onto the stack and calls itself for the second time with the value 2. This time, the `else` statement is again executed, and `n` (value = 2) is pushed onto the stack as the function calls itself for the third time with the value 1. Now, the `if` statement is executed and as `n` = 1, the function returns 1. Since the value of `Factorial(1)` is now known, it reverts to its second execution by popping the last value 2 from the stack and multiplying it by 1. This operation gives the value of `Factorial(2)`, so the function reverts to its first execution by popping the next value 3 from the stack and multiplying it with the factorial, giving the value 6, which the function finally returns.

From this example, we notice the following:

1. The `Factorial()` function in Program Code 4.2 runs three times for `n` = 3, out of which it calls itself two times. The number of times a function calls itself is known as the *recursive depth* of that function.
2. Each time the function calls itself, it stores one or more variables on the stack. Since stacks hold a limited amount of memory, the functions with a high recursive depth may crash because of non-availability of memory. Such a situation is known as *stack overflow*.
3. Recursive functions usually have (and in fact should have) a *terminating* (or *end*) *condition*. The `Factorial()` function in Program Code 4.2 stops calling itself when `n` = 1. If this condition was not present, the function would keep calling itself with the values 3, 2, 1, 0, −1, −2, and so on. Such recursion is known as *endless recursion*.
4. All recursive functions go through two distinct phases. The first phase, *winding*, occurs when the function calls itself and pushes values onto the stack. The second phase, *unwinding*, occurs when the function pops values from the stack, usually after the end condition.

4.4 VARIANTS OF RECURSION

Depending on the following characterization, the recursive functions are categorized as direct, indirect, linear, tree, and tail recursions. Recursion may have any one of the following forms:

1. A function calls itself.
2. A function calls another function which in turn calls the caller function.
3. The function call is part of the same processing instruction that makes a recursive function call.

A few more terms that are used with respect to recursion are explained in the following section.

Binary recursion A *binary recursive* function calls itself twice. Fibonacci numbers computation, quick sort, and merge sort are examples of binary recursion.

Program Code 4.5 is an example of a binary recursion as the function `Fib()` calls itself twice.

PROGRAM CODE 4.5

```
int Fib(n)
{
   if(n == 1 ||n == 2)
      return 1;
   else
      return(Fib(n - 1) + Fib(n - 2));
}
```

n-ary recursion and permutations The most general form of recursion is *n-ary recursion*, where *n* is not a constant but some parameter of a function. Functions of this kind are useful in generating combinatorial objects such as permutations.

4.4.1 Direct Recursion

Recursion is when a function calls itself. Recursion is said to be *direct* when a function calls itself directly, and it is said to be *indirect* when it calls another function which in turn calls it. The `Factorial()` function we discussed in Program Code 4.2 is an example of direct recursion. The `Power()` function in Program Code 4.6 is for computing the value of Eq. (4.4) recursively. It is a slightly modified version of Program Code 4.3.

PROGRAM CODE 4.6

```
int Power(int x, int y)
{
   if(y == 1)
      return x;
   else
      return (x * Power(x, y - 1));
}
```

4.4.2 Indirect Recursion

A function is said to be indirectly recursive if it calls another function, which in turn calls it. Program Code 4.7 is an example of an indirect recursion, where the function `Fact()` calls the function `Dummy()`, and the function `Dummy()` in turn calls `Fact()`.

PROGRAM CODE 4.7

```
int Fact(int n)
{
   if(n <= 1)
      return 1;
   else
      return (n * Dummy(n - 1));
}

void Dummy(int n)
{
   Fact(n);
}
```

4.4.3 Tail Recursion

A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Tail recursion is also used to return the value of the last recursive call as the value of the function. Tail recursion is advantageous as the amount of information that must be stored during computation is independent of the number of recursive calls. The `Factorial()` function in Program Code 4.2 is an example of a non-tail recursive function. The `Binary_Search()` function in Program Code 4.8 is an example of a tail recursive function.

PROGRAM CODE 4.8

```
int Binary_Search(int A[], int low, int high, int key)
{
   int mid;
   if(low <= high)
   {
      mid = (low + high)/2;
      if(A[mid] == key)
         return mid;
      else if(key < A[mid])
         return Binary_Search(A, low, mid - 1, key);
      else
         return Binary_Search(A, mid + 1, high, key);
   }
   return -1;
}
```

4.4.4 Linear Recursion

Depending on the way the recursion grows, it is classified as *linear* or *tree*. A recursive function is said to be *linearly recursive* when no pending operation involves another recursive call, for example, the `Fact()` function. This is the simplest form of recursion and occurs when an action has a simple repetitive structure consisting of some basic steps followed by the action again. The `Factorial()` function in Program Code 4.2 is an example of linear recursion.

4.4.5 Tree Recursion

In a recursive function, if there is another recursive call in the set of operations to be completed after the recursion is over, this is called a *tree recursion.* Examples of tree recursive functions are the quick sort and merge sort algorithms, the FibSeries algorithm, and so on.

The Fibonacci function FibSeries() is defined as

$$\text{FibSeries}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{FibSeries}(n-1) + \text{FibSeries}(n-2), & \text{otherwise} \end{cases}$$

Let $n = 5$.
FibSeries(0) = 0
FibSeries(1) = 1
FibSeries(2) = FibSeries(0) + FibSeries(1) = 1
FibSeries(3) = FibSeries(1) + FibSeries(2) = 2
FibSeries(4) = FibSeries(2) + FibSeries(3) = 3
FibSeries(5) = FibSeries(3) + FibSeries(4) = 5

Figure 4.1 demonstrates this explanation for $n = 4$.

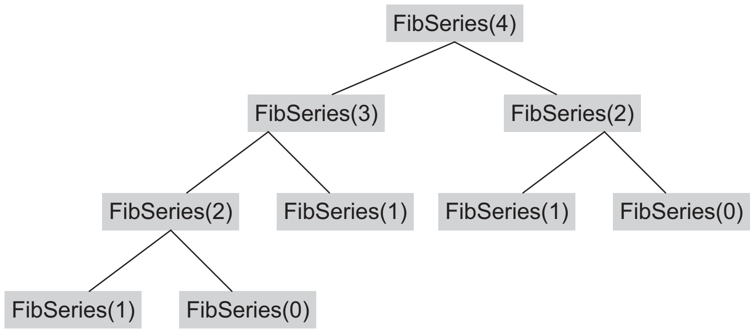


Fig. 4.1 Recursive calls in Fibonacci recursive function for $n = 4$

4.5 EXECUTION OF RECURSIVE CALLS

Let us now see how recursive calls are executed. At every recursive call, all reference parameters and local variables are pushed onto the stack along with the function value and return address. The data is conceptually placed in a *stack frame*, which is pushed onto the system stack. A stack frame contains four different elements:

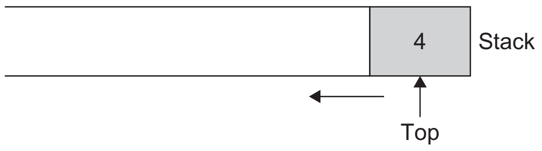
1. The reference parameters to be processed by the called function
2. Local variables in the calling function
3. The return address
4. The expression that is to receive the return value, if any

Consider the following two lines from the `Factorial()` function in Program Code 4.2:

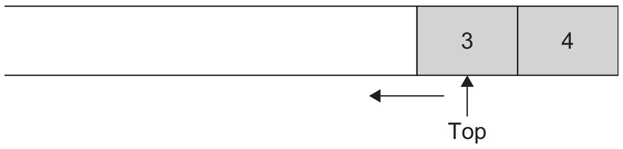
```
if(n <= 1) return 1;
else return n * Factorial(n - 1);
```

Consider the first call as `Factorial(4)`. Now,

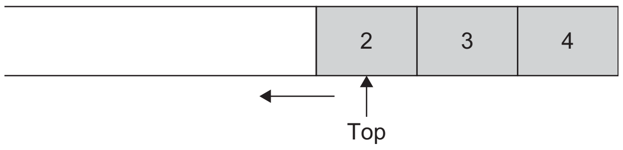
1. $n = 4$
   Hence, statement 2, which is a recursive call, is executed.
   Push 4 onto the stack and call `Factorial(4 - 1)`.



2. $n = 3$
   Hence, push 3 onto the stack and call `Factorial(2)`.



3. $n = 2$
   Hence, push 2 onto the stack and call `Factorial(1)`.



4. $n = 1$
   Now execute statement 1, which returns 1.
5. Pop the contents and $n = 2$, so now the expression becomes 2×1.
6. Now, $n = 3$ after popping the top of the stack contents.
   Therefore, the expression is $3 \times 2 \times 1$.
7. After popping the top of the stack contents applying $n = 4$, the expression is $4 \times 3 \times 2 \times 1 = 24$.
8. After popping the top of the stack contents, we get to know that the stack is empty, and the answer is $4! = 24$.

At the end condition, when no more recursive calls are made, the following steps are performed:

1. If the stack is empty, then execute a normal return.
2. Otherwise, pop the stack frame, that is, take the values of all the parameters that are on the top of the stack and assign these values to the corresponding variables.
3. Use the return address to locate the place where the call was made.
4. Execute all the statements from that place (address) where the call was made.
5. Go to step 1.

4.6 RECURSIVE FUNCTIONS

Recursion is usually viewed by students as a mystical technique that is useful only for some very special class of problems such as computing factorials or the Fibonacci series. This is not true. Practically, any function written using an iterative code can be converted into a recursive code. Of course, this does not guarantee that the resulting program will be easy to understand, but often, the program results in a compact and readable code.

Let us see when recursion is an appropriate solution. One instance is when the problem itself is recursively defined. Appropriate examples of this could be factorial and binomial coefficients.

1. $n! = n \times (n - 1)!\ \{\text{if } n = 1, n! = 1\}$
2. $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$
3. $\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$
4. $x^y = x \times x^{y-1}$

Recursive functions are often simple and elegant, and their correctness can be easily verified. Many mathematical functions are defined recursively, and their translation into a programming language is often easy. Recursion is natural in Ada, Algol, C, C++, Haskell, Java, Lisp, ML, Modula, Pascal, and many other programming languages. When used carelessly, recursion can sometimes result in an inefficient function. Recursive functions are closely related to inductive definitions of functions in mathematics. To evaluate whether an algorithm is to be written using recursion, we must first try to deduce an inductive definition of the algorithm.

Algorithms that are by nature recursive, such as the factorial, Fibonacci, or power, can be implemented as either iterative or recursive code. However, recursive functions are generally smaller and more efficient than their looping equivalents.

Let us consider an example. Consider a given set of cardinality $n \geq 1$. The problem is to print all the permutations of the set. For example, if the set is {1, 2, 3}, then all the permutations are as follows:

$$\{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\}, \{2, 3, 1\}, \{3, 1, 2\}, \text{and } \{3, 2, 1\}$$

The total number of possible permutations of a set of cardinality n is $n!$. The easiest way to generate these permutations is as follows:

$$\text{Let } S = \{a, b, c, d\}$$

Generate each permutation by printing the following:

1. a followed by the permutations of set $\{b, c, d\}$
2. b followed by the permutations of set $\{a, c, d\}$
3. c followed by the permutations of set $\{a, b, d\}$
4. d followed by the permutations of set $\{a, b, c\}$

Here, the phrase 'followed by' is the part that introduces recursion. This approach implies that we can solve the problem for a set with n elements if we had an algorithm that worked on $(n - 1)$ elements. These considerations lead to Algorithm 4.3.

ALGORITHM 4.3

```
Perm(A, i, n)
begin
   if(i = n) then
      print(A) and return
   B = A
   for j = i to n do
   begin
      Interchange(A, i, j)
      Perm(A, i + 1, n)
      A = B
   end
end
```

Moreover, recursion is also useful when the data structure that the algorithm is to operate on is recursively defined. Examples of such data structures are linked lists and trees. One more instance when recursion is valuable is when we use 'divide and conquer' and 'backtracking' as algorithm design paradigms. *Divide and conquer* is a technique where, for a function to compute n inputs, the strategy suggests splitting the inputs into k distinct subsets, $1 < k \leq n$, yielding k sub-problems. These sub-problems must then be solved and should be combined to get the final solution. If the sub-problem is still large, the technique is reapplied. The reapplication is expressed better by the recursive function. Recursion is a technique that allows us to break down a problem into one or more sub-problems that are similar in form to the original problem. Examples include binary search, merge sort, and quick sort.

4.6.1 Writing Recursive Code

The general approach to writing a recursive function is listed in the following sequence:

1. Write the function header so you are sure what the function will do and how it will be called. Identify some unit of measure for the size of the problem the function or procedure will work on. Then, pretend that the task is to write a function that will work on problems of all sizes.
2. Decompose the problem into sub-problems. Identify clearly the non-recursive case of the problem. Make it as small as possible. The function will nearly always begin by testing for this non-recursive case, also known as the *base case* or the *end condition*.
3. Write recursive calls to solve those sub-problems whose form is similar to that of the original problem.
4. Write the code to combine, enhance, or modify the results of the recursive call(s), if necessary, to construct the desired return value or create the desired side effects.
5. Write the end condition(s) to handle any situations that are not handled properly by the recursive portion of the program.

4.6.2 Tower of Hanoi: An Example of Recursion

The use of recursion often makes everything simpler. First, find out the recurring data and the essential feature of the problem that should change as the function calls itself. In the Tower of Hanoi solution, one recurs on the largest disk to be moved. That is, one has to write a recursive function that takes the largest disk as a parameter in the tower to be moved. The function should take three parameters indicating from which peg the tower should be moved (source), to which peg it should go (dest), and the last peg (spare), which is used temporarily.

Let us consider the initial position of the problem as in Fig. 4.2.

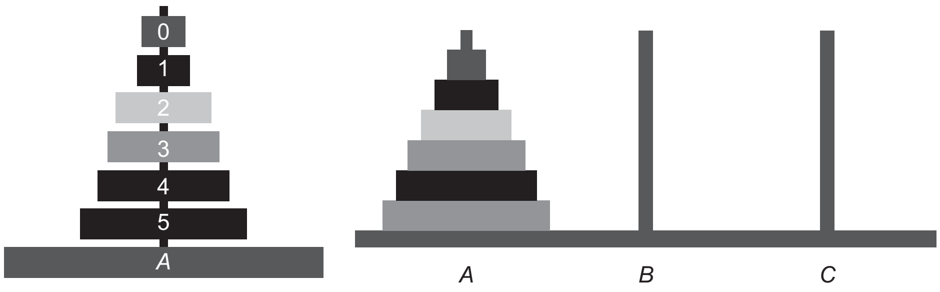


Fig. 4.2 Tower of Hanoi—initial position