onto the bottom rung, the *base case*, and for each rung we can then climb to the rung exactly above the rung we're on, the *step*.[2]

D. Fibonacci Numbers

Fibonacci numbers, denoted as F_n, are numbers exhibiting recursive behavior that form a sequence, Fibonacci sequence, where each number is the sum of two previous numbers in the same sequence, starting from 0 at the zeroth index and 1 for the first index. Fibonacci number can also be formally expressed as

The base case,

$$F_0 = 0, F_1 = 1$$

And the recursive rule,

$$F_n = F_{n-1} + F_{n-2}$$

As such, the beginning of the sequence is thus, $(0,)\ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

Fibonacci numbers appear unexpectedly frequently in mathematics and have a strong connection to *the golden ratio*, φ, such that there exists a formula for finding the *n*th Fibonacci number expressed in terms of *n* and the golden ratio, and implies that the ratio of two consecutive Fibonacci tends to the golden ratio when *n* is nearing infinity, namely *Binet's formula*:

$$F_n = \frac{\varphi^n - \Psi^n}{\varphi - \Psi}$$

$$where\ \varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887 \ldots$$

$$and\ \Psi = \frac{1-\sqrt{5}}{2} = -\frac{1}{\varphi} \approx -0.6180339887 \ldots$$

E. Matrices

Matrix is a form of depicting a two-dimensional array of numbers, symbols, or expressions, usually written in box brackets or parentheses. A matrix has the number of row and column identity, i.e. an *n x m* matrix denotes a two-dimensional array with *n* rows and *m* columns. The element of a matrix ***M*** can be referred as m_{ij}, where *i* is the row index and *j* is the column index. For example, below is a 2 x 3 matrix ***A*** whose element a_{ij} equals to $i + j$.

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

Basic operations of matrices include addition, scalar multiplication, transposition, matrix multiplication, row operations, and submatrix obtaining. Here, we focus on matrix multiplication.

Matrix multiplication is a binary or two-parameter operation in such a way that for an *n* x *m* matrix **A** and an *m* x *p* matrix **B** being multiplied (order matters), the resulted product is an *n* x *p* matrix **AB** where each entry of **AB** is the summation of the *m* entries across a row of **A** that are multiplied with the *m* entries down a column of **B**.

Or more formally, the definition of matrix multiplication is that if **A** is an *n* x *m* matrix and **B** is an *m* x *p* matrix,

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}, \boldsymbol{B} = \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mp} \end{pmatrix}$$

then the matrix product **C** = **AB** is defined as an *n* x *p* matrix:

$$\boldsymbol{C} = \begin{pmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{np} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj}$$

$$for\ i = 1, 2, \ldots, n\ and\ j = 1, 2, \ldots, p$$

III. (1, 2)-Composition and Fibonacci Numbers

It is a bit trickier to count the number of (1, 2)-compositions of *n* instead of the number of general compositions of *n*. To start with, we can divide the (1, 2)-compositions of *n* into *n* - *k* cases as multicombinations using the stars-and-bars representation: the first case, $\binom{n}{0}$ is defining the number of (1, 2) compositions of *n* where it consists of zero 2's, the second case, $\binom{n-1}{1}$ is defining the number of (1, 2)-compositions of *n* where it consists of one 2's, and so on until the last case, $\binom{n-k}{k}$ is defining the number of compositions of *n* where it consists of *k* 2's. Hence, *k* is the floor of *n*/2 as there is no composition of *n* where the number of 2's exceeds *k*/2.

Henceforth, the function that returns the number of (1, 2)-compositions of *n* can be expressed formally as

$$f_n = \binom{n}{0} + \binom{n-1}{1} + \cdots + \binom{n-k}{k}$$

$$where\ k = \left\lfloor \frac{n}{2} \right\rfloor$$

or sophisticatedly,

$$f_n = \sum_{k=0}^{\left\lfloor \frac{n}{2} \right\rfloor} \binom{n-k}{k}$$

Furthermore, we elaborate the connection between Fibonacci sequence to the sum of shallow diagonals in Pascal's triangle that can be figured as follows

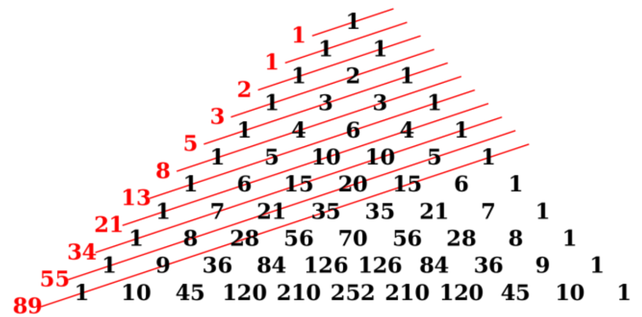

Figure 1. Fibonacci sequence and Pascal's triangle.

Now, we have the premise that the sum of shallow lesser diagonals with row index *n* in Pascal's triangle is the *n*+1th Fibonacci number[3], which is formally described as, and we want to show that

$$f_n = \sum_{k=0}^{\left\lfloor \frac{n}{2} \right\rfloor} \binom{n-k}{k} = F_{n+1}$$

The easiest way to prove something Fibonacci-sequence-related is probably to use induction theory. The proof is simpler if we include the terms where *k* is greater than *n* - *k*, i.e.

$$f_n = \sum_{k=0}^{n} \binom{n-k}{k} = F_{n+1}$$

and where $n - k < k$, the terms hold zero.

For the base cases, we can easily verify that

$$f_0 = \sum_{k=0}^{0} \binom{0-k}{k} = \binom{0}{0} = 1 = F_1$$

$$f_1 = \sum_{k=0}^{0} \binom{1-k}{k} = \binom{1}{0} = 1 = F_2$$

$(0! = 1$, therefore $\binom{0}{0} = \binom{1}{0} = 1)$

For the recursive behavior, it is satisfied that

$$f_n + f_{n+1} = \sum_{k=0}^{n} \binom{n-k}{k} + \sum_{k=0}^{n+1} \binom{n+1-k}{k}$$

by changing the left sum index $k \mapsto k - 1$ we got

$$\sum_{k=1}^{n+1} \binom{n+1-k}{k-1} + \sum_{k=0}^{n+1} \binom{n+1-k}{k}$$

pulling out the $k = 0$ term from the right sum we got

$$\sum_{k=1}^{n+1} \binom{n+1-k}{k-1} + \binom{n+1-0}{0} + \sum_{k=1}^{n+1} \binom{n+1-k}{k}$$

$$= 1 + \sum_{k=1}^{n+1} \binom{n+1-k}{k-1} + \sum_{k=1}^{n+1} \binom{n+1-k}{k}$$

using the combination property $\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$ we got

$$1 + \sum_{k=1}^{n+1} \binom{n+2-k}{k} - \sum_{k=1}^{n+1} \binom{n+1-k}{k} + \sum_{k=1}^{n+1} \binom{n+1-k}{k}$$

$$= 1 + \sum_{k=1}^{n+1} \binom{n+2-k}{k} = \binom{n+2-0}{0} + \sum_{k=1}^{n+1} \binom{n+2-k}{k}$$

$$= \sum_{k=0}^{n+1} \binom{n+2-k}{k} = f_{n+2} = f_n + f_{n+1}$$

thus completing the proof.

Trivially, another proof showing that $f_n = F_{n+1}$ using logical analysis on the recursive behavior is that we can describe the problem into a conjecture that defines as: if f_n is the number of (1, 2)-compositions of n, then we can conduct premises and recursive function behavior as follows:

- For $n = 0$, the only possible (1, 2)-composition is the empty composition. Thus, $f_0 = 1$.
- For $n = 1$, the only possible (1, 2)-composition is 1. Thus, $f_1 = 1$.
- For $n > 1$, we can either take a 1 as the first composition element such that the remaining n will be $n - 1$ or take 2 such that the remaining n will be $n - 2$. This way, we deduce that $f_n = f_{n-1} + f_{n-2}$.
- As f_n holds the recursive property of F_n, which is g_n where $g_n = g_{n-1} + g_{n-2}$, and with f_0 having the same value as F_1 and f_1 having the same value as F_2, we can safely conclude that $f_n = F_{n+1}$.

IV. Faster Fibonacci Algorithms

As we have found out that the number of (1, 2)-compositions of n, f_n is equals to the Fibonacci sequence F_{n+1}, we now shall conduct a deeper analysis on finding the most efficient and accurate algorithm to be executed on computer programs. We will begin by analyzing the crudest, *brute-force* algorithm, to the most efficient, *divide-and-conquer* matrix-Fibonacci algorithm, and show how each of the algorithm perform in terms of their respective algorithmic (i.e. time and space) complexity and perhaps return value accuracy.

A. Naïve Algorithm

Let's begin by going with the formal definition of Fibonacci, where $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$. Henceforth, there is a naïve approach for a computer program written in C++ to find the nth Fibonacci number using a recursive function that exactly tells the properties of a Fibonacci sequence as-is, which is

```
int SlowF (int n) {
     if (n == 0) return 0;
     if (n == 1) return 1;
     return SlowF (n - 1) + SlowF (n - 2);
}
```

With a little analysis, we can deduce that each of the function call will call another two of itself, with the parameter being passed is the number preceding the current parameter (i.e. $n - 1$ and $n - 2$) and ends only when the parameter input hits 0 or 1. By this means, *SlowF*(n) will call *SlowF*($n - 1$) and *SlowF*($n - 2$), and for each of the *SlowF*'s being called, it will call another two *SlowF*'s. Henceforth, the time complexity of this algorithm is $T(n) = T(n - 1) + T(n - 2)$, or in big-O notation, $O(2^n)$ as it keeps multiplying by two until the base case that is being reduced by one at a time, i.e. the exponential time complexity.

```
Calling SlowF(20) 1000 times: 62485 microseconds
Calling SlowF(21) 1000 times: 93774 microseconds
Calling SlowF(22) 1000 times: 187444 microseconds
Calling SlowF(23) 1000 times: 272041 microseconds
Calling SlowF(24) 1000 times: 453063 microseconds
Calling SlowF(25) 1000 times: 745666 microseconds
Calling SlowF(26) 1000 times: 1179692 microseconds
Calling SlowF(27) 1000 times: 1922371 microseconds
```

Figure 2. Every increase in n means approximately a double increase in time for *SlowF*(n) function calls.

As for the space complexity, seemingly not much is different. With each function call being stored in the function stack, the memory allocation for every *SlowF* function being called is probably also $O(2^n)$. But this might not be that simple, as the recursive calls are not computed in the same time, but sequentially. That means that *SlowF*($n - 2$) will only compute after *SlowF*($n - 1$). Hence, even though we create 2^n recursive calls, only n will be active at a time, meaning that the function stack space complexity of this algorithm is $O(n)$ at the worst.

B. Bottom-Up Dynamic Programming

The next faster algorithm we can consider is using *memorized brute-force*, i.e. the dynamic programming. Implementation is quite simple, there will be a *while-do* loop that stores the value of *cur* (F_{idx}), *b1* ($F_{\text{idx-1}}$), *b2* ($F_{\text{idx-2}}$), and *idx*:

```
int DpF (int n) {
     if (n == 0) return 0;
     int b2, b1 = 0, cur = 1, index = 1;
     while (index < n) {
           b2 = b1; b1 = cur;
           cur = b1 + b2; index++;
     }
     return cur;
}
```

The difference between bottom-up and top-down dynamic programming is that top-down is practically the *SlowF* algorithm with the addition of a memo, i.e. an array to store the value of F_n that has been calculated and returns that value instead of the function definition if the value is not NULL (undefined) thus increasing the memory allocation by O(n), while bottom-up finds the value of F_n from the bottom, F_1, thus is able to throw away the value of unneeded F_{n-3}, F_{n-4}, and so forth.

As we can see, there is only a *while-do* loop that runs depending on the n, i.e. $T(n) = n - 1$. Thus, for this algorithm, the complexity of *DpF*(n) is undoubtedly O(n) with a space complexity of O(1) as the memory allocation is constant.

```
Calling DpF(1000000) 1 times: 31283 microseconds
Calling DpF(2000000) 1 times: 31246 microseconds
Calling DpF(3000000) 1 times: 62482 microseconds
Calling DpF(4000000) 1 times: 80809 microseconds
Calling DpF(5000000) 1 times: 93685 microseconds
Calling DpF(6000000) 1 times: 109393 microseconds
Calling DpF(7000000) 1 times: 140592 microseconds
Calling DpF(8000000) 1 times: 156215 microseconds
```

Figure 3. Every increase in n means approximately a linear increase in time for *DpF*(n) function calls.

C. Binet's Formula

As Binet's formula is a well-known formula in finding the nth Fibonacci number, a faster Fibonacci-finding algorithm, also written in C++, now arises:

```
int BinetF (int n) {
     float phi = (1 + sqrt(5))/2;
     float res = pow(phi, n);
     res -= pow((-1/phi), n);
     res /= sqrt(5);
     return int(res);
}
```

For this algorithm, the time complexity depends on how the *pow* function works in C++. As the *pow* function uses a divide-and-conquer approach for the implementation (i.e. *pow(a,b) = pow(a,b/2) * pow(a,b/2) * pow(a,b%2)*), we can deduce that the time complexity of Binet's Formula depicted as *BinetF*(n) is at average and at worst O(log n).

The space complexity is pretty similar, by *BinetF* being called in parameter inputs of n, $n/2$, …, 0, the calls will be put into a function stack memory, which is O(log n) as the parameter keeps dividing by two until the base case, 0 or 1.

Unfortunately, Binet's Formula implementation in a computer program has a little caveat: as floating number is depicted in binary, where the mantissa and exponent are strictly producing the power of 2's, it is physically impossible for an entity to act as the exact copy of φ, an irrational number.[4] As a result, the reduced time complexity comes with a cost of accuracy — something we must avoid at any cost.

```
BinetF(28) = 317811, true F(28) = 317811
BinetF(29) = 514229, true F(29) = 514229
BinetF(30) = 832040, true F(30) = 832040
BinetF(31) = 1346269, true F(31) = 1346269
BinetF(32) = 2178309, true F(32) = 2178309
BinetF(33) = 3524579, true F(33) = 3524578
BinetF(34) = 5702889, true F(34) = 5702887
```

Figure 4. Inaccuracy for larger n in *BinetF*(n).

D. Matrix Exponentiation

There is a little trick in finding the nth Fibonacci number, or even generally back-calling recursive functions, by utilizing matrix exponentiation. The algorithm is quite straightforward, what drives it to efficiency the most is actually the mathematical theory behind recursive functions being depicted as matrices.

For Fibonacci sequence, suppose we have these matrices

$$\boldsymbol{A} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \boldsymbol{F} = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

if we perform **AF**, it will result as follows

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n + F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

if n equals to 1, then

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}$$

and if we multiply **A** to the left and the right side, then

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_3 \\ F_2 \end{pmatrix}$$

that means generally for a matrix exponentiation of $\mathbf{A}^n$

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

Going by this equation, we can implement an algorithm based on the properties of a matrix and divide-and-conquer strategy to find F_n with the efficiency and accuracy nearing perfect. To begin with, we will export the integer exponentiation at O(log n) to matrix exponentiation, as such the base cases of *MatrixPow*(*M, n*) will be defined as:

- For $n = 0$, returns the identity matrix, **I**.
- For $n = 1$, returns the matrix itself, **M**.
- For n > 1, returns the *MatrixPow*(*M, n/2*) * *MatrixPow*(*M, n/2*) * *MatrixPow*(*M, n%2*) where *%* is the modulo operation.

As such, the implementation of this algorithm is as follows

```
#define ROW 2
#define COL 2

struct Matrix {
  int Elmt[ROW][COL];
};

Matrix MatrixMul (Matrix M, Matrix N) {
  Matrix R;
  for (int i = 0; i < ROW; i++)
    for (int j = 0; j < COL; j++) {
      int sum = 0;
      for (int k = 0; k < COL; k++)
        sum += M.Elmt[i][k]*N.Elmt[k][j];
      R.Elmt[i][j] = sum;
    }
  return R;
}
```

```
Matrix MatrixPow (Matrix M, int n) {
  if (n == 0){
    Matrix Identity;
    Identity.Elmt[0][0] = 1;
    Identity.Elmt[0][1] = 0;
    Identity.Elmt[1][0] = 0;
    Identity.Elmt[1][1] = 1;
    return Identity;
  }
  if (n == 1) return M;
  Matrix sub = MatrixPow (M, n/2);
  sub = MatrixMul(sub, sub);
  sub = MatrixMul(MatrixPow(M, n%2), sub);
  return sub;
}

int Fibonacci (int n){
  if (n == 0) return 0;
  Matrix F;
  F.Elmt[0][0] = 1;
  F.Elmt[0][1] = 1;
  F.Elmt[1][0] = 1;
  F.Elmt[1][1] = 0;
  F = MatrixPow (F, n);
  return F.Elmt[1][0];
}
```

As this algorithm behaves similar to the *pow* algorithm, with an addition of the constant O(1) matrix multiplication, the time complexity is $T(n) = 1 + T(n/2)$, where $T(1) = T(0) = 1$. Henceforth, the algorithm runs in a logarithmic time, as such, the time complexity in the big-O notation is O(log n). The space complexity also is affected by the dynamically stored matrices which is allocated at O(log n) and the stack functions which is also allocated at O(log n), thus the space complexity in the big-O notation is also O(log n).

```
Calling Fibonacci(1000000) 1000 times: 156167 microseconds
Calling Fibonacci(2000000) 1000 times: 156258 microseconds
Calling Fibonacci(4000000) 1000 times: 171834 microseconds
Calling Fibonacci(8000000) 1000 times: 188747 microseconds
Calling Fibonacci(16000000) 1000 times: 187410 microseconds
Calling Fibonacci(32000000) 1000 times: 187456 microseconds
Calling Fibonacci(64000000) 1000 times: 203075 microseconds
```

Figure 4. O(log n) time complexity, logarithmically related.

```
Fibonacci(80) = 23416728348467685, true F(80) = 23416728348467685
Fibonacci(81) = 37889062373143906, true F(81) = 37889062373143906
Fibonacci(82) = 61305790721611591, true F(82) = 61305790721611591
Fibonacci(83) = 99194853094755497, true F(83) = 99194853094755497
Fibonacci(84) = 160500643816367088, true F(84) = 160500643816367088
Fibonacci(85) = 259695496911122585, true F(85) = 259695496911122585
Fibonacci(86) = 420196140727489673, true F(86) = 420196140727489673
Fibonacci(87) = 679891637638612258, true F(87) = 679891637638612258
```

Figure 5. Accuracy is not affected as it involves no floating point. The only limiter is the *int* size, which could be overcome by representing integers as *strings* or *BigInts (Java)*. Note that above picture is implemented using *long long* instead of *int*.

As such, the nth Fibonacci number can be found via matrix multiplication. And to count the number of (1, 2)-compositions of n, i.e. f_n can be solved by finding F_{n+1}.

V. Conclusion

Induction is a common mathematical proof technique that can be used in mainly to properties that exhibit recursive behaviors. The number of (1, 2)-compositions of n, i.e. ways n could be arranged with the sum of 1's and 2's where the order matters, or f_n can be proven by induction to have the same value as the $n+1$th Fibonacci number, F_{n+1}. That is,

$$f_n = F_{n+1}$$

Divide-and-conquer strategy is one of the most efficient strategy in solving most of programming problems, as it reduces the algorithmic complexity to as low as logarithmic time complexity. Utilizing divide-and-conquer in matrix exponentiation helps reducing the Fibonacci-number-finding function without sacrificing the accuracy. Time complexity for the provided solution is at worst O(log n).

VI. Appendix

Figure 1 is taken from RDBury's image upload on Commons. All images shown after **Figure 1** are taken personally by the author. The time calculation in microseconds is done in a machine with Intel i7-8550U processor, using chrono library coded in C++ and compiled using gcc version 6.3.0 (MinGW).

VII. Acknowledgment

The author would like to express gratitude to Dr. Judhi Santoso, M. Sc. as the lecturer of IF2120 Discrete Mathematics in the author's class. The author would also like to thank the Stack Overflow community that helped a lot with tutorials regarding testing the time complexity and induction theorems. The author also is grateful to the author's friends and family for always giving inspirations and the passion to never give up.

References

[1] Heubach, Silvia; Mansour, Toufik. "*Compositions of n with parts in a set*". Congressus Numerantium, 2004. 168: 33–51.

[2] Graham, Ronald L.; Knuth, Donald E.; Patashnik, Oren. *Concrete Mathematics - A foundation for computer science (2nd ed.)*. Reading, MA, USA: Addison-Wesley Professional, 1994, page 3 margins.

[3] Stanley, Richard. *Enumerative Combinatorics I (2nd ed.)*. Cambridge University, 2011, p. 121, Ex 1.35.

[4] Rojas, Raúl. "*The Z1: Architecture and Algorithms of Konrad Zuse's First Computer*". arXiv:1406.1886 article, 2014.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2018

Asif Hummam Rais
13517099