RECURSSION AND FIBONACCI SERIES

What is Recursion?

In computer science, recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem. Recursion solves such recursive problems by using functions that call themselves from within their own code. Recursion are mainly of two types depending on whether a function calls itself from within itself or more than one function call one another mutually. The first one is called **direct recursion** and another one is called **indirect recursion**.

The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

— Niklaus Wirth, Algorithms + Data Structures = Programs, 1976

Most computer programming languages support recursion by allowing a function to call itself from within its own code. Some functional programming languages (for instance, Clojure) do not define any looping constructs but rely solely on recursion to repeatedly call code. It is proved in computability theory that these recursive-only languages are Turing complete; this means that they are as powerful (they can be used to solve the same problems) as imperative languages based on control structures such as `while` and `for`. Repeatedly calling a function from within itself may cause the call stack to have a size equal to the sum of the input sizes of all involved calls. It follows that, for problems that can be solved easily by iteration, recursion is generally less efficient, and, for large problems, it is fundamental to use optimization techniques such as tail call optimization. A common algorithm design tactic is to divide a problem into sub-problems of the same type as the original, solve those sub-problems, and combine the results. This is often referred to as the ***divide-and-conquer method***; when combined with a ***lookup table*** that stores the results of previously solved sub-problems (to avoid solving them repeatedly and incurring extra computation time), it can be referred to as ***dynamic programming or memoization.***

Base case :

A recursive function definition has one or more *base cases*, meaning input(s) for which the function produces a result trivially (without recurring), and one or more

recursive cases, meaning input(s) for which the program recurs (calls itself). For example, the factorial function can be defined recursively by the equations 0! = 1 and, for all $n > 0$, $n! = n(n-1)!$. Neither equation by itself constitutes a complete definition; the first is the base case, and the second is the recursive case. Because the base case breaks the chain of recursion, it is sometimes also called the "terminating case".

The **base case**, or **halting case**, of a function is the problem that we know the answer to, that can be solved without any more recursive calls. The base case is what stops the recursion from continuing on forever. Every recursive function must have at least one base case (many functions have more than one).

The job of the recursive cases can be seen as breaking down complex inputs into simpler ones. In a properly designed recursive function, with each recursive call, the input problem must be simplified in such a way that eventually the base case must be reached. (Functions that are not intended to terminate under normal circumstances—for example, some system and server processes—are an exception to this.) Neglecting to write a base case, or testing for it incorrectly, can cause an infinite loop.

For some functions (such as one that computes the series for **e = 1/0! + 1/1! + 1/2! + 1/3! + ...**) there is not an obvious base case implied by the input data; for these one may add a parameter (such as the number of terms to be added, in our series example) to provide a 'stopping criterion' that establishes the base case. Such an example is more naturally treated by **corecursion**, where successive terms in the output are the partial sums; this can be converted to a recursion by using the indexing parameter to say "compute the *n*th term (*n*th partial sum)".

Recursive Data Types :

Many computer programs must process or generate an arbitrarily large quantity of data. Recursion is a technique for representing data whose exact size is unknown to the programmer: the programmer can specify this data with a self-referential definition. There are two types of self-referential definitions: inductive and coinductive definitions.

Inductively defined data

An inductively defined recursive data definition is one that specifies how to construct instances of the data. For example, linked lists can be defined inductively (here, using Haskell syntax):

```
data ListOfStrings = EmptyList | Cons String ListOfStrings
```

The code above specifies a list of strings to be either empty, or a structure that contains a string and a list of strings. The self-reference in the definition permits the construction of lists of any (finite) number of strings.

Another example of inductive definition is the natural numbers (or positive integers):

```
A natural number is either 1 or n+1, where n is a natural number.
```

Similarly recursive definitions are often used to model the structure of expressions and statements in programming languages. Language designers often express grammars in a syntax such as **Backus–Naur form**; here is such a grammar, for a simple language of arithmetic expressions with multiplication and addition:

```
<expr> ::= <number>
         | (<expr> * <expr>)
         | (<expr> + <expr>)
```

This says that an expression is either a number, a product of two expressions, or a sum of two expressions. By recursively referring to expressions in the second and third lines, the grammar permits arbitrarily complicated arithmetic expressions such as `(5 * ((3 * 6) + 8))`, with more than one product or sum operation in a single expression.

Coinductively defined data and corecursion

A coinductive data definition is one that specifies the operations that may be performed on a piece of data; typically, self-referential coinductive definitions are used for data structures of infinite size.

A coinductive definition of infinite streams of strings, given informally, might look like this :

```
A stream of strings is an object s such that:
 head(s) is a string, and
 tail(s) is a stream of strings.
```

This is very similar to an inductive definition of lists of strings; the difference is that this definition specifies how to access the contents of the data structure—namely, via the accessor functions head and tail—and what those contents may be, whereas the inductive definition specifies how to create the structure and what it may be created from.

Corecursion is related to coinduction, and can be used to compute particular instances of (possibly) infinite objects. As a programming technique, it is used most often in the context of lazy programming languages, and can be preferable to recursion when the desired size or precision of a program's output is unknown. In such cases the program requires both a definition for an infinitely large (or infinitely precise) result, and a mechanism for taking a finite portion of that result. The problem of computing the first n prime numbers is one that can be solved with a corecursive program .

Types Of Recursion

Single recursion and multiple recursion:

Recursion that contains only a single self-reference is known as single recursion, while recursion that contains multiple self-references is known as multiple recursion. Standard examples of single recursion include list traversal, such as in a linear search, or computing the factorial function, while standard examples of multiple recursion include tree traversal, such as in a depth-first search.

Single recursion is often much more efficient than multiple recursion, and can generally be replaced by an iterative computation, running in linear time and requiring constant space. Multiple recursion, by contrast, may require exponential time and space, and is more fundamentally recursive, not being able to be replaced by iteration without an explicit stack.

Multiple recursion can sometimes be converted to single recursion (and, if desired, thence to iteration). For example, while computing the Fibonacci sequence naively entails multiple iteration, as each value requires two previous values, it can be computed by single recursion by passing two successive values as parameters. This is more naturally framed as corecursion, building up from the initial values, while tracking two successive values at each step – see corecursion: examples. A more sophisticated example involves using a threaded binary tree, which allows iterative tree traversal, rather than multiple recursion.

Indirect recursion:

Most basic examples of recursion, and most of the examples presented here, demonstrate ***direct recursion***, in which a function calls itself. *Indirect* recursion occurs when a function is called not by itself but by another function that it called (either directly or indirectly). For example, if *f* calls *f,* that is direct recursion, but if *f* calls *g* which calls *f,* then that is indirect recursion of *f.* Chains of three or more functions are possible; for example, function 1 calls function 2, function 2 calls function 3, and function 3 calls function 1 again.

Indirect recursion is also called mutual recursion, which is a more symmetric term, though this is simply a difference of emphasis, not a different notion. That is, if *f* calls *g* and then *g* calls *f,* which in turn calls *g* again, from the point of view of *f* alone, *f* is indirectly recursing, while from the point of view of *g* alone, it is indirectly recursing, while from the point of view of both, *f* and *g* are mutually recursing on each other. Similarly a set of three or more functions that call each other can be called a set of mutually recursive functions.

Anonymous recursion:

Recursion is usually done by explicitly calling a function by name. However, recursion can also be done via implicitly calling a function based on the current context, which is particularly useful for anonymous functions, and is known as anonymous recursion. It is possible to create recursive anonymous functions in JavaScript. Usually we need a name for a function to recursively refer to itself, but since anonymous functions have no name, JavaScript provides arguments.

Structural versus generative recursion:

Some authors classify recursion as either "structural" or "generative". The distinction is related to where a recursive procedure gets the data that it works on, and how it processes that data:

> *Functions that consume structured data typically decompose their arguments into their immediate structural components and then process those components. If one of the immediate components belongs to the same class of data as the input, the function is recursive. For that reason, we refer to these functions as (STRUCTURALLY) RECURSIVE FUNCTIONS.*
>
> ***— Felleisen, Findler, Flatt, and Krishnaurthi, How to Design Programs, 2001***

Thus, the defining characteristic of a structurally recursive function is that the argument to each recursive call is the content of a field of the original input. Structural recursion includes nearly all tree traversals, including XML processing, binary tree creation and search, etc. By considering the algebraic structure of the natural numbers (that is, a natural number is either zero or the successor of a natural number), functions such as factorial may also be regarded as structural recursion.

Generative recursion is the alternative:

> **Many well-known recursive algorithms generate an entirely new piece of data from the given data and recur on it. *HtDP* (*How to Design Programs*) refers to this kind as generative recursion. Examples of generative recursion include:**

gcd, quicksort, binary search, mergesort, Newton's method, fractals, and adaptive integration.

— Matthias Felleisen, Advanced Functional Programming, 2002[7]

This distinction is important in proving termination of a function.

- All structurally recursive functions on finite (inductively defined) data structures can easily be shown to terminate, via structural induction: intuitively, each recursive call receives a smaller piece of input data, until a base case is reached.
- Generatively recursive functions, in contrast, do not necessarily feed smaller input to their recursive calls, so proof of their termination is not necessarily as simple, and avoiding infinite loops requires greater care. These generatively recursive functions can often be interpreted as corecursive functions – each step generates the new data, such as successive approximation in Newton's method – and terminating this corecursion requires that the data eventually satisfy some condition, which is not necessarily guaranteed.
- In terms of loop variants, structural recursion is when there is an obvious loop variant, namely size or complexity, which starts off finite and decreases at each recursive step.
- By contrast, generative recursion is when there is not such an obvious loop variant, and termination depends on a function, such as "error of approximation" that does not necessarily decrease to zero, and thus termination is not guaranteed without further analysis.

Difference Between Recursion And Iteration

Recursion and iteration are equally expressive: recursion can be replaced by iteration with an explicit call stack, while iteration can be replaced with tail recursion. Which approach is preferable depends on the problem under consideration and the language used. In imperative programming, iteration is preferred, particularly for simple recursion, as it avoids the overhead of function calls and call stack management, but recursion is generally used for multiple recursion. By contrast, in functional languages recursion is preferred, with tail recursion optimization leading to little overhead. Implementing an algorithm using iteration may not be easily achievable.

Compare the templates to compute x_n defined by $x_n = f(n, x_{n-1})$ from x_{base}:

```
function recursive(n)
    if n == base
        return x_base
    else
        return f(n, recursive(n-1))
```

```
function iterative(n)
    x = x_base
    for i = base+1 to n
        x = f(i, x)
    return x
```

For an imperative language the overhead is to define the function, and for a functional language the overhead is to define the accumulator variable x.

For example, a factorial function may be implemented iteratively in C by assigning to a loop index variable and accumulator variable, rather than by passing arguments and returning values by recursion:

```c
unsigned int factorial(unsigned int n) {
  unsigned int product = 1; // empty product is 1
  while (n) {
    product *= n;
    --n;
  }
  return product;
}
```

Expressive power

Most programming languages in use today allow the direct specification of recursive functions and procedures. When such a function is called, the program's runtime environment keeps track of the various instances of the function (often using a call stack, although other methods may be used). Every recursive function can be transformed into an iterative function by replacing recursive calls with iterative control constructs and simulating the call stack with a stack explicitly managed by the program.

Conversely, all iterative functions and procedures that can be evaluated by a computer (see Turing completeness) can be expressed in terms of recursive

functions; iterative control constructs such as while loops and for loops are routinely rewritten in recursive form in functional languages.[11][12] However, in practice this rewriting depends on tail call elimination, which is not a feature of all languages. C, Java, and Python are notable mainstream languages in which all function calls, including tail calls, may cause stack allocation that would not occur with the use of looping constructs; in these languages, a working iterative program rewritten in recursive form may overflow the call stack, although tail call elimination may be a feature that is not covered by a language's specification, and different implementations of the same language may differ in tail call elimination capabilities.

Performance issues :

In languages (such as C and Java) that favor iterative looping constructs, there is usually significant time and space cost associated with recursive programs, due to the overhead required to manage the stack and the relative slowness of function calls; in functional languages, a function call (particularly a tail call) is typically a very fast operation, and the difference is usually less noticeable.

As a concrete example, the difference in performance between recursive and iterative implementations of the "factorial" example above depends highly on the compiler used. In languages where looping constructs are preferred, the iterative version may be as much as several orders of magnitude faster than the recursive one. In functional languages, the overall time difference of the two implementations may be negligible; in fact, the cost of multiplying the larger numbers first rather than the smaller numbers (which the iterative version given here happens to do) may overwhelm any time saved by choosing iteration.

Stack space :

In some programming languages, the maximum size of the call stack is much less than the space available in the heap, and recursive algorithms tend to require more stack space than iterative algorithms. Consequently, these languages sometimes place a limit on the depth of recursion to avoid stack overflows; Python is one such language. Note the caveat below regarding the special case of tail recursion.

Vulnerability :

Because recursive algorithms can be subject to stack overflows, they may be vulnerable to pathological or malicious input. Some malware specifically targets a program's call stack and takes advantage of the stack's inherently recursive

nature. Even in the absence of malware, a stack overflow caused by unbounded recursion can be fatal to the program, and exception handling logic may not prevent the corresponding process from being terminated.

Multiply recursive problems :

Multiply recursive problems are inherently recursive, because of prior state they need to track. One example is tree traversal as in depth-first search; though both recursive and iterative methods are used, they contrast with list traversal and linear search in a list, which is a singly recursive and thus naturally iterative method. Other examples include divide-and-conquer algorithms such as Quicksort, and functions such as the Ackermann function. All of these algorithms can be implemented iteratively with the help of an explicit stack, but the programmer effort involved in managing the stack, and the complexity of the resulting program, arguably outweigh any advantages of the iterative solution.

Refactoring recursion :

Recursive algorithms can be replaced with non-recursive counterparts. One method for replacing recursive algorithms is to simulate them using heap memory in place of stack memory. An alternative is to develop a replacement algorithm entirely based on non-recursive methods, which can be challenging. For example, recursive algorithms for matching wildcards, such as Rich Salz' wildmat algorithm, were once typical. Non-recursive algorithms for the same purpose, such as the Krauss matching wildcards algorithm, have been developed to avoid the drawbacks of recursion and have improved only gradually based on techniques such as collecting tests and profiling performance.

FIBONACCI SERIES

Fibonacci coding encodes an integer into binary number using Fibonacci Representation of the number. The idea is based on Zeckendorf's Theorem which states that every positive integer can be written uniquely as a sum of distinct non-neighboring Fibonacci numbers (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..).

The Fibonacci code word for a particular integer is exactly the integer's Zeckendorf representation with the order of its digits reversed and an additional "1" appended to the end. The extra 1 is appended to indicate the end of code (Note that the code never contains two consecutive 1s as per Zeckendorf's Theorem. The representation uses Fibonacci numbers starting from 1 (2'nd

Fibonacci Number). So the Fibonacci Numbers used are 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141, …….

The Fibonacci sequence is a type series where each number is the sum of the two that precede it. It starts from 0 and 1 usually. The Fibonacci sequence is given by 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, and so on. The numbers in the Fibonacci sequence are also called **Fibonacci numbers**. Any Fibonacci number can be calculated using the golden ratio, $F_n = (\Phi^n - (1-\Phi)^n) / \sqrt{5}$, Here φ is the golden ratio and Φ ≈ 1.618034. 2) The ratio of successive Fibonacci numbers is called the "golden ratio". Let A and B be the two consecutive numbers in the Fibonacci sequence.

```
Input: n = 1
Output: 11
1 is first Fibonacci number in this representation
and an extra 1 is appended at the end.

Input:  n = 11
Output: 001011
11 is sum of 8 and 3.  The last 1 represents extra 1
that is always added. A 1 before it represents 8. The
third 1 (from beginning) represents 3.
```

RELATION TO THE GOLDEN RATIO

Closed-form expression :

Like every sequence defined by a linear recurrence with constant coefficients, the Fibonacci numbers have a closed-form expression. It has become known as Binet's formula, named after French mathematician Jacques Philippe Marie Binet, though it was already known by Abraham de Moivre and Daniel Bernoulli:

$$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi} = \frac{\varphi^n - \psi^n}{\sqrt{5}},$$

where

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803\,39887\ldots$$

is the golden ratio, and ψ is its conjugate:[23]

$$\psi = \frac{1-\sqrt{5}}{2} = 1 - \varphi = -\frac{1}{\varphi} \approx -0.61803\,39887\ldots.$$

Since $\psi = -\varphi^{-1}$, this formula can also be written as

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}.$$

To see the relation between the sequence and these constants,[24] note that φ and ψ are both solutions of the equation

$$x^2 = x + 1 \quad \text{and, thus,} \quad x^n = x^{n-1} + x^{n-2},$$

so the powers of φ and ψ satisfy the Fibonacci recursion. In other words,

$$\varphi^n = \varphi^{n-1} + \varphi^{n-2}$$

and

$$\psi^n = \psi^{n-1} + \psi^{n-2}.$$

It follows that for any values a and b, the sequence defined by

$$U_n = a\varphi^n + b\psi^n$$

satisfies the same recurrence.

$$U_n = a\varphi^n + b\psi^n = a(\varphi^{n-1} + \varphi^{n-2}) + b(\psi^{n-1} + \psi^{n-2}) = a\varphi^{n-1} + b\psi^{n-1} + a\varphi^{n-2} + b\psi^{n-2} = U_{n-1} + U_{n-2}$$

If a and b are chosen so that $U_0 = 0$ and $U_1 = 1$ then the resulting sequence U_n must be the Fibonacci sequence. This is the same as requiring a and b satisfy the system of equations:

$$\begin{cases} a + b = 0 \\ \varphi a + \psi b = 1 \end{cases}$$

which has solution

$$a = \frac{1}{\varphi - \psi} = \frac{1}{\sqrt{5}}, \quad b = -a,$$

producing the required formula.

Taking the starting values U_0 and U_1 to be arbitrary constants, a more general solution is:

$$U_n = a\varphi^n + b\psi^n$$

where

$$a = \frac{U_1 - U_0\psi}{\sqrt{5}}$$

$$b = \frac{U_0\varphi - U_1}{\sqrt{5}}.$$

Computation by rounding :

Since

$$\left|\frac{\psi^n}{\sqrt{5}}\right| < \frac{1}{2}$$

for all $n \geq 0$, the number F_n is the closest integer to $\frac{\varphi^n}{\sqrt{5}}$. Therefore, it can be found by rounding, using the nearest integer function:

$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rceil, \ n \geq 0.$$

In fact, the rounding error is very small, being less than 0.1 for $n \geq 4$, and less than 0.01 for $n \geq 8$.

Fibonacci numbers can also be computed by truncation, in terms of the floor function:

$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor, \ n \geq 0.$$

As the floor function is monotonic, the latter formula can be inverted for finding the index $n(F)$ of the smallest Fibonacci number that is not less than a positive integer F:

$$n(F) = \left\lceil \log_\varphi \left(F \cdot \sqrt{5} - \frac{1}{2} \right) \right\rceil,$$

where $\log_\varphi(x) = \ln(x)/\ln(\varphi) = \log_{10}(x)/\log_{10}(\varphi)$, $\ln(\varphi) = 0.481211\ldots$, and $\log_{10}(\varphi) = 0.208987\ldots$

Magnitude :

Since F_n is asymptotic to $\varphi^n/\sqrt{5}$, the number of digits in F_n is asymptotic to $n \log_{10} \varphi \approx 0.2090\, n$. As a consequence, for every integer $d > 1$ there are either 4 or 5 Fibonacci numbers with d decimal digits.

More generally, in the base b representation, the number of digits in F_n is asymptotic to $n \log_b \varphi$.

Limit of consecutive quotients :

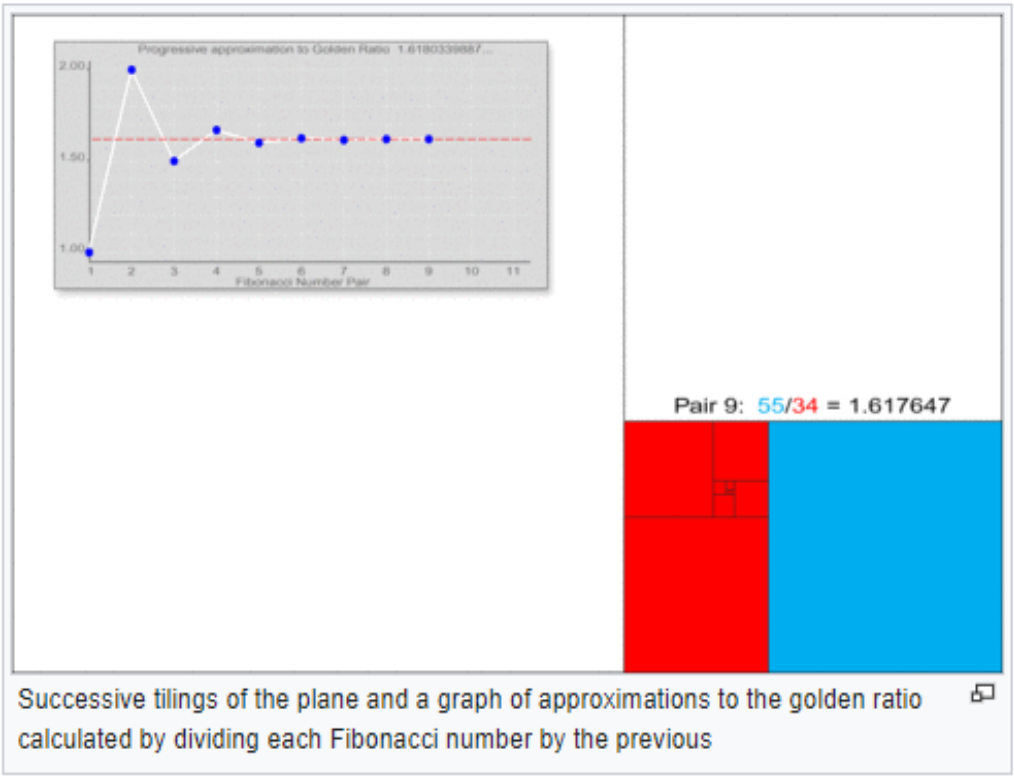
Johannes Kepler observed that the ratio of consecutive Fibonacci numbers converges. He wrote that "as 5 is to 8 so is 8 to 13, practically, and as 8 is to 13, so is 13 to 21 almost", and concluded that these ratios approach the golden ratio

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \varphi.$$

This convergence holds regardless of the starting values U0 and U1 unless $U_1 = -U_0/\varphi$

. This can be verified using Binet's formula. For example, the initial values 3 and 2 generate the sequence 3, 2, 5, 7, 12, 19, 31, 50, 81, 131, 212, 343, 555, ... The ratio of consecutive terms in this sequence shows the same convergence towards the golden ratio.

In general, $\lim_{n\to\infty} \frac{F_{n+m}}{F_n} = \varphi^m$, because the ratios between consecutive Fibonacci numbers approaches



Successive tilings of the plane and a graph of approximations to the golden ratio calculated by dividing each Fibonacci number by the previous

Decomposition of powers :

Since the golden ratio satisfies the equation

$$\varphi^2 = \varphi + 1,$$

this expression can be used to decompose higher powers φ^n

as a linear function of lower powers, which in turn can be decomposed all the way down to a linear combination of φ^n and 1. The resulting recurrence relationships yield Fibonacci numbers as the linear coefficients:

$$\varphi^n = F_n\varphi + F_{n-1}.$$

This equation can be proved by induction on $n \geq 1$:

$$\varphi^{n+1} = \left(F_n\varphi + F_{n-1}\right)\varphi = F_n\varphi^2 + F_{n-1}\varphi = F_n\left(\varphi + 1\right) + F_{n-1}\varphi = \left(F_n + F_{n-1}\right)\varphi + F_n = F_{n+1}\varphi + F_n.$$

For $\psi = -1/\varphi$, it is also the case that $\psi^2 = \psi + 1$ and it is also the case that

$$\psi^n = F_n\psi + F_{n-1}.$$

These expressions are also true for $n < 1$ if the Fibonacci sequence F_n is extended to negative integers using the Fibonacci rule

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

APPLICATIONS

MATHEMATICS :

The Fibonacci numbers occur in the sums of "shallow" diagonals in Pascal's triangle (see binomial coefficient):[59]

The generating function can be expanded into

$$\frac{x}{1-x-x^2} = x + x^2(1+x) + x^3(1+x)^2 + \cdots + x^{k+1}(1+x)^k + \cdots = \sum_{n=0}^{\infty} F_n x^n$$

and collecting like terms of x^n, we have the identity

$$F_n = \sum_{k=0}^{\left\lfloor \frac{n-1}{2} \right\rfloor} \binom{n-k-1}{k}.$$

To see how the formula is used, we can arrange the sums by the number of terms present:

$$
\begin{aligned}
5 &= 1+1+1+1+1 \\
&= 2+1+1+1 \quad = 1+2+1+1 = 1+1+2+1 = 1+1+1+2 \\
&= 2+2+1 \quad = 2+1+2 \quad = 1+2+2
\end{aligned}
$$

which is

$$\binom{5}{0} + \binom{4}{1} + \binom{3}{2}$$

where we are choosing the positions of *k* twos from *n-k-1* terms.

These numbers also give the solution to certain enumerative problems,[60] the most common of which is that of counting the number of ways of writing a given number *n* as an ordered sum of 1s and 2s (called compositions); there are F_{n+1} ways to do this (equivalently, it's also the number of domino tilings of the 2 x n rectangle). For example, there are $F_{5+1} = F_6 = 8$ ways one can climb a staircase of 5 steps, taking one or two steps at a time:

$$
\begin{aligned}
5 &= 1+1+1+1+1 = 2+1+1+1 = 1+2+1+1 = 1+1+2+1 = 2+2+1 \\
&= 1+1+1+2 \quad = 2+1+2 \quad = 1+2+2
\end{aligned}
$$

The figure shows that 8 can be decomposed into 5 (the number of ways to climb 4 steps, followed by a single-step) plus 3 (the number of ways to climb 3 steps, followed by a double-step). The same reasoning is applied recursively until a single step, of which there is only one way to climb.

The Fibonacci numbers can be found in different ways among the set of binary strings, or equivalently, among the subsets of a given set.

- The number of binary strings of length *n* without consecutive 1s is the Fibonacci number F_{n+2}. For example, out of the 16 binary strings of length 4, there are $F_6 = 8$ without consecutive 1s – they are 0000, 0001, 0010, 0100, 0101, 1000, 1001, and 1010. Such strings are the binary representations of Fibbinary numbers. Equivalently, F_{n+2} is the number of subsets *S* of {1, ..., *n*} without consecutive integers, that is, those *S* for which $\{i, i+1\} \not\subseteq S$ for every *i*. A bijection with the sums to *n+1* is to replace *1* with *0* and *2* with *10*, and drop the last zero.

- The number of binary strings of length n without an odd number of consecutive 1s is the Fibonacci number F_{n+1}. For example, out of the 16 binary strings of length 4, there are $F_5 = 5$ without an odd number of consecutive 1s – they are 0000, 0011, 0110, 1100, 1111. Equivalently, the number of subsets S of {1, ..., n} without an odd number of consecutive integers is F_{n+1}. A bijection with the sums to n is to replace *1* with *0* and *2* with *11*.
- The number of binary strings of length n without an even number of consecutive 0s or 1s is $2F_n$. For example, out of the 16 binary strings of length 4, there are $2F_4 = 6$ without an even number of consecutive 0s or 1s – they are 0001, 0111, 0101, 1000, 1010, 1110. There is an equivalent statement about subsets.
- Yuri Matiyasevich was able to show that the Fibonacci numbers can be defined by a Diophantine equation, which led to his solving Hilbert's tenth problem.[61]
- The Fibonacci numbers are also an example of a complete sequence. This means that every positive integer can be written as a sum of Fibonacci numbers, where any one number is used once at most.
- Moreover, every positive integer can be written in a unique way as the sum of *one or more* distinct Fibonacci numbers in such a way that the sum does not include any two consecutive Fibonacci numbers. This is known as Zeckendorf's theorem, and a sum of Fibonacci numbers that satisfies these conditions is called a Zeckendorf representation. The Zeckendorf representation of a number can be used to derive its Fibonacci coding.
- Starting with 5, every second Fibonacci number is the length of the hypotenuse of a right triangle with integer sides, or in other words, the largest number in a Pythagorean triple, obtained from the formula

$$(F_n F_{n+3})^2 + (2F_{n+1} F_{n+2})^2 = {F_{2n+3}}^2.$$

- The sequence of Pythagorean triangles obtained from this formula has sides of lengths (3,4,5), (5,12,13), (16,30,34), (39,80,89), ... The middle side of each of these triangles is the sum of the three sides of the preceding triangle.
- The Fibonacci cube is an undirected graph with a Fibonacci number of nodes that has been proposed as a network topology for parallel computing.
- Fibonacci numbers appear in the ring lemma, used to prove connections between the circle packing theorem and conformal maps.

COMPUTER SCIENCE :

- The Fibonacci numbers are important in computational run-time analysis of Euclid's algorithm to determine the greatest common divisor of two integers: the worst case input for this algorithm is a pair of consecutive Fibonacci numbers.[64]
- Fibonacci numbers are used in a polyphase version of the merge sort algorithm in which an unsorted list is divided into two lists whose lengths correspond to sequential Fibonacci numbers – by dividing the list so that the two parts have lengths in the approximate proportion φ. A tape-drive implementation of the polyphase merge sort was described in *The Art of Computer Programming*.
- A Fibonacci tree is a binary tree whose child trees (recursively) differ in height by exactly 1. So it is an AVL tree, and one with the fewest nodes for a given height — the "thinnest" AVL tree. These trees have a number of vertices that is a Fibonacci number minus one, an important fact in the analysis of AVL trees.[65]
- Fibonacci numbers are used by some pseudorandom number generators.
- Fibonacci numbers arise in the analysis of the Fibonacci heap data structure.
- A one-dimensional optimization method, called the Fibonacci search technique, uses Fibonacci numbers.[66]
- The Fibonacci number series is used for optional lossy compression in the IFF 8SVX audio file format used on Amiga computers. The number series compands the original audio wave similar to logarithmic methods such as μ-law.[
- Some Agile teams use a modified series called the "Modified Fibonacci Series" in planning poker, as an estimation tool. Planning Poker is a formal part of the Scaled Agile Framework.
- Fibonacci coding
- NegaFibonacci coding

NATURE :

Fibonacci sequences appear in biological settings, such as branching in trees, arrangement of leaves on a stem, the fruitlets of a pineapple, the flowering of artichoke, an uncurling fern and the arrangement of a pine cone, and the family tree of honeybees.Kepler pointed out the presence of the Fibonacci sequence in nature, using it to explain the (golden ratio-related) pentagonal form of some flowers. Field daisies most often have petals in counts of Fibonacci numbers.In 1754, Charles Bonnet discovered that the spiral phyllotaxis of plants were frequently expressed in Fibonacci number series.

Przemysław Prusinkiewicz advanced the idea that real instances can in part be understood as the expression of certain algebraic constraints on free groups, specifically as certain Lindenmayer grammars.

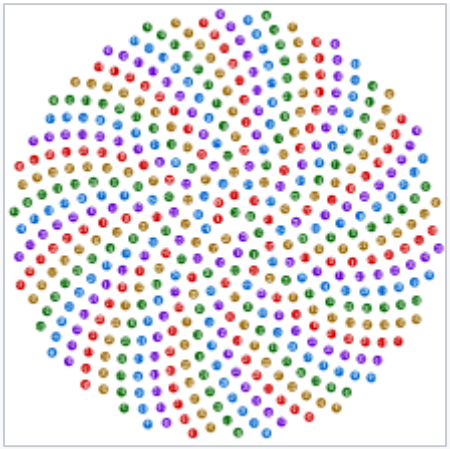


Illustration of Vogel's model for $n = 1 \ldots 500$

A model for the pattern of florets in the head of a sunflower was proposed by Helmut Vogel [de] in 1979. This has the form

where n is the index number of the floret and c is a constant scaling factor; the florets thus lie on Fermat's spiral. The divergence angle, approximately 137.51°, is the golden angle, dividing the circle in the golden ratio. Because this ratio is irrational, no floret has a neighbor at exactly the same angle from the center, so the florets pack efficiently. Because the rational approximations to the golden ratio are of the form $F(j):F(j + 1)$, the nearest neighbors of floret number n are those at $n \pm F(j)$ for some index j, which depends on r, the distance from the center. Sunflowers and similar flowers most commonly have spirals of florets in clockwise and counter-clockwise directions in the amount of adjacent Fibonacci numbers,[80] typically counted by the outermost range of radii.[81]

Fibonacci numbers also appear in the pedigrees of idealized honeybees, according to the following rules:

- If an egg is laid by an unmated female, it hatches a male or drone bee.
- If, however, an egg was fertilized by a male, it hatches a female.

Thus, a male bee always has one parent, and a female bee has two. If one traces the pedigree of any male bee (1 bee), he has 1 parent (1 bee), 2 grandparents, 3 great-grandparents, 5 great-great-grandparents, and so on. This sequence of numbers of parents is the Fibonacci sequence. The number of ancestors at each level, F_n, is the number of female ancestors, which is F_{n-1}, plus the number of male ancestors, which is F_{n-2}.[82] This is under the unrealistic assumption that the ancestors at each level are otherwise unrelated.