APPENDIX H
RECURSIVE PROCEDURES

William Stallings

Supplement to
Computer Organization and Architecture, Ninth Edition
Prentice Hall 2012
ISBN: 013293633X
http://williamstallings.com/ComputerOrganization

Recursion is an important concept that cuts across many areas of computer science. Daniel McCracken, a noted computer science educator, put it this way [MCCR87]:

> **Is Recursion an Advanced Topic?**
> Absolutely not. Recursion is fundamental in computer science, whether understood as a mathematical concept, a programming technique, away of expressing an algorithm, or a problem-solving approach.

Recursion is listed as a required topic, both for the Programming Fundamentals Course and the Discrete Structures Course, recommended in *Computer Curricula 2001*, from the Joint Task Force on Computing Curricula of the IEEE Computer Society and the ACM, for the Undergraduate Program in Computer Science.

We briefly mentioned recursive procedures in Chapter 12 and the concept warrants elaboration, which we do in this appendix. Many students find recursion a difficult concept to grasp. Accordingly, this appendix covers a number of examples and uses various methods of description and presentation.

H.1 RECURSION

One of the classic examples of recursion is the factorial function. The factorial of a positive integer is computed as that integer times all of the integers below it up to and including 1. For example, factorial(5) is the same as $5 \times 4 \times 3 \times 2 \times 1$, and factorial(3) is $3 \times 2 \times 1$. Now consider: Suppose you want to compute the factorial of 27, and you already know the value of factorial(26). Then, you don't need to compute $27 \times 26 \times 25 \times \ldots \times 1$. Instead, you need only compute 27 × factorial(26). Similarly, if you know the value of

factorial(53), then the value of factorial(54) is computed as factorial(54) = 54 × factorial(53).

Thus, in general, we can state that factorial(n) = n × factorial(n – 1). However, as written, this definition involves an infinite recurrence, with the procedure calling itself recursively indefinitely. We need a condition that will stop the recursion; this is known as a **base case**. For the factorial function, the base case is n = 1, with factorial(1) = 1. We can then write the factorial function in C as follows:

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

The reader may at first be uncomfortable with the concept of a function or procedure that calls itself. We look at how to implement such a function in a later section. For now, just assume that it does work - that a function can call itself without causing an error condition in the operating system.

So far, what we have seen presents the essence of recursion in programming. First, it involves a function that calls itself. And second, the function definition must include a base case that enables the function to terminate the recursive process.

The factorial function is an example of the use of recursion to program a mathematical algorithm. But recursion is far more versatile. It is often appropriate for dealing with linked lists, tree structures, and search and sort algorithms. As an example, consider the binary search algorithm. Let sortedArray[ ] be an array sorted in descending order. The binary search algorithm locates a particular value in the sorted list. If the value is found,

the algorithm returns the position of the value. If the value is not in the list, then the algorithm returns the negative of the insertion position:

```
int BinarySearch(int sortedArray[], int key, int first, int last) {
    if (first <= last) {
       int mid = (first + last) / 2;
       if (key == sortedArray[mid])
          return mid;                                          /* found */
       else if (key < sortedArray[mid])
           return BinarySearch(sortedArray, key, first, mid-1);
       else                                  /* key > sortedArray[mid] */
           return BinarySearch(sortedArray, key, mid+1, last);
   }
   return -(first);                                          /* not found */
}
```

H.2 ACTIVATION TREE REPRESENTATION

To facilitate the understanding of recursive procedures, [HAYN95] introduced the activation tree. The recursion tree is a tree where each node is the "current environment." That is, each node contains parameters, local variables, and return values. Using this technique, it is easy to identify a node as a particular procedure executing in a particular environment. The parent of a node is the procedure that called the node. The children of a node are the procedures, which that node calls.

Figure H.1a shows the generic form of an activation tree node. The node includes the function name, the parameters passed to the function, and the values returned by the function. If the function name appears multiple times in the function definition, then each is distinguished by a unique subscript. Figure H.1b shows an example of the call/return structure of a procedure that has been called. The circled numbers in the nodes indicate the order in which nodes are activated. The dynamic execution of the program follows depth first traversal of the activation tree. The top node is the first invocation of the function. Multiple invocations from a node are listed as

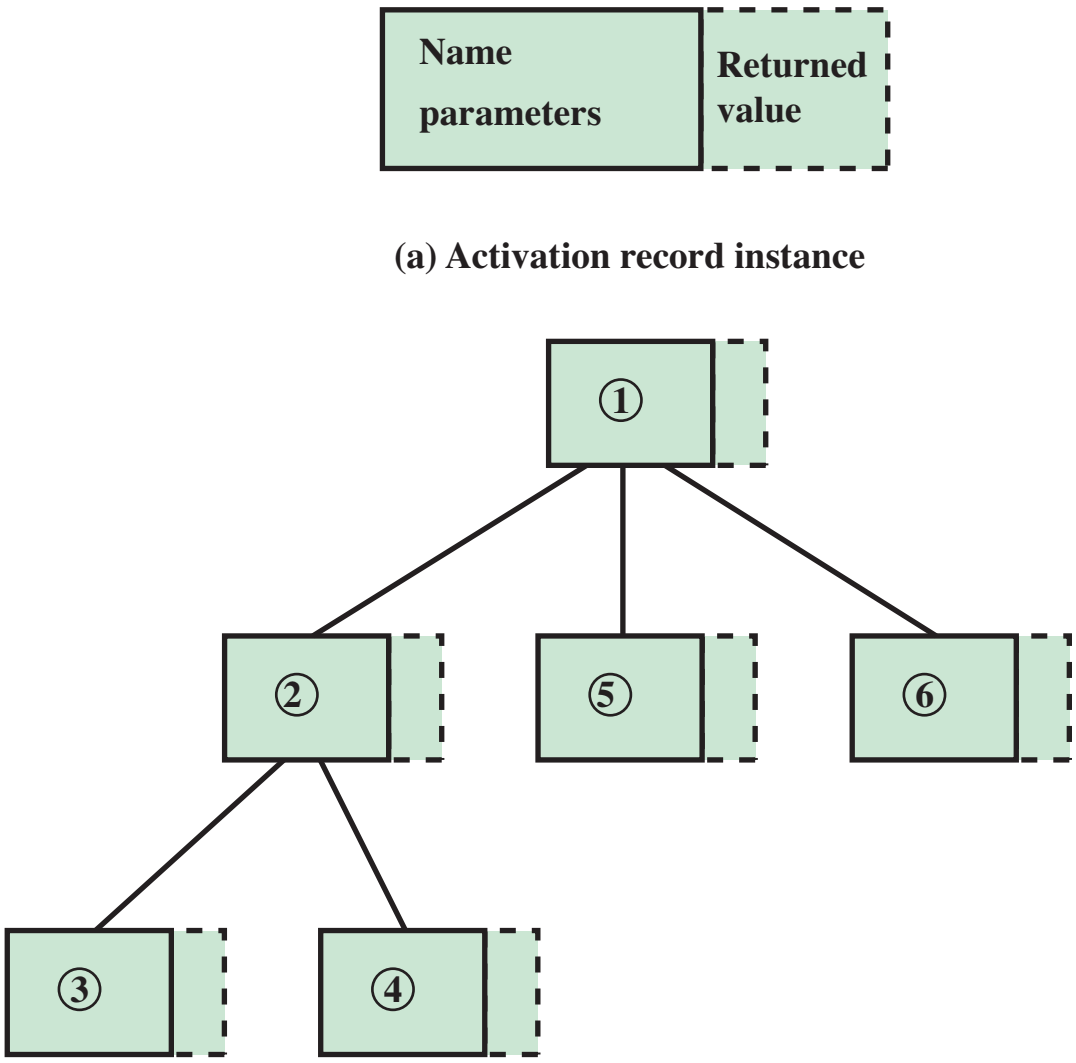


Figure H.1 Activation Record Conventions

children, with the invocations left to right. All of this should be clear as we examine two examples.

Fibonacci Numbers

Fibonacci numbers are defined as follows:

Fib(k) = Fib(k – l) + Fib(k – 2);

Fib(2) = 1;

Fib(l) = 1;

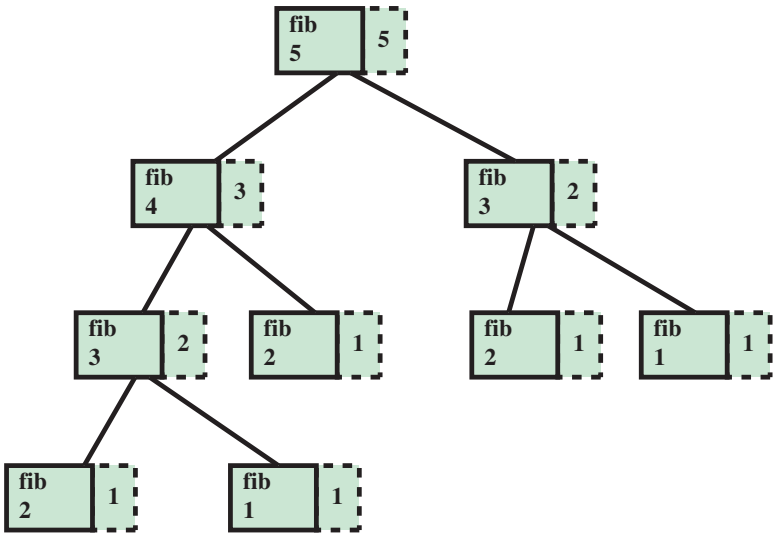
That is, each Fibonacci number is the sum of the preceding two numbers. It turns out that Fibonacci series occur in many contexts in nature. The start of the series is as follows:

1 1 2 3 4 8 13 21 34 55 89 144 233 377 610 987

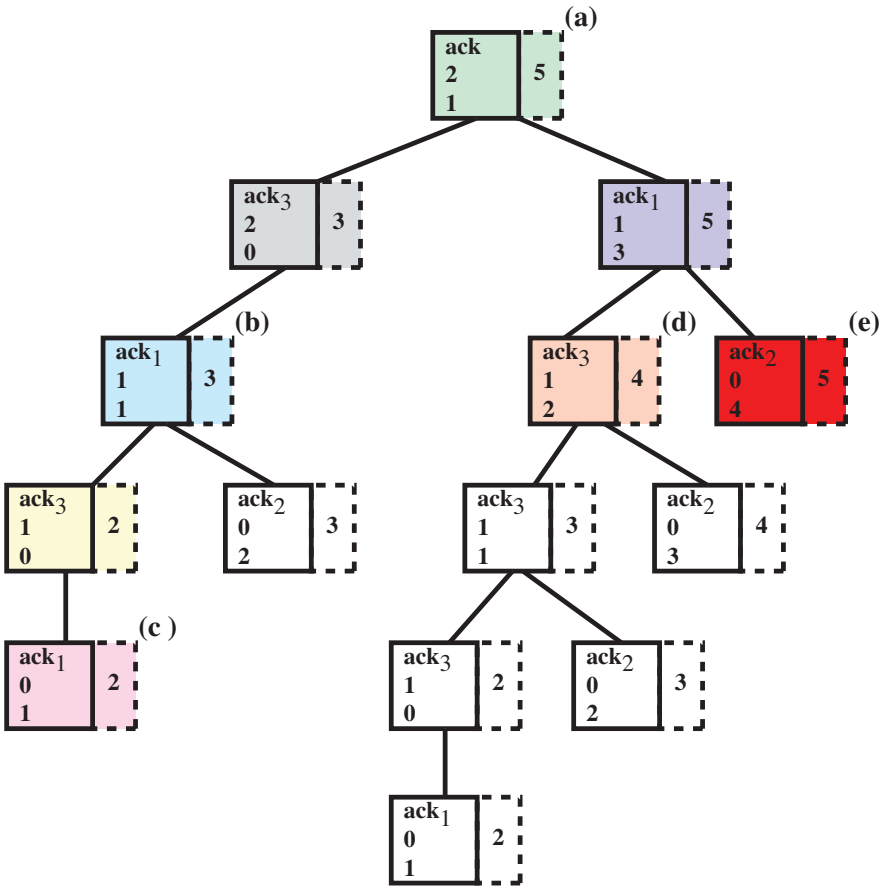
Here is a recursive program for the Fibonacci series:

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

Figure H.2a shows the activation tree for fib(5), which yields a result of 5. The topology of the tree is derived from the execution, and the execution is closely tied to the inductive definition (which was used to write the function). For example, the activation tree shows that the node for Fib(5) has two children: Fib(4) and Fib(3). That is, the value calculated for Fib(5) must use the values calculated for Fib(4) and Fib(3. The activation tree has the return values placed in the right-side box, making the return value accessible to the student. According to the definition of Fib, the values for Fib(4) and Fib(3) must be added together in order to obtain the value for



(a) Activation tree for Fibonacci(5)



(b) Activation tree for Ackermann(2, 1)

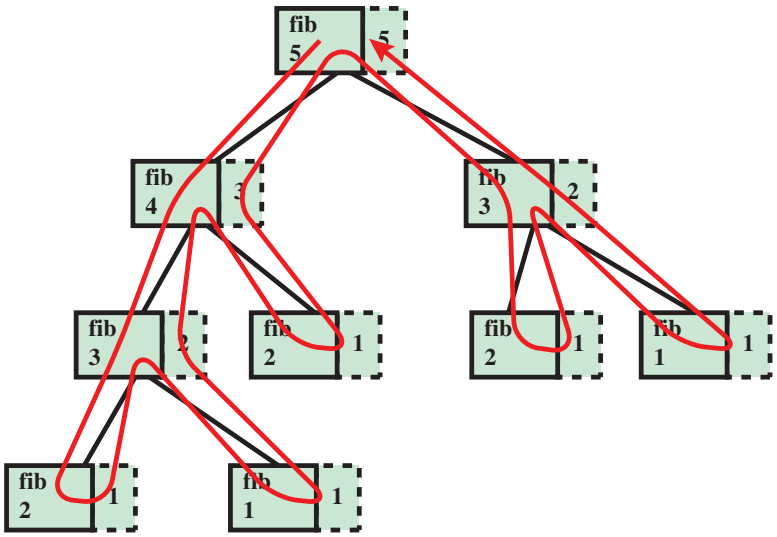
Figure H.2 Recursion Examples

Fib(5). If we replace the "5" in Fib(5) by "*n*", then the activation tree tells us that Fib(*n*) is a function of Fib(*n* – 1) and Fib(*n* – 2). The leaves of the activation tree correspond to the base case(s) of the inductive definition. Thus, Fib(2) returns, immediately, with the value of 1.

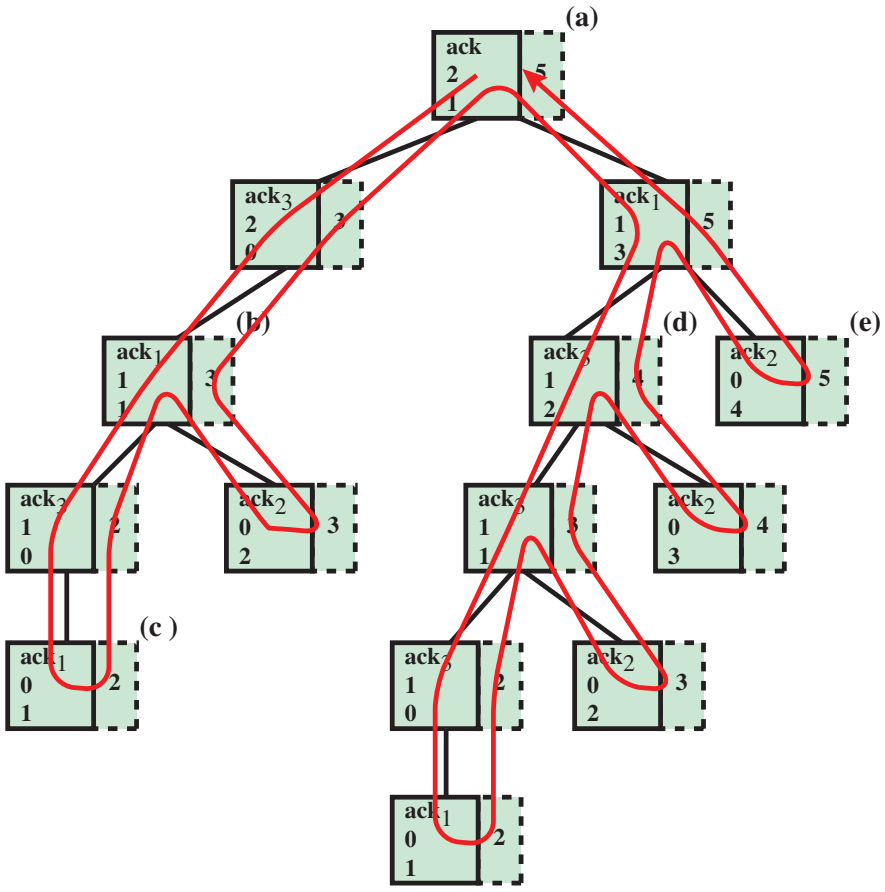
Here is a corresponding execution trace:

```
Entering: FIB , Argument list: (5)
     Entering: FIB , Argument list: (4)
          Entering: FIB , Argument list: (3)
               Entering: FIB , Argument list: (2)
               Exiting: FIB , Value: 1
               Entering: FIB , Argument list:. (1)
               Exiting: FIB , Value: 1
          Exiting: FIB , Value: 2
          Entering: FIB , Argument list: (2)
          Exiting: FIB , Value: 1
     Exiting: FIB , Value: 3
     Entering: FIB , Argument list: (3)
          Entering: FIB , Argument list: (2)
          Exiting: FIB , Value: 1
          Entering: FIB , Argument list: (1)
          Exiting: FIB , Value: 1
     Exiting: FIB , Value: 2
Exiting: FIB , Value: 5
```

Figure H.3a repeats Figure H.2a, showing the sequence in which the activation records are visited. This is a depth-first traversal of the tree, with a left-to-right sequence at each level.



(a) Activation tree for Fibonacci(5)



(b) Activation tree for Ackermann(2, 1)

Figure H.3 Recursion Examples: Execution Trace

Ackermann's Function

A more interesting recursive function is the deceptively simple Ackermann's function:

```
int ack(int m, int n) {
    if (m==0) return (n + 1);
    if (n==0) return (ack (m-1,1));
    return (ack (m-1, ack(m, n-1) ) );
}
```

In this case, the function is invoked recursively but, more than that, the function appears as an argument inside the recursive use of the function. The value of this function grows very rapidly, even for small inputs, and even though increases only result from addition of 1.. For example, A(4, 2) contains 19,729 decimal digits.

To clarify the operation of this function, we label each recursive call with a unique subscript. In each case, it is the same function; the subscripts simply make it easier to keep track of what is happening:

```
int ack(int m, int n) {
    if (m==0) return (n + 1);
    if (n==0) return (ack1 (m-1,1));
    return (ack2 (m-1, ack3(m, n-1) ) );
}
```

Figure H.2b shows the activation tree for ack(2, 1), which has the following execution trace:

```
Entering: ACK, Argument list: (2 1)                    //(a)
    Entering: ACK, Argument list: (2 0)
        Entering: ACK, Argument list: (1 1)            //(b)
            Entering: ACK, Argument list: (1 0)
                Entering: ACK, Argument list: (0 1)//(c)
                Exiting: ACK, Value: 2
            Exiting: ACK, Value: 2
            Entering: ACK, Argument list: (0 2)
            Exiting: ACK, Value: 3
        Exiting: ACK, Value: 3
    Exiting: ACK, Value: 3
    Entering: ACK, Argument list: (1 3)
        Entering: ACK, Argument list: (1 2)            //(d)
            Entering: ACK, Argument list: (1 1)
                Entering: ACK, Argument list: (1 0)
                    Entering: ACK, Argument list: (0 1)
                    Exiting: ACK, Value: 2
                Exiting: ACK, Value: 2
                Entering: ACK, Argument list: (0 2)
                Exiting: ACK, Value: 3
            Exiting: ACK, Value: 3
            Entering: ACK, Argument list: (0 3)
            Exiting: ACK, Value: 4
        Exiting: ACK, Value: 4
        Entering: ACK, Argument list: (0 4)            //(e)
        Exiting: ACK, Value: 5
    Exiting: ACK, Value: 5
Exiting: ACK, Value: 5
```

Some of the lines in the execution trace are labeled, which correspond to labels on the activation tree. This is for illustrative purposes only.

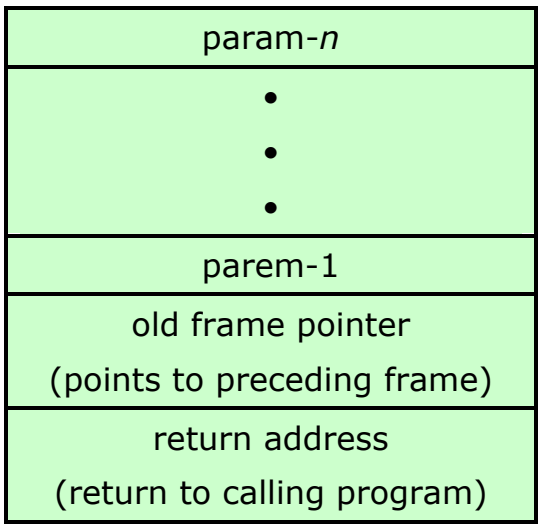
Figure H.3b shows the sequence in which the activation records are visited.

H.3 STACK IMPLEMENTATION

Implementation of recursive procedures in programming languages almost always involves the use of a stack. Each call of the procedure causes a stack frame, or activation record instance, to be pushed onto the control stack. As

discussed in Chapter 12 (Figure 12.10), a stack frame includes a return address, passed parameters, a frame pointer, and perhaps other bookkeeping information When the called procedure returns to the calling procedure, the stack from for the called procedure is popped from the stack.

The exact contents and organization of a stack frame depend on the implementation. For our purposes, we consider a "model" stack frame that includes the following elements, illustrated with the top of the stack at the top of the diagram:



Associated with the stack are a stack pointer, which points to the current top element of the stack, and a frame pointer, which points to the old frame pointer field in the top frame. When a call is made from the current program, a new frame is created by:

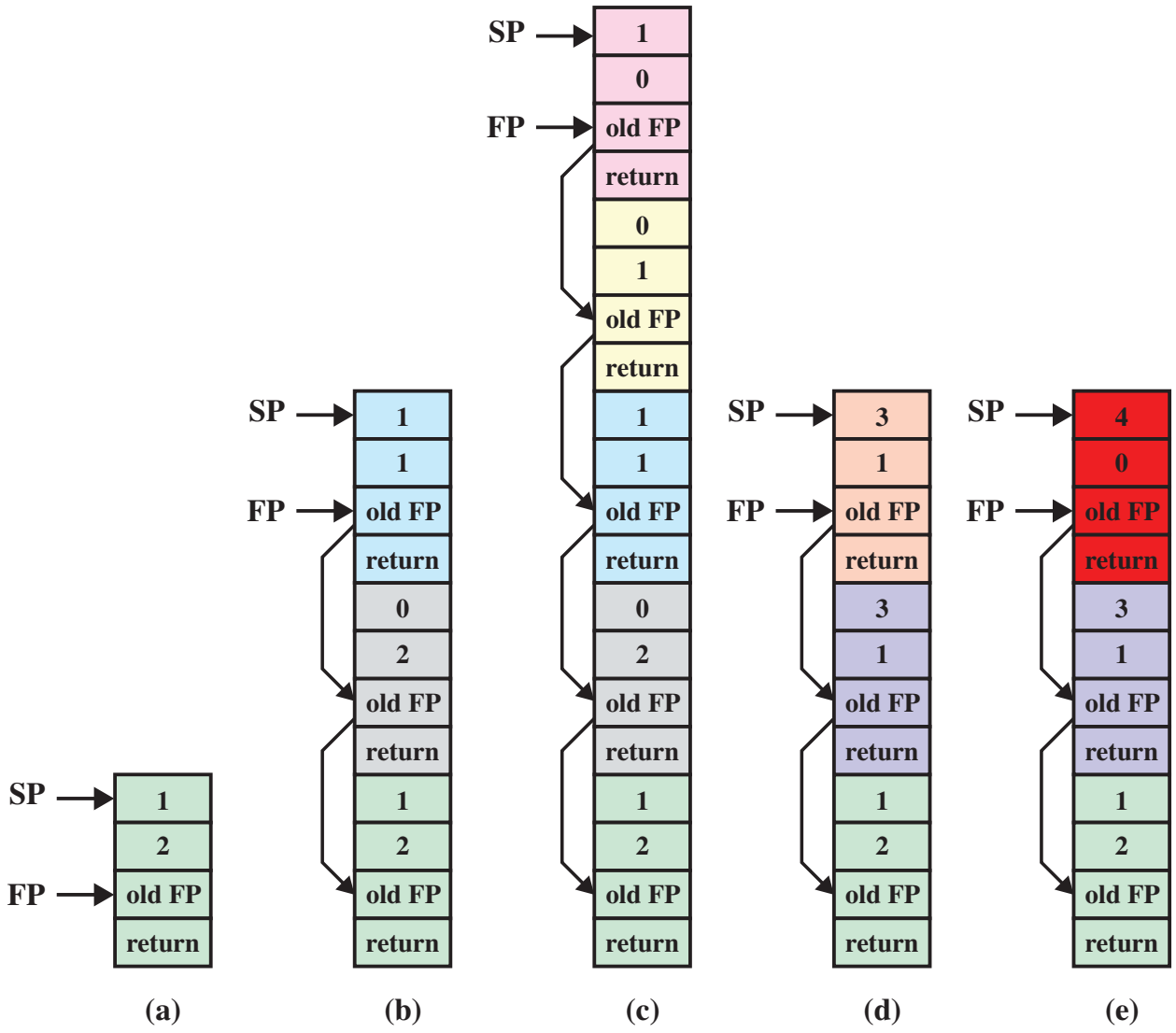
1. Push the return address of the calling program.
2. Push the current frame pointer.
3. Push all parameters to be passed to the called program.

The called program my access the passed parameters. The called program may call another program, resulting in a new stack frame being pushed onto the stack. When the called program returns:

1. Pop all the elements of the current frame from the stack.
2. If any parameters are to be returned to the calling program, push these onto the stack, where they become part of the calling program's stack frame.
3. Update the stack pointer to the new top of the stack.
4. Update the frame pointer to the value that was in the old frame pointer field that has just been popped from the stack.
5. Resume execution at the return address that has just been popped from the stack.

The preceding technique works whether the programs are distinct or all the same. That is, the same technique works with a recursive program that calls itself as well as the more usual situation.

Figure H.4 shows the control stack at the labeled points in Figure H.2b for the function Ackermann (2, 1).



SP = stack pointer
FP = base pointer

Figure H.4 Snapshots of Stack During Execution of Figure H.2b

H.4 RECURSION AND ITERATION

It can be shown that any recursive definition of an algorithm can be rewritten using only iteration, that is, using only loop constructs [RICE65]. In many cases, the recursive definition is more compact in a programming language that allows recursion, and is more understandable, particularly if the function operates on a recursive data structure, such as a tree. However, typically, a recursive program uses both more memory and more processing time than an equivalent iterative solution.

As an example, consider the factorial function, which we defined as:

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

This can easily be rewritten iteratively as:

```
int factorial(int n)
{
   int nfactorial;
   for (nfactorial = 1; n != 0; --n)
     nfactorial *= n;
   return nfactorial;
}
```

Now, to compute factorial(n) recursively would require *n* instances of the variable to be created on the control stack and *n* calls and returns, which generates *n* stack frame creations and deletions. In the iterative solution, there is a single variable, and only one stack frame created.

However, there are many cases in which the recursive technique is more natural and easier to program. Consider the binary search algorithm introduced earlier. To repeat, the recursive version is as follows:

```
int BinarySearch(int sortedArray[], int key, int first, int last) {
    if (first <= last) {
       int mid = (first + last) / 2;
       if (key == sortedArray[mid])
          return mid;                                        /* found */
       else if (key < sortedArray[mid])
           return BinarySearch(sortedArray, key, first, mid-1);
       else                                  /* key > sortedArray[mid] */
           return BinarySearch(sortedArray, key, mid+1, last);
   }
   return -(first);                                       /* not found */
}
```

Here as an iterative version:

```
int BinarySearch(int sortedArray[], int key, int first, int last) {
    int low = first;
    int high = last;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (key == sortedArray[mid])
           return mid;                                      /* found */
        else if (key < sortedArray[mid])
           high = mid - 1;
         else                                /* key > sortedArray[mid] */
           low = mid + 1;
   }
    return -(first);                                       /* not found */
}
```

The iterative algorithm requires two temporary variables, and even given knowledge of the algorithm it is more difficult to understand the process by simple inspection, although the two algorithms are very similar in their steps.

Table H.1 summarizes some of the differences between recursion and iteration.

References

HAYN95 Haynes, S. "Explaining Recursion to the Unsophisticated." *SIGSCE Bulletin,* September 1995.

MCCR87 McCracken, D. "Ruminations on Computer Science Curricula." *Communications of the ACM*, January 1987.

RICE65 Rice, H. "Recursion and Iteration." " *Communications of the ACM*, February 1965.

Table H.1 Properties of Iteration and Recursion

Property	Iteration Loop	Recursive Procedure
Repetition	Execute the same block of code repeatedly to obtain the result; signal the intent to repeat by looping back to the loop entrance.	Execute the same block of code repeatedly to obtain the result; signal the intent to repeat by calling itself.
Terminating conditions	In order to guarantee that it will terminate, a loop must have one or more conditions that cause it to terminate and it must be guaranteed at some point to hit one of these conditions.	In order to guarantee that it will terminate, a recursive function requires a base case that causes the function to stop recursing.
State	Current state is updated as the loop progresses	Current state is passed as parameters