

Chapter 2

Recursive Call



Abstract Recursive call may appear difficult when you initially approach, but it cannot be overlooked when it comes to studying algorithms. It bears a close connection with mathematical induction, and those who once had a frustrating experience with mathematical induction in the past may feel intimidated, but there is nothing to fear. It is my opinion that anyone who understands grammatically correct your native language and knows how to count natural numbers can master the use of recursive calls. In this chapter, we will attempt to understand recursive calls and their correct usage by considering two themes: the “Tower of Hanoi ” and “Fibonacci numbers.”.

— What you will learn: —

- Recursive call
- Tower of Hanoi
- Fibonacci numbers
- Divide-and-conquer
- Dynamic programming

2.1 Tower of Hanoi

Under a dome that marks the center of the world in a great temple in Benares, India, there are three rods. In the beginning of the world, God piled 64 discs of pure gold onto one of the rods in descending order of size from bottom to top. Monks spend days and nights transferring discs from rod number 1 to rod number 3 following a certain rule. The world is supposed to end as soon as the monks finish moving all the discs. The rules for moving the discs around are quite simple. In the first place, only one disc can be moved at a time. Moreover, a disc cannot be placed upon a smaller one. How can the monks move the disks in an efficient manner? Moreover, how many times would the monks have to move the discs around to finish moving all discs?

This puzzle is known as the Tower of Hanoi. It was originally proposed in 1883 by French mathematician E. Lucas. An example of the Tower of Hanoi available in

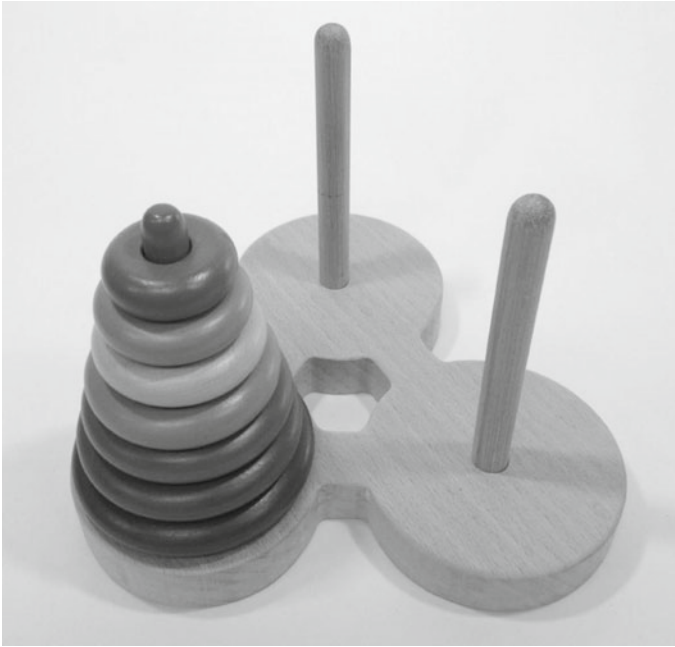


Fig. 2.1 Tower of Hanoi with 7 discs

the market is shown in Fig. 2.1. This example has seven discs. In this chapter, we will consider algorithms to move discs and the number of moves $H(n)$, where n is the number of discs.

We simplify descriptions by denoting the transfer of disc i from rod j to rod k as $(i; j \rightarrow k)$. Consider that the discs are enumerated in ascending order of size, starting from the smallest one. Let us start with a simple case, which is a convenient way to familiarize ourselves with the operation. The simplest case is of course for $n = 1$. If there is only one disc, it is straightforward to perform operation $(1; 1 \rightarrow 3)$, which consists of simply moving the disc from rod 1 to rod 3; thus, $H(1) = 1$. Next, let us consider the case $n = 2$. In the beginning, the only operations that are possible are $(1; 1 \rightarrow 3)$ or $(1; 1 \rightarrow 2)$. As we eventually want to execute $(2; 1 \rightarrow 3)$, we choose the latter alternative. With a little thought, it is not difficult to conclude that the best operations are $(1; 1 \rightarrow 2)$, $(2; 1 \rightarrow 3)$, and $(1; 2 \rightarrow 3)$. Therefore, $H(2) = 3$. At this point, we start getting confused for values of $n = 3$ or greater. After some thought, we notice the following three points:

- We necessarily have to perform operation $(3; 1 \rightarrow 3)$.
- Operation $(3; 1 \rightarrow 3)$ requires all discs except 3 to be moved from rod 1 to rod 2.
- After performing $(3; 1 \rightarrow 3)$, all discs already moved to rod 2 must be transferred to rod 3.

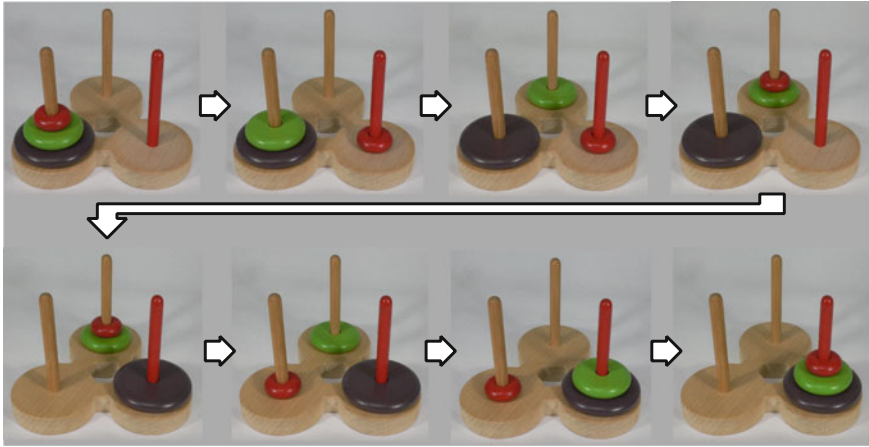


Fig. 2.2 View of the transitions of the Tower of Hanoi for $n = 3$

Regarding the “evacuation” operation of “moving all discs from rod 1 to rod 2,” we can reuse the procedure previously used for $n = 2$. In addition, the same procedure can be used for “moving all discs from rod 2 to rod 3.” In other words, the following procedure will work:

- Move discs 1, 2 from rod 1 to rod 2: $(1; 1 \rightarrow 3), (2; 1 \rightarrow 2), (1; 3 \rightarrow 2)$
- Move disc 3 to the target: $(3; 1 \rightarrow 3)$
- Move discs 1, 2 from rod 2 to rod 3: $(1; 2 \rightarrow 1), (2; 2 \rightarrow 3), (1; 1 \rightarrow 3)$

Therefore, we have $H(3) = 7$ (Fig. 2.2).

Here, let us further look into the rationale behind the $n = 3$ case. We can see that it can be generalized to the general case of moving k discs, i.e.,

- It is always true that we have to execute operation $(k; 1 \rightarrow 3)$.
- Executing operation $(k; 1 \rightarrow 3)$ requires all $k - 1$ discs to be moved from disc 1 to disc 2.
- After executing operation $(k; 1 \rightarrow 3)$, all discs sent to rod 2 must be moved to rod 3.

An important point here is the hypothesis that “the method for moving $k - 1$ discs is already known.” In other words, if we **hypothesize** that the problem of moving $k - 1$ discs is already solved, it is possible to move k discs. (By contrast, as it is not possible to move k discs without moving $k - 1$ discs, this is also a necessary condition.) This rationale is the core of a recursive call, that is, it is a technique that can be used when the two following conditions are met:

- (1) Upon solving a given problem, the algorithm for solving its own partial problem can be reused;
- (2) The solution to a sufficiently small partial problem is already known. In the case of the Tower of Hanoi in question, the solution to the $k = 1$ case is trivial.

If the above is written as an algorithm, the following algorithm for solving the Tower of Hanoi problem takes shape. By calling this algorithm as $\text{Hanoi}(64, 1, 3)$, in principle the steps for moving the discs will be output.

Algorithm 11: Algorithm $\text{Hanoi}(n, i, j)$ for solving the Tower of Hanoi problem

Input : number of discs n , rods i and j

Output: procedure for moving discs around

```

1 if  $n = 1$  then
2   output "move disc 1 from rod  $i$  to rod  $j$ "
3 else
4   let  $k$  be the other rod than  $i, j$ ;
5    $\text{Hanoi}(n - 1, i, k)$ ;
6   output "move disc  $n$  from rod  $i$  to rod  $j$ ";
7    $\text{Hanoi}(n - 1, k, j)$ ;
8 end

```

We can see that the above can be written in a straightforward manner, with no unnecessary moves. Here, a reader who is not familiarized with recursive calls may feel somewhat fooled. In particular, the behavior of variable n and that of variables i, j may seem a little odd. At first sight, variable n seems to have multiple meanings. For example, when $\text{Hanoi}(3, 1, 2)$ is executed, $n = 3$ in the beginning, but as the execution of $\text{Hanoi}(n - 1, i, k)$ proceeds, we require $n = 2$ inside the call. Further into the process, a function call takes place with $n = 1$, and the sequence of recursive calls stops. An example of a $\text{Hanoi}(3, 1, 2)$ execution is shown in Fig. 2.3. Within this sequence of calls, "variable n " represents different roles under the same name. This may seem strange to the reader. This is useful for those who are familiarized, but may confuse those who are not. Readers who are not particularly interested may proceed, but unconvinced ones are invited to refer to Sect. 2.1.2, "Mechanism of Recursive Calls."

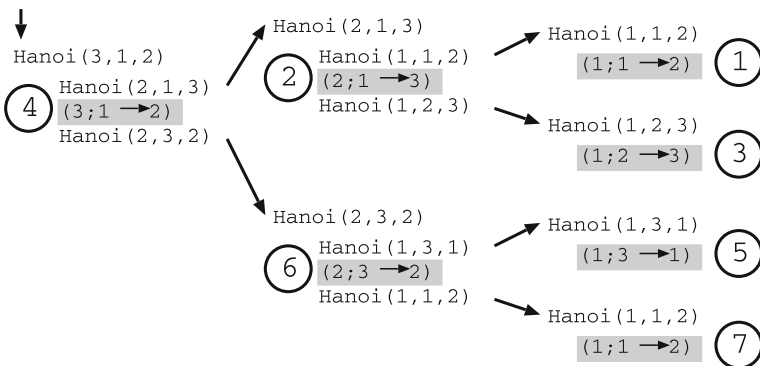



Fig. 2.3 Execution flow of $\text{Hanoi}(3, 1, 2)$. The parts displayed against a gray background are outputs, and the circled numbers denote the order of output

Exercise 18  Write $\text{Hanoi}(4, 1, 3)$ down by hand. What is the value of $H(4)$? Predict the value of the general case $H(n)$.

2.1.1 Analysis of the Tower of Hanoi

If we actually compute $H(4)$ of Exercise 18, we are surprised by the unexpectedly large number obtained. In the present section, we will consider the value of $H(n)$ in the general case. In terms of expressions, the analysis above can be formulated as the following two expressions:

$$H(1) = 1$$

$$H(n) = H(n-1) + 1 + H(n-1) = 2H(n-1) + 1 \quad (\text{for } n > 1)$$


Recursive calls are intimately related to mathematical induction. This is not surprising, considering that they almost represent the two sides of the same coin. However, we will try the easiest way. First, let us add 1 to both sides of the equation $H(n) = 2H(n-1) + 1$, obtaining $H(n) + 1 = 2H(n-1) + 2 = 2(H(n-1) + 1)$. Denoting $H'(n) = H(n) + 1$, we obtain $H'(1) = H(1) + 1 = 2$, and we can further write $H'(n) = 2H'(n-1)$. By further expanding the right-hand side for a general n , we obtain:

$$H'(n) = 2H'(n-1) = 2(2H'(n-2)) = \dots = 2^{n-1}H'(1) = 2^n$$

returning to the original equation, we have:

$$H(n) = H'(n) - 1 = 2^n - 1.$$

In other words, roughly speaking, adding one disc increases the number of moves by approximately two times. As we have seen in Sect. 1.5, this is an exponential function and the number is known to grow explosively.

Exercise 19  How many moves $H(n)$ are required to move 64 discs? Considering that it takes 1 second to move a single disc, what is the approximate time left before the end of the world?

2.1.2 Recurrent Call Mechanism

If we think about it, we realize that the mechanism of recurrent calls is a strange one. Why do we attach different meanings to variables with the same name and manage to control the process without confusion? It is worth taking an alternative approach to

carefully consider this. For example, let us look at the following Duplicate algorithm, which results from a slight modification of the Tower of Hanoi algorithm:

Algorithm 12: Duplicate(n)

Input : Natural number n

Output: if $n > 1$ then display n twice each time, with a recurrent call in between

```

1 if  $n = 1$  then
2   | output the value of  $n$ 
3 else
4   | output the value of  $n$ ;
5   | Duplicate( $n - 1$ );
6   | output the value of  $n$ ;
7   | Duplicate( $n - 1$ );
8 end

```

The algorithm has an impressive name, but is quite simple. It works as follows when actually executed. Because it basically repeats the action “output n and attach the output up to $n - 1$ ” twice, it is easy to consider the sequences that are output by starting with a small value of n :

- Duplicate(1) produces 1 as the output.
- Duplicate(2) produces 2121 as the output.
- Duplicate(3) produces 3212132121 as the output.
- Duplicate(4) produces 4321213212143212132121 as the output.

The sequences of numbers that are output seem to have a meaning, but in fact, they do not. It is worth noting “variable n ” here. It is nested upon each recursive call and controlled in separate memory areas under the same name n . Let us present an aspect of this recursive call in the form of a diagram (Fig. 2.4, where the second half is omitted). Roughly speaking, the vertical axis shows the process flow, which represents time evolution. The issue is the horizontal axis. The horizontal axis shows the “level” of recursive calls. The level becomes deeper and deeper as we proceed rightwards. When the subroutine named Duplicate is called with an argument n as in Duplicate(n), the computer first allocates local memory space for argument n and stores this value. When the recursive call ends, this memory area is released/cleared by deleting its contents.

To further clarify this problem, let us denote the memory area allocated upon the i -th call to Duplicate(n) as n_i . When Duplicate(4) is called, memory for n_1 is first allocated, and when Duplicate(3) is executed in between, memory n_2 is allocated, and so on. When n_i is no longer necessary, it is released. Figure 2.4 also shows aspects of memory allocation and clearance. If n_i is correctly managed, recursive calls are executed correctly. On the other hand, if any kind of confusion occurs, it is not possible to correctly identify the variables. Here we denote the event “allocated variable n_i ” as $[$, and the event “released variable n_i ” as $]$. Using this notation, if we place the events involving variable allocation and release in Fig. 2.4 in chronological order, we obtain the following sequence of symbols. The figure shows only the first

On the other hand, it is also useful to have variables that “always mean the same thing, anywhere.” These are called “**global variables**.” Managing “global variables” is not difficult. However, excessive use of global variables is not a smart practice for a programmer. The best practice is to divide the problem into local problems, and then encapsulate and solve them within their ranges.

2.2 Fibonacci Numbers

Fibonacci numbers denote a sequence of numbers that go under the name of Italian mathematician Leonardo Fibonacci.¹ More precisely, these numbers are known as Fibonacci sequence, and the numbers that belong to this are called Fibonacci numbers. The sequence actually has quite a simple definition, namely:

$$\begin{aligned} F(1) &= 1, \\ F(2) &= 1, \\ F(n) &= F(n-1) + F(n-2) \quad (\text{for } n > 2). \end{aligned}$$

In concrete terms, the sequence can be enumerated as 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, The Fibonacci sequence $F(n)$ is known to attract widespread interest for frequently appearing as a numerical model when we try to build models representing natural phenomena, such as the arrangement of seeds in sunflowers, the number of rabbit siblings, and others. Let us deepen our understanding of recursive calls using this sequence. At the same time, let us discover the limitations of recursive calls and introduce further refinements.

2.2.1 Computing Fibonacci Numbers $F(n)$ Arising from Recursive Calls

There is one particular aspect that must be considered carefully when working with recursive calls. We have seen that, in recursive calls, the algorithm capable of solving its own partial problem is reused. It is worth noting to what extent this “partial problem” is necessary. In the Tower of Hanoi problem, we reused the algorithm for solving the partial problem of size $n-1$ to solve a problem of size n . Considering the definition of Fibonacci numbers $F(n)$, the values of both $F(n-1)$ and $F(n-2)$ are required. From this fact, we learn that it does not suffice to show a clear solution for the small partial problem in $n=1$, but the solutions for $n=1$ and $n=2$ are

¹**Leonardo Fibonacci:1170?–1250?:**

Italian mathematician. His real name was Leonardo da Pisa, which means “Leonardo from Pisa.” “Leonardo Fibonacci” means “Leonardo, son of Bonacci.” In fact, he did not invent Fibonacci numbers himself. They borrowed his name due to the popularity gained after he mentioned them in his book “Liber Abaci” (book on abacus).

also needed. If we construct the algorithm according to its definition while paying attention to this point, we obtain the algorithm below. Using this algorithm, the n -th Fibonacci number $F(n)$ can be obtained by calling $\text{Fibr}(n)$, which produces the returned value as its output.

Algorithm 13: recursive algorithm $\text{Fibr}(n)$ to compute the n -th Fibonacci number $F(n)$


Input : n

Output: the n -th Fibonacci number $F(n)$

```

1 if  $n = 1$  then return 1;
2 if  $n = 2$  then return 1;
3 return  $\text{Fibr}(n - 1) + \text{Fibr}(n - 2)$ ;

```

Exercise 20  Implement and execute the algorithm above. How will the program behave when n is made slightly larger (by a few tens)?

2.2.2 Execution Time of Fibr Based on Recursive Calls

When we implement and execute $\text{Fibr}(n)$, execution clearly slows down for values around $n = 40$ and above. Why does this occur? Let us consider an execution time $t_F(n)$. When considering execution times, examining the recursive equation is inevitable. In concrete terms, we have:

- For $n = 1$: a constant time $t_F(1) = c_1$
- For $n = 2$: a constant time $t_F(2) = c_2$
- For $n > 2$: for a given constant c_3 , $t_F(n) = t_F(n - 1) + t_F(n - 2) + c_3$

Here, the value of the constant itself has no meaning, and therefore, we can simply denote them as c . By adding c to both sides of the equation, for $n > 2$ we can write:

$$t_F(n) + c = (t_F(n - 1) + c) + (t_F(n - 2) + c)$$

In other words, if we introduce another function $t'_F(n) = t_F(n) + c$, we can write:

$$t'_F(n) = t'_F(n - 1) + t'_F(n - 2),$$

which is exactly equivalent to the definition of Fibonacci numbers. In other words, the execution time $t_F(n)$ of $\text{Fibr}(n)$ can be considered proportional to the value of Fibonacci numbers. If we anticipate the contents of Sect. 2.2.4 below, it is known that Fibonacci numbers $F(n)$ will be $F(n) \sim \Theta(1.618^n)$. In other words, Fibonacci numbers constitute an exponential function. Therefore, if we compute Fibonacci numbers according to their definition, the computation time tends to increase exponentially, that is, if n becomes larger, the computation time of a naïve implementation is excessively large.

2.2.3 Fast Method for Computing Fibonacci Numbers

We learnt that, according to the definition, the time required to compute Fibonacci numbers increases exponentially. However, some readers may feel intrigued by that. For instance, to compute $F(9)$, only $F(8)$ to $F(1)$ suffice. First, let us solve the question why the time required for computing Fibonacci numbers increases exponentially. For example, $F(9)$ can be computed if $F(8)$ and $F(7)$ are available. Computing $F(8)$ only requires knowing $F(7)$ and $F(6)$, and computing $F(7)$ only requires $F(6)$ and $F(5)$, and so forth. This calling mechanism is illustrated in Fig. 2.5.

What we need to note here is “who is calling who.” For instance, if we pay attention to $F(7)$, this value is called twice, that is, upon the computation of both $F(9)$ and $F(8)$. That is, $F(7)$ is nested twice to compute $F(9)$. Likewise, $F(6)$ is called from $F(8)$ and $F(7)$, and $F(7)$ is called twice. This structure becomes more evident as we approach the leaves of the tree structure. $F(2)$ and $F(3)$ appear in Fig. 2.5 remarkably often. Thus, the $\text{Fibr}(n)$ algorithm based on recursive calls computes the same value over and over again in a wasteful fashion.

Once the problem becomes clear at this level, the solution naturally becomes evident. The first idea that pops up is to use arrays. Using an array, we can start computing from the smallest value and refer to this value when necessary. Concretely speaking, we can use the following algorithm:

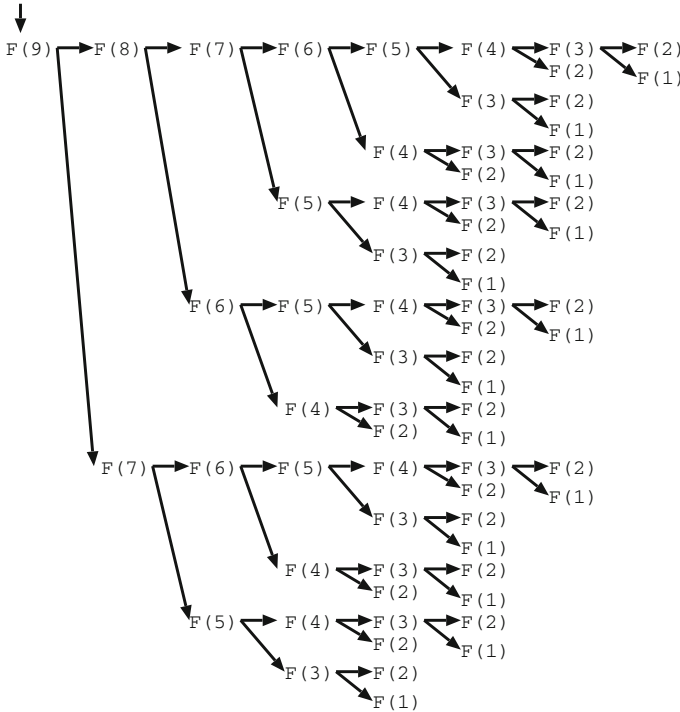


Fig. 2.5 Aspect of the computation of the Fibonacci number $F(9)$

Algorithm 14: Algorithm Fiba(n) to compute the n -th Fibonacci number $F(n)$ using array $Fa[]$

Input : n
Output: n -th Fibonacci number $F(n)$

```

1  $Fa[1] \leftarrow 1$ ;
2  $Fa[2] \leftarrow 1$ ;
3 for  $i \leftarrow 3, 4, \dots, n$  do
4    $Fa[i] \leftarrow Fa[i - 1] + Fa[i - 2]$ ;
5 end
6 output  $Fa[n]$ ;

```

First, $F(1)$ and $F(2)$ are directly computed when $i = 1, 2$. Then, for $i > 2$ it is quite clear that for computing $Fa[i]$ we already have the correct values of $Fa[i - 1]$ and $Fa[i - 2]$. These two observations indicate to us that Fibonacci numbers can be correctly computed by this algorithm. Therefore, the computation time of the Fiba(n) algorithm is linear, that is, proportional to n , which means it is extremely fast.

Let us stop for a while to think ahead. For $i > 2$, what is necessary for computing $F(i)$ is $F(i - 1)$ and $F(i - 2)$, and elements further behind are not needed. In other words, provided that we keep in mind the last two elements, the array itself is unnecessary. With that perception in mind, we can write an even more efficient algorithm (from the viewpoint of memory space).


Algorithm 15: Algorithm Fib2(n) to compute the n -th Fibonacci number $F(n)$ without using arrays

Input : n
Output: n -th Fibonacci number $F(n)$

```

1 if  $n < 3$  then
2   output "1";
3 else
4    $Fa1 \leftarrow 1$ ;                                /* Memorize  $F(1)$  */
5    $Fa2 \leftarrow 1$ ;                                /* Memorize  $F(2)$  */
6   for  $i \leftarrow 3, 4, \dots, n$  do
7      $Fa \leftarrow Fa2 + Fa1$ ; /* Compute/memorize  $F(i) = F(i - 1) + F(i - 2)$  */
8      $Fa1 \leftarrow Fa2$ ;          /* Update  $F(i - 2)$  with  $F(i - 1)$  */
9      $Fa2 \leftarrow Fa$ ;           /* Update  $F(i - 1)$  with  $F(i)$  */
10  end
11  output  $Fa$ ;
12 end

```

Exercise 21  Implement and execute the algorithm above. Confirm that the answer is output immediately even for very large values of n .

2.2.4 Extremely Fast Method to Compute Fibonacci Numbers


Fibonacci numbers $F(n)$ have fascinated mathematicians since ancient times due to their intriguing and interesting properties. In this section, let us borrow some of the


results of such mathematical research. In fact, it is known that the general term of Fibonacci numbers $F(n)$ can be expressed by the following expression:

$$F(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$$

It is fascinating that even though $\sqrt{5}$ appears several times in the equation, the result $F(n)$ is always a natural number for any input n that is a natural number. If we implement an algorithm that computes this equation, the computation time of a Fibonacci number $F(n)$ ends in a constant time. (Considering the errors involved in the computation of $\sqrt{5}$, there might be cases in which the algorithm that allows computation in linear time of the previous section is preferable.)

In the present Section, let us prove that the Fibonacci number $F(n)$ approximately follows $\Theta(1.618^n)$. First, consider the two terms $\left(\frac{1+\sqrt{5}}{2}\right)^n$ and $\left(\frac{1-\sqrt{5}}{2}\right)^n$. If we actually compute them, we obtain $\left(\frac{1+\sqrt{5}}{2}\right) = 1.61803\dots$ and $\left(\frac{1-\sqrt{5}}{2}\right) = -0.61803\dots$. Thus, the absolute value of the former is considerably larger than 1, whereas the absolute value of the latter is considerably smaller than 1. Therefore, as n increases the value of $\left(\frac{1+\sqrt{5}}{2}\right)^n$ increases very fast, whereas the value of $\left(\frac{1-\sqrt{5}}{2}\right)^n$ becomes small very quickly. Thus, the value of $F(n)$ is approximately $1.618^n / \sqrt{5}$, which is of the order $\Theta(1.618^n)$, i.e., an exponential function.

Exercise 22  Prove by means of mathematical induction that the general term equation for Fibonacci numbers $F(n)$ actually holds.

Exercise 23  Investigate the relation between Fibonacci numbers $F(n)$ and the golden ratio ϕ . The **golden ratio** ϕ is a constant defined as $\phi = \frac{1+\sqrt{5}}{2}$, and its value is approximately 1.618.

2.3 Divide-and-Conquer and Dynamic Programming

When tackling a large problem using a computer, approaching the problem directly as a whole often does not lead to good results. In such cases, we must first consider whether it is possible to split the large problem into small partial problems. In many cases, a large problem that cannot be tackled as it is can be split into sufficiently small partial problems that can be appropriately handled individually. Solving these individual partial problems and combining their solutions often permits the original large problem to be solved. This method is known as “**divide and conquer**.” The idea is that a large target must be divided to allow for proper management. Political units such as countries, prefectures, cities, towns, and villages also constitute examples of this idea. The divide-and-conquer method can be considered a “top-down” approach.

The Tower of Hanoi problem was successfully solved by splitting the original problem consisting of moving n discs into the following partial problems:

- Partial problem where the number of discs is $n - 1$
- Partial problem of moving only 1 disc (the largest one)

This is an example of divide-and-conquer. Regarding the computation of Fibonacci numbers, using a recursive definition for computation can be considered a divide-and-conquer method. However, in the case of Fibonacci numbers, we found a computation method that is much more efficient than the divide-and-conquer method. The limitation of the “divide-and-conquer” method for Fibonacci numbers is that the solution to the problems that had already been solved is forgotten every time. This is a characteristic of top-down approaches that is difficult to avoid, a fatal limitation in the case of Fibonacci numbers. This condition resembles the situation of vertical administration systems where similar organizations exist in several places.

Dynamic programming is a method conceived to address such problems. In this method, minutely sliced problems are solved in a bottom-up approach. When optimal solutions to partial problems are found, such that no further improvement is possible, only the necessary information is retained and all the unnecessary information is forgotten. In the Fibonacci numbers example, an important point is that computations are carried out in a determined order, starting from the small values. Because only the values of the Fibonacci numbers immediately preceding $F(i - 1)$ and $F(i - 2)$ are needed, values further back can be forgotten. In other words, for Fibonacci number computation, it suffices to retain only the previous two values.

In the case of the Tower of Hanoi problem, because all discs must be moved every time, the problem does not fit well in the dynamic programming framework. For this reason, the solution is limited to the divide-and-conquer method.