CHAPTER 10

Recursion

Recursion is a process for solving problems by subdividing a larger problem into smaller cases of the problem itself and then solving the smaller, more trivial parts. Recursion is a powerful programming and problem-solving tool. It can be used with a wide range of problems from basic traditional iterations to the more advanced backtracking problems. While recursion is very powerful, recursive solutions are not always the most efficient. In some instances, however, recursion is the implementation of choice as it allows us to easily develop a solution for a complicated problem that may otherwise be difficult to solve.

10.1 Recursive Functions

A function (or method) can call any other function (or method), including itself. A function that calls itself is known as a ***recursive function***. The result is a virtual loop or repetition used in a fashion similar to a `while` loop.

Consider the simple problem of printing the integer values from 1 to n in reverse order. The iterative solution for this problem is rather simple when using a loop construct. But it can also be solved recursively, that is, using a recursive function. Suppose we have implemented the following recursive function:

```
def printRev( n ):
  if n > 0 :
    print( n )
    printReverse( n-1 )
```

and we call the function with an argument of 4:

```
printRev( 4 )
```

The current sequential flow of execution is interrupted and control is transferred to the `printRev()` function with a value of 4 being assigned to argument `n`. The body of `printRev()` begins execution at the first statement. Since 4 is greater than 0, the body of the `if` statement is executed. When the flow of execution reaches the `printRev(3)` function call, the sequential flow is once again interrupted as control is transferred to another instance of the `printRev()` function. The body of this instance begins execution at the first statement, but this time with $n = 3$. Figure 10.1(a) illustrates the execution of the recursive function as a group of boxes with each box representing a single invocation of the `printRev()` function. The boxes contain the contents of local variables and only the statements of the function actually executed. Each recursive call to `printRev()` is shown inside its own box with the boxes positioned at the point where the function was invoked.

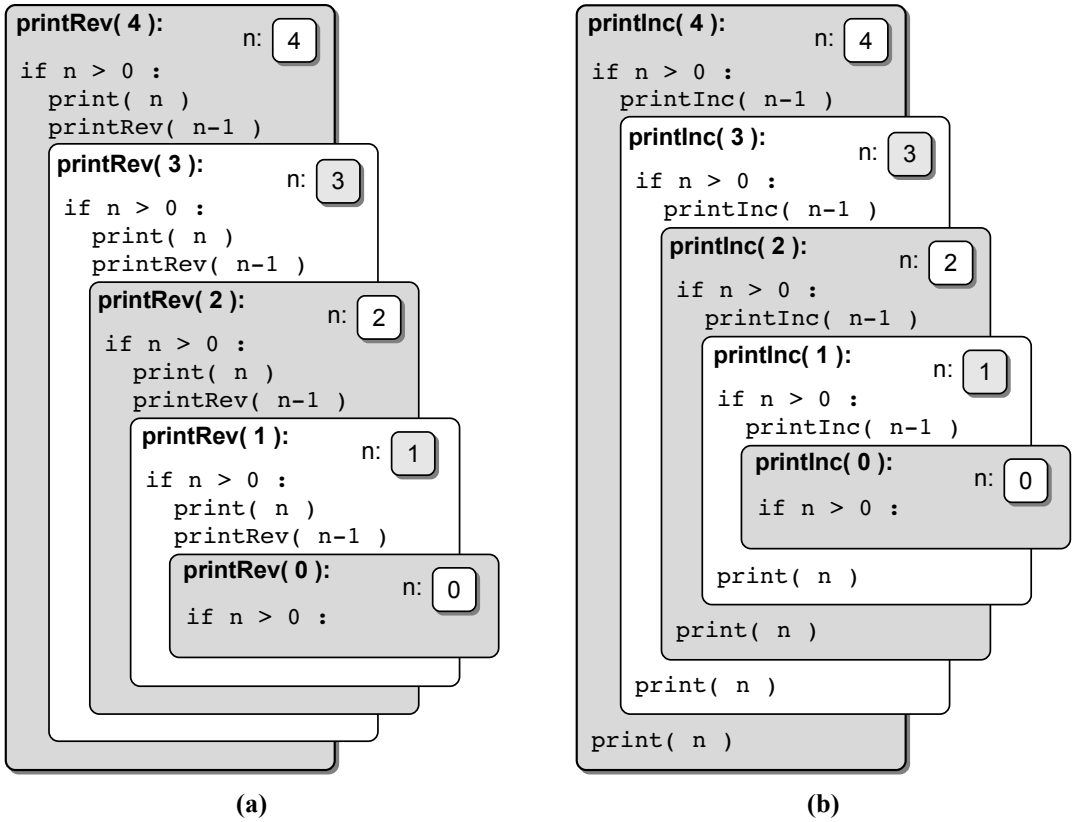


Figure 10.1: Recursive flow of execution: (a) `printRev()` and (b) `printInc()`.

These ***recursive calls*** continue until a value of zero is passed to the function, at which time the body of the `if` statement is skipped and execution reaches the end of the function. As with any function call, when execution reaches the end of the function or the `return` statement is encountered, execution returns to the location where the function was originally called. In this case, the call to `printRev(0)`

NOTE

Local Variables. Like any other function, each call to a recursive function creates new instances of all local reference variables used within that function. Changing the contents of a local reference variable does not affect the contents of other instances of that variable.

was made from within the `printRev(1)` instance. Thus, execution returns to the first statement after that invocation. After execution returns to the `printRev(1)` instance, the end of that function is reached and again execution is returned to the location where that instance was invoked. The result is a chain of recursive returns or ***recursive unwinding*** back to the original `printRev(4)` function call.

What changes would be needed to create a recursive function to print the same integer values in increasing order instead of reverse order? We can change the location of the recursive call to change the behavior of the recursive solution. Consider a new recursive print function:

```
def printInc( n ):
  if n > 0 :
    printInc( n-1 )
    print( n )
```

In this version, the recursive call is made before the value of `n` is printed. The result is a series of recursive calls before any other action is performed. The actual printing of the values is performed after each instance of the function returns, as illustrated in Figure 10.1(b).

10.2 Properties of Recursion

All recursive solutions must satisfy three rules or properties:

1. A recursive solution must contain a ***base case***.
2. A recursive solution must contain a ***recursive case***.
3. A recursive solution must make progress toward the base case.

A recursive solution subdivides a problem into smaller versions of itself. For a problem to be subdivided, it typically must consist of a data set or a term that can be divided into smaller sets or a smaller term. This subdivision is handled by the recursive case when the function calls itself. In the `printRev()` function, the recursive case is performed for all values of $n > 0$.

The base case is the terminating case and represents the smallest subdivision of the problem. It signals the end of the virtual loop or recursive calls. In `printRev()`, the base case occurred when $n = 0$ and the function simply returned without performing any additional operations.

Finally, a recursive solution must make progress toward the base case or the recursion will never stop resulting in an infinite virtual loop. This progression

typically occurs in each recursive call when the larger problem is divided into smaller parts. The larger data set is subdivided into smaller sets or the larger term is reduced to a smaller value by each recursive call. In our recursive printing solution, this progression is accomplished by subtracting one from the current value of `n` in each recursive function call.

10.2.1 Factorials

The factorial of a positive integer n can be used to calculate the number of permutations of n elements. The function is defined as:

$$n! = n * (n-1) * (n-2) * \ldots * 1$$

with the special case of $0! = 1$. This problem can be solved easily using an iterative implementation that loops through the individual values $[1 \ldots n]$ and computes a product of those values. But it can also be solved with a recursive solution and provides a simple example of recursion. Consider the factorial function on different integer values:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 2 * 1 \\ 3! &= 3 * 2 * 1 \\ 4! &= 4 * 3 * 2 * 1 \\ 5! &= 5 * 4 * 3 * 2 * 1 \end{aligned}$$

After careful inspection of these equations, it becomes obvious each of the successive equations, for $n > 1$, can be rewritten in terms of the previous equation:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 * (1-1)! \\ 2! &= 2 * (2-1)! \\ 3! &= 3 * (3-1)! \\ 4! &= 4 * (4-1)! \\ 5! &= 5 * (5-1)! \end{aligned}$$

Since the function is defined in terms of itself and contains a base case, a recursive definition can be produced for the factorial function as shown here. Listing 10.1 provides a recursive implementation of the factorial function.

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1)!, & \text{if } n > 0 \end{cases}$$

Listing 10.1 The `fact()` recursive function.

```
1 # Compute n!
2 def fact( n ):
3   assert n >= 0, "Factorial not defined for negative values."
4   if n < 2 :
5     return 1
6   else :
7     return n * fact(n - 1)
```

10.2.2 Recursive Call Trees

Figure 10.1 used boxes to represent function invocations and to illustrate the flow of execution for two recursive functions. The specific placement of the boxes illustrated the different results that were achieved depending on the location of the recursive call within the function. This type of illustration can be very helpful to visualize the flow of execution within and between various functions, but it's not as useful in developing and understanding recursive functions.

When developing or evaluating a recursive function, we typically use a ***recursive call tree*** such as the one for the factorial function illustrated in Figure 10.2. The diagram consists of small boxes and directed edges between the boxes. Each box represents a function call and is labeled with the name of the function and the actual arguments passed to the function when it was invoked. The directed edges between the boxes indicate the flow of execution. The solid edges indicate the function from which a call originated. For example, in Figure 10.2, we see the call to `fact(5)` was made from the `main()` function while the call to `fact(4)` was made during the execution of `fact(5)`. The dashed edges indicate function returns and are labeled with the return value if a value is returned to the caller.

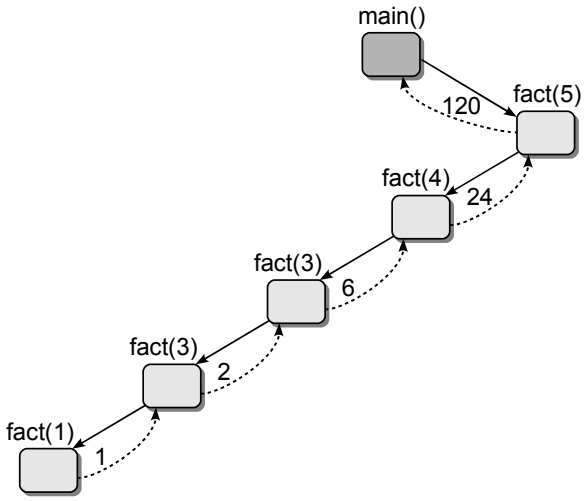


Figure 10.2: Recursive call tree for `fact(5)`.

If a function makes multiple calls to other functions, each function call is indicated in the tree by a box and directed edge. The edges are listed left to right in the order the calls are made. For example, suppose we execute the following simple program that consists of three functions. The resulting call tree is shown in Figure 10.3.

```
# A sample program containing three functions.
def main():
  y = foo( 3 )
  bar( 2 )

def foo( x ):
  if x % 2 != 0 :
    return 0
  else :
    return x + foo( x-1 )

def bar( n ):
  if n > 0 :
    print( n )
    bar( n-1 )

main()
```

Since the main routine makes two function calls, both are indicated as directed edges originating from the `main()` box. You will also notice the `foo()` function makes a recursive call to itself, but the second call is not indicated in the call tree. The reason is during this execution of the program, with the given arguments to `foo()`, the logical condition in the `if` statement evaluates to true and, thus, the recursive call is never made. The call tree only shows the functions actually called during a single execution, which is based on a given set of data, a specific function argument value, or specific user input.

To follow the flow of execution in Figure 10.3, we start with the top box, the one to which no solid directed edges flow into. In this case, that box is the `main()` function. From the main routine, we take the path along the leftmost solid edge

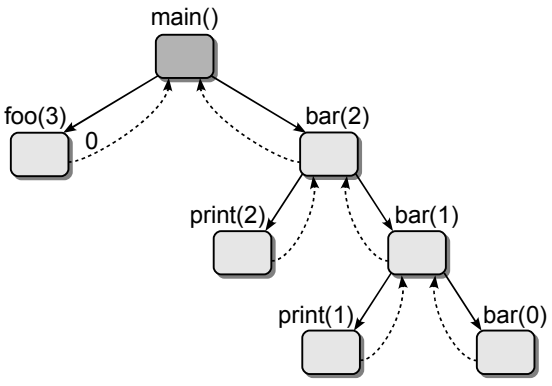


Figure 10.3: Sample call tree with multiple calls from each function.

leading to the `foo(3)` box. Since that function executes the return statement, we follow the dashed edge back to the main routine. Execution continues by following the next edge out of the main routine box, which leads us to the `bar(2)` function box. From there, we continue to follow the directed edges between the boxes and eventually return to the `main()` routine box with no further edges to follow. At that point, execution terminates.

10.2.3 The Fibonacci Sequence

The ***Fibonacci sequence*** is a sequence of integer values in which the first two values are both 1 and each subsequent value is the sum of the two previous values. The first 11 terms of the sequence are:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

The nth Fibonacci number can be computed by the recurrence relation (for $n > 0$):

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

A recursive function for the computing the nth Fibonacci number is shown in Listing 10.2. This function illustrates the use of multiple recursive calls from within the body of the function. The call tree corresponding to the function call `fib(6)` is illustrated in Figure 10.4.

Listing 10.2 The `fib()` recursive function.

```
1 # Compute the nth number in the Fibonacci sequence.
2 def fib( n ):
3   assert n >= 1, "Fibonacci not defined for n < 1."
4   if n == 1 :
5     return 1
6   else :
7     return fib(n - 1) + fib(n - 2)
```

10.3 How Recursion Works

When a function is called, the sequential flow of execution is interrupted and execution jumps to the body of that function. When the function terminates, execution returns to the point where it was interrupted before the function was invoked. But how does it know where to return? We know for sure, it's not magic.

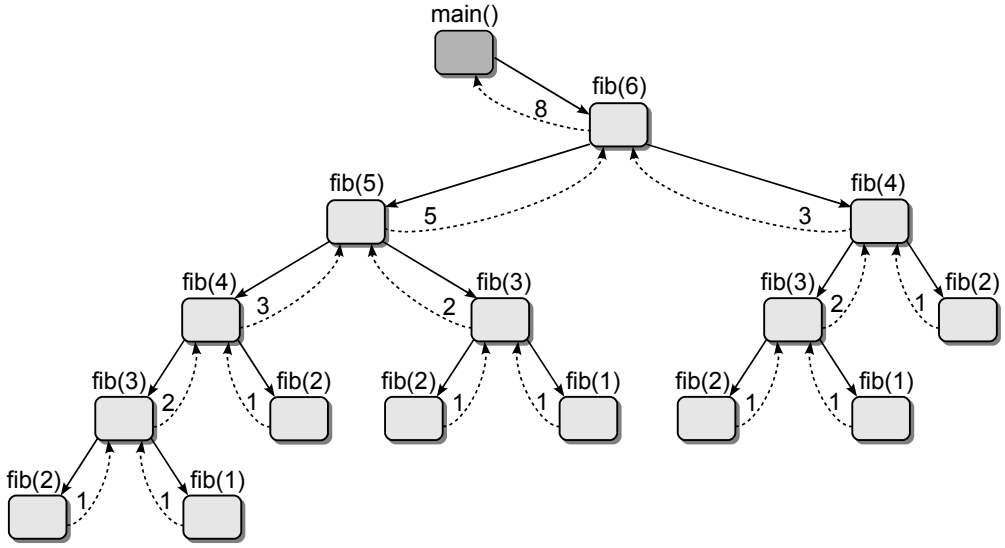


Figure 10.4: Recursive call tree for `fib(6)`.

10.3.1 The Run Time Stack

Each time a function is called, an ***activation record*** is automatically created in order to maintain information related to the function. One piece of information is the ***return address***. This is the location of the next instruction to be executed when the function terminates. Thus, when a function returns, the address is obtained from the activation record and the flow execution can return to where it left off before the function was called.

The activation records also include storage space for the allocation of local variables. Remember, a variable created within a function is local to that function and is said to have local scope. Local variables are created when a function begins execution and are destroyed when the function terminates. The lifetime of a local variable is the duration of the function in which it was created.

An activation record is created per function call, not on a per function basis. When a function is called, an activation record is created for that call and when it terminates the activation record is destroyed. The system must manage the collection of activation records and remember the order in which they were created. The latter is necessary to allow the system to backtrack or return to the next statement in the previous function when an invoked function terminates. It does this by storing the activation records on a ***run time stack***. The run time stack is just like the stack structure presented in Chapter 7 but it's hidden from the programmer and is automatically maintained. Consider the execution of the following code segment, which uses the factorial function defined earlier:

```
def main():
  y = fact( 5 )

main()
```

When the main routine is executed, the first activation record is created and pushed onto the run time stack, as illustrated in Figure 10.5(a). When the factorial function is called, the second activation record is created and pushed onto the stack, as illustrated in Figure 10.5(b), and the flow of execution is changed to that function.

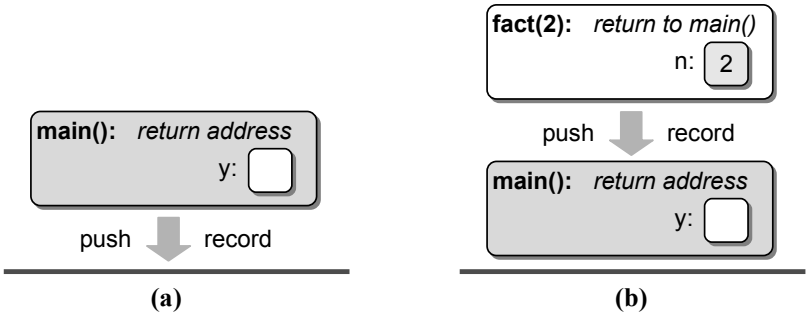


Figure 10.5: The initial run time stack for the sample code segment.

The factorial function is recursively called until the base case is reached with a value of $n = 0$. At this point, the run time stack contains four activation records, as illustrated Figure 10.6(a). When the base case statement at line 5 of Listing 10.1 is executed, the activation record for the function call `fact(0)` is popped from the stack, as illustrated in Figure 10.6(b), and execution returns to the function instance `fact(1)`. This process continues until all of the activation records have been popped from the stack and the program terminates.

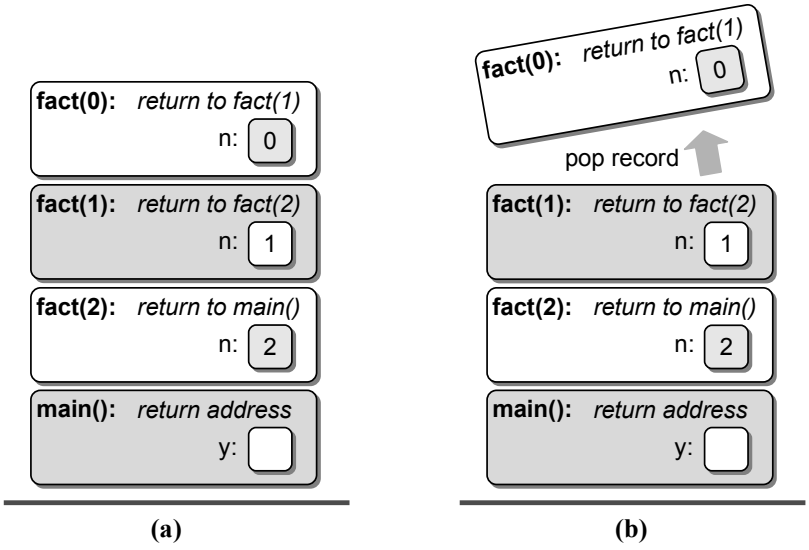


Figure 10.6: The run time stack for the sample program when the base case is reached.

10.3.2 Using a Software Stack

Using recursion in solving problems is very similar to using the software implemented stack structure. In fact, any solution that can be implemented using a stack structure can be implemented with recursion, and vice versa. Consider the problem of printing in reverse order the items stored in a singly linked list such as the one shown in Figure 10.7.

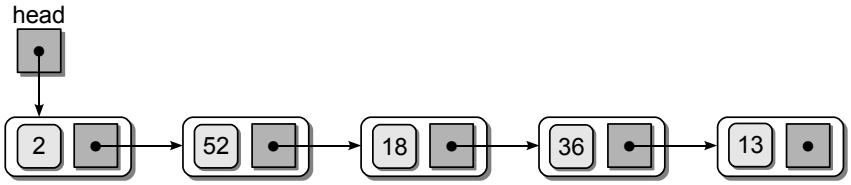


Figure 10.7: A sample singly linked list.

Since the links are in one direction, we cannot easily access the nodes in reverse order. A brute-force approach to solving this problem would be to use nested loops to iterate over the linked list multiple times with each iteration of the inner loop ending one node shorter than previous iteration. This approach is implemented by the function in Listing 10.3, but it has a run time of $O(n^2)$.

Listing 10.3 A brute-force approach for printing a singly linked list in reverse order.

```
# Print the contents of a singly linked list in reverse order.

def printListBF( head ):
   # Count the number of nodes in the list.
  numNodes = 0
  curNode = head
  while curNode is not None :
    curNode = curNode.next
    numNodes += 1

   # Iterate over the linked list multiple times. The first iteration
   # prints the last item, the second iteration prints the next to last
   # item, and so on.
  for i in range( numNodes ):
     # The temporary pointer starts from the first node each time.
    curNode = head

     # Iterate one less time for iteration of the outer loop.
    for j in range( numNodes - 1 ):
      curNode = curNode.next

     # Print the data in the node referenced by curNode.
    print( curNode.data )
```

To provide a more efficient solution to the problem, a stack structure can be used to push the data values onto the stack, one at a time, as we traverse through the linked list. Then, the items can be popped and printed resulting in the reverse order listing. This solution is provided in Listing 10.4 and the resulting stack after the iteration over the list and before the items are popped is shown in Figure 10.8. Assuming the use of the linked list version of the stack, `printListStack()` has a run time of $O(n)$, the proof of which is left as an exercise.

Listing 10.4 Using a stack to print a linked list in reverse order.

```
# Print the contents of a linked list in reverse order using a stack.

from lliststack import Stack

def printListStack( head ):
   # Create a stack to store the items.
  s = Stack()

   # Iterate through the list and push each item onto the stack.
  curNode = head
  while curNode is not None :
    s.push( curNode.data )
    curNode = curNode.next

   # Repeatedly pop the items from the stack and print them until the
   # stack is empty.
  while not s.isEmpty() :
   item = s.pop()
   print item
```

A recursive solution for this problem is also possible. To design the solution, we use the divide and conquer strategy introduced in Chapter 4. With this strategy, you solve the larger problem by dividing it into smaller problems of itself and solving the smaller parts individually. A linked list is by definition a recursive structure. That is, the list can be thought of as a node linked to a sublist of nodes

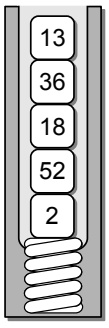


Figure 10.8: The resulting stack after iterating over the linked list from Figure 10.7.

as illustrated in Figure 10.9(a). If we carry this idea further, then each link in the list can be thought of as linking the node to a sublist of nodes, as illustrated in Figure 10.9(b). With this view of the list, we can print the list in reverse order by recursively printing the sublist pointed to by the node and then printing the contents of the node itself.

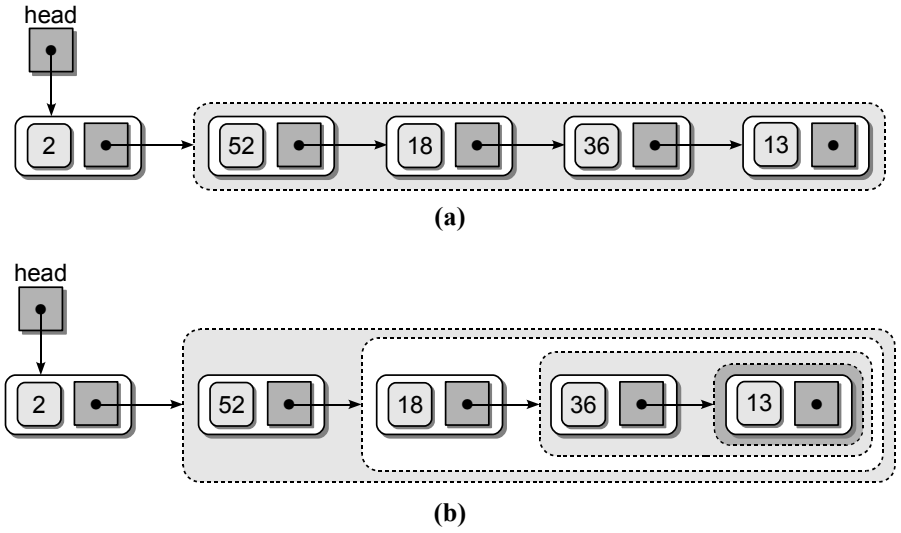


Figure 10.9: Abstract view of the linked list as a node linked to a sublist of nodes.

The solution subdivides the problem into smaller parts of itself and contains a recursive case. But what about the base case, which ends the subdivision and in turn stops the recursive calls? The recursion should stop when the next sublist is empty, which occurs when the link field of the last node is null. A simple and elegant Python implementation for the recursive solution is provided in Listing 10.5.

Listing 10.5 The `printList()` recursive function.

```
1 # Print the contents of a linked list using recursion.
2 def printList( node ):
3   if node is not None :
4     printList( node.next )
5     print( node.data )
```

To help visualize how the `printList()` function works, the call stack and the linked list are illustrated in Figure 10.10. It assumes the main routine is defined as follows:

```
def main():
  head = buildLinkedList()
  printList( head )
```

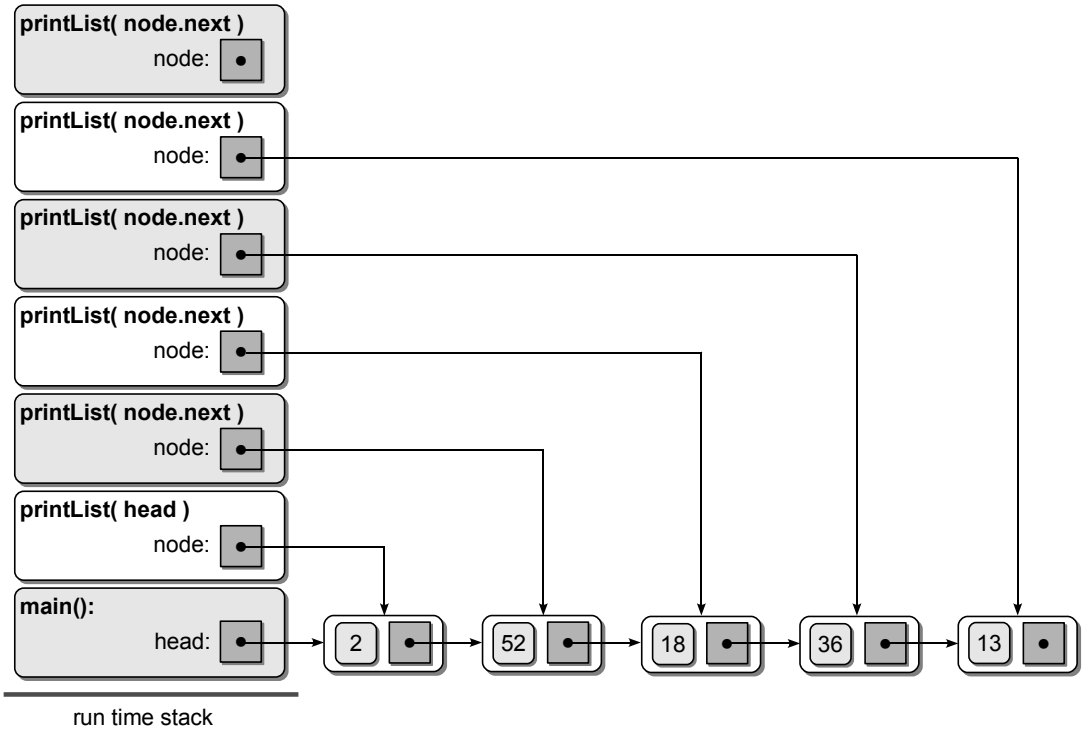


Figure 10.10: The run time stack for the `printList()` function when the base case is reached while processing the linked list from Figure 10.7.

As the recursion progresses and more activation records are pushed onto the run time stack, the corresponding `node` argument is assigned to point to the next node in the list. When the base case is reached, the `node` argument of each of the preceding function calls will point to a different node in the list. Thus, as the recursion unwinds, the contents of each node can be printed, resulting in a listing of the values in reverse order.

Both the `printListStack()` and `printList()` functions provide an implementation to the problem of printing a linked list in reverse order. The former requires the use of an explicit loop for iterating through the list and a stack to store the data for later printing. In the recursive version the loop and stack are implicit. The loop is replaced by the recursive calls while the user-specified stack is replaced by the run time stack on which activation records are pushed for each call to the function.

10.3.3 Tail Recursion

Sometimes an algorithm is easy to visualize as a recursive operation, but when implemented as a recursive function, the solution proves to be inefficient. In these cases, it can be beneficial to use a non-recursive implementation that makes use of a basic iterative loop or the software Stack ADT from Chapter 7.

The main reason for using recursion is to push values onto the run time stack that need to be saved until the recursive operation used to solve the smaller sub-

problem returns. For example, consider the function in Listing 10.5, which printed the contents of a linked list in reverse order. We had to save a reference to each node until the recursive process began unwinding, at which time the node values could be printed. By using recursion, these references were automatically pushed onto the run time stack as part of the activation record each time the function was called. Now suppose we implement a recursive function to print the items of a linked list in order from beginning to end instead of using a simple loop structure:

```
def printInorder( node ):
  if node is not None :
    print( node.data )
    printInorder( node.next )
```

This function eliminates the need for a loop to iterate through the list, but do we have anything to save on the stack until the recursive call returns? The answer is no. When the recursive call returns, the function is finished and it simply returns. This is an example of ***tail recursion***, which occurs when a function includes a single recursive call as the last statement of the function. In this case, a stack is not needed to store values to be used upon the return of the recursive call and thus a solution can be implemented using an iterative loop instead.

10.4 Recursive Applications

There are many applications that can be solved using recursion. Some in fact, can only be solved using recursion. In this section, we introduce some of the classic problems that require the use of recursion or can benefit from a recursive solution.

10.4.1 Recursive Binary Search

The binary search algorithm, which we introduced in Chapter 4, improves the search time required to locate an item in a sorted sequence. We provided an iterative implementation of the binary search algorithm in Chapter 4, but the algorithm can also be implemented recursively since it can be expressed in smaller versions of itself. In searching for a target within the sorted sequence, the middle value is examined to determine if it is the target. If not, the sequence is split in half and either the lower half or the upper half of the sequence is examined depending on the logical ordering of the target value in relation to the middle item. In examining the smaller sequence, the same process is repeated until either the target value is found or there are no additional values to be examined. A recursive implementation of the binary search algorithm is provided in Listing 10.6. As with the earlier version, this version also works with virtual subsequences instead of physically splitting the original sequence. The two arguments, `first` and `last`, indicate the range of elements within the current virtual subsequence. On the first call to the function, these values are set to the full range of the original sequence.

Listing 10.6 A recursive implementation of the binary search algorithm.

```
1  # Performs a recursive binary search on a sorted sequence.
2  def recBinarySearch( target, theSeq, first, last ):
3     # If the sequence cannot be subdivided further, we are done.
4    if first > last :    # base case #1
5      return False
6    else :
7       # Find the midpoint of the sequence.
8      mid = (last + first) // 2
9       # Does the element at the midpoint contain the target?
10     if theSeq[mid] == target :
11       return True      # base case #2
12
13      # or does the target precede the element at the midpoint?
14     elif target < theSeq[mid] :
15       return recBinarySearch( target, theSeq, first, mid - 1 )
16
17      # or does it follow the element at the midpoint?
18     else :
19       return recBinarySearch( target, theSeq, mid + 1, last )
```

We evaluated the binary search algorithm in Chapter 4 and found it required $O(\log n)$ time in the worst case. To determine the run time of a recursive implementation, we must consider the time required to execute a single invocation of the function and the number of times the function is called. In evaluating the time of a single function invocation, we use the same technique as we applied in previous chapters and sum the times of the individual statements. The recursive calls, however, are not included since their times will be computed separately. For the `recBinarySearch()` function, we can quickly determine that each non-recursive function call statement only requires $O(1)$ time.

To help determine the number of times the recursive function is called, we can use its recursive call tree. Consider the recursive call tree for the binary search algorithm, as shown at the top of Figure 10.11, which results when searching for value 8 in the sequence shown at the bottom of the figure. The number of function call boxes in the tree for a given sequence of length n will indicate the total number of times the function is called. There are two recursive calls to `recBinarySearch()` within the function body, but only one will be executed for each invocation. Thus, there will be a single function call box at each level in the call tree and we only have to determine how many levels there are in the call tree when searching a sequence of n items.

The worst case occurs when the target value is not in the sequence, which can be determined when the `first` and `last` markers cross each other with `first > last`. As with the iterative version of the algorithm, the number of elements in the sorted sequence that must be searched is reduced by half each time the function is called. We know from Chapter 4 that repeatedly reducing the input size by half requires $\log n$ reductions in order to reach the case where there are no additional elements to be searched. Thus, the recursive version of the binary search requires $O(\log n)$ time in the worst case.

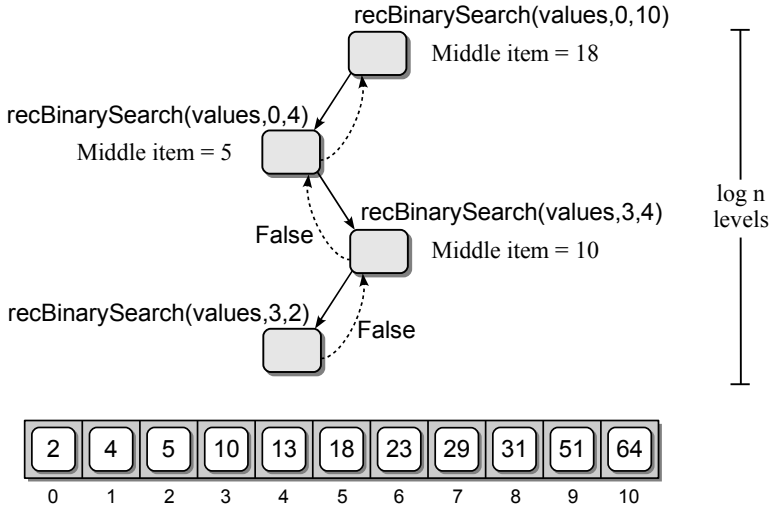


Figure 10.11: The recursive call tree for the binary search algorithm (top) when searching for value 8 in the given sequence (bottom).

10.4.2 Towers of Hanoi

The ***Towers of Hanoi*** puzzle, invented by the French mathematician Edouard Lucas in 1883, consists of a board with three vertical poles and a stack of disks. The diameter of the disks increases as we progress from the top to bottom, creating a tower structure. The illustration in Figure 10.12 shows the board, the three towers, and five disks. Any number of disks can be used with the puzzle, but we use five for ease of illustration.

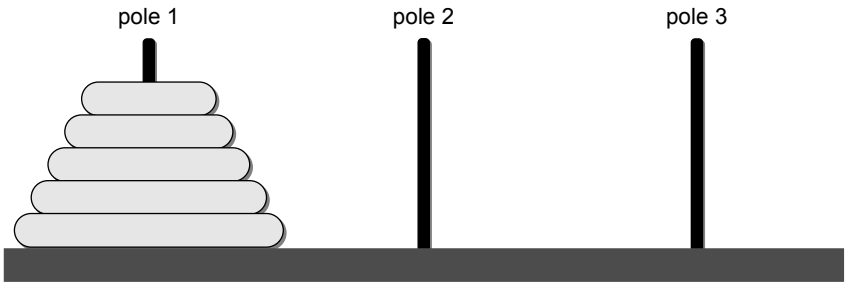


Figure 10.12: The Towers of Hanoi puzzle with five disks.

The objective is to move all of the disks from the starting pole to one of the other two poles to create a new tower. There are, however, two restrictions: (1) only one disk can be moved at a time, and (2) a larger disk can never be placed on top of a smaller disk.

How would you go about solving this problem recursively? Of course you need to think about the base case, the recursive case, and how each recursive call reduces the size of the problem. We will derive all of these in time, but the easiest way to solve this problem is to think about the problem from the bottom up. Instead of thinking about the easiest step, moving the top disk, let's start with the hardest step of moving the bottom disk.

Suppose we already know how to move the top four disks from pole A to pole B, resulting in the board shown in Figure 10.13. Moving the disk from pole A to pole C is now rather easy since it's the only disk left on pole A and there are no disks on pole C. After moving the largest disk, we then move the other four disks from pole B to pole C.

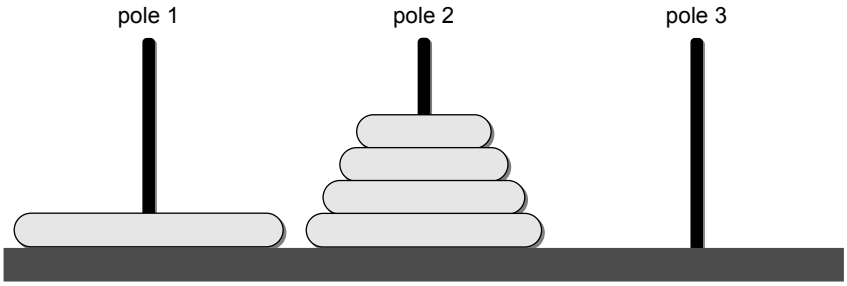


Figure 10.13: The Towers of Hanoi puzzle with four disks moved to pole 2.

Of course, we still have to figure out how to move the top four disks. There is no reason we cannot use the same technique to move the top four disks and in fact we must use the same technique, which leads to a recursive solution. Given n disks and three poles labeled source (S), destination (D), and intermediate (I), we can define the recursive operation as:

- Move the top $n - 1$ disks from pole S to pole I using pole D.
- Move the remaining disk from pole S to pole D.
- Move the $n - 1$ disks from pole I to pole D using pole S.

The first and last steps are recursive calls to the same operation but using different poles as the source, destination, and intermediate designations. The base case, which is implicit in this description, occurs when there is a single disk to move, requiring that we skip the first and last step. Finally, the solution makes progress toward the base case since the recursive calls move one less disk than the current invocation. Eventually, we will end up with a single disk to move.

The high-level solution given above for solving the Towers of Hanoi puzzle can be easily converted to a Python function, as shown in Listing 10.7. For the second step of the process where we actually move a disk, we simply print a message indicating which disk was moved and the two poles it was moved between.

Listing 10.7 Recursive solution for the Towers of Hanoi puzzle.

```
1 # Print the moves required to solve the Towers of Hanoi puzzle.
2 def move( n, src, dest, temp ):
3   if n >= 1 :
4     move( n - 1, src, temp, dest )
5     print( "Move %d -> %d" % (src, dest))
6     move( n - 1, temp, dest, src )
```

To see how this recursive solution works, consider the puzzle using three disks and the execution of the function call:

```
move( 3, 1, 3, 2 )
```

The output produced from the execution is shown here while the first four moves of the disks are illustrated graphically in Figure 10.14.

```
Move 1 -> 3
Move 1 -> 2
Move 3 -> 2
Move 1 -> 3
Move 2 -> 1
Move 2 -> 3
Move 1 -> 3
```

To evaluate the time-complexity of the `move()` function, we need to determine the cost of each invocation and the number of times the function is called for any value of n. Each function invocation only requires $O(1)$ time since there are only two non-recursive function call steps performed by the function, both of which require constant time.

Next, we need to determine how many times the function is called. Consider the recursive call tree in Figure 10.15, which results from the function invocation `move(n, 1, 3, 2)`. The first invocation of `move()` results in two recursive calls, both of which move $n-1$ disks. Both of these invocations each make two recursive calls to move $n-2$ disks. Those four invocations each make two recursive calls to move $n-3$ disks and so on until there is a single disk to be moved.

To determine the total number of times the function is called, we need to calculate the number of times the function executes at each level of the call tree and then sum those values to obtain the final result. The number of function calls at each level is double the number of calls at the previous level. If we label each level of the call tree starting with 0 at the top and going down to $n-1$ at the bottom, then the number of function calls at each level i is equal to 2^i. Summing the number of calls at each level results in the summation:

$$2^0 + 2^1 + 2^2 + \cdots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i$$

or a total of $2^n - 1$ function calls. Thus, the recursive solution for solving the Towers of Hanoi problem requires exponential time of $O(2^n)$ in the worst case.

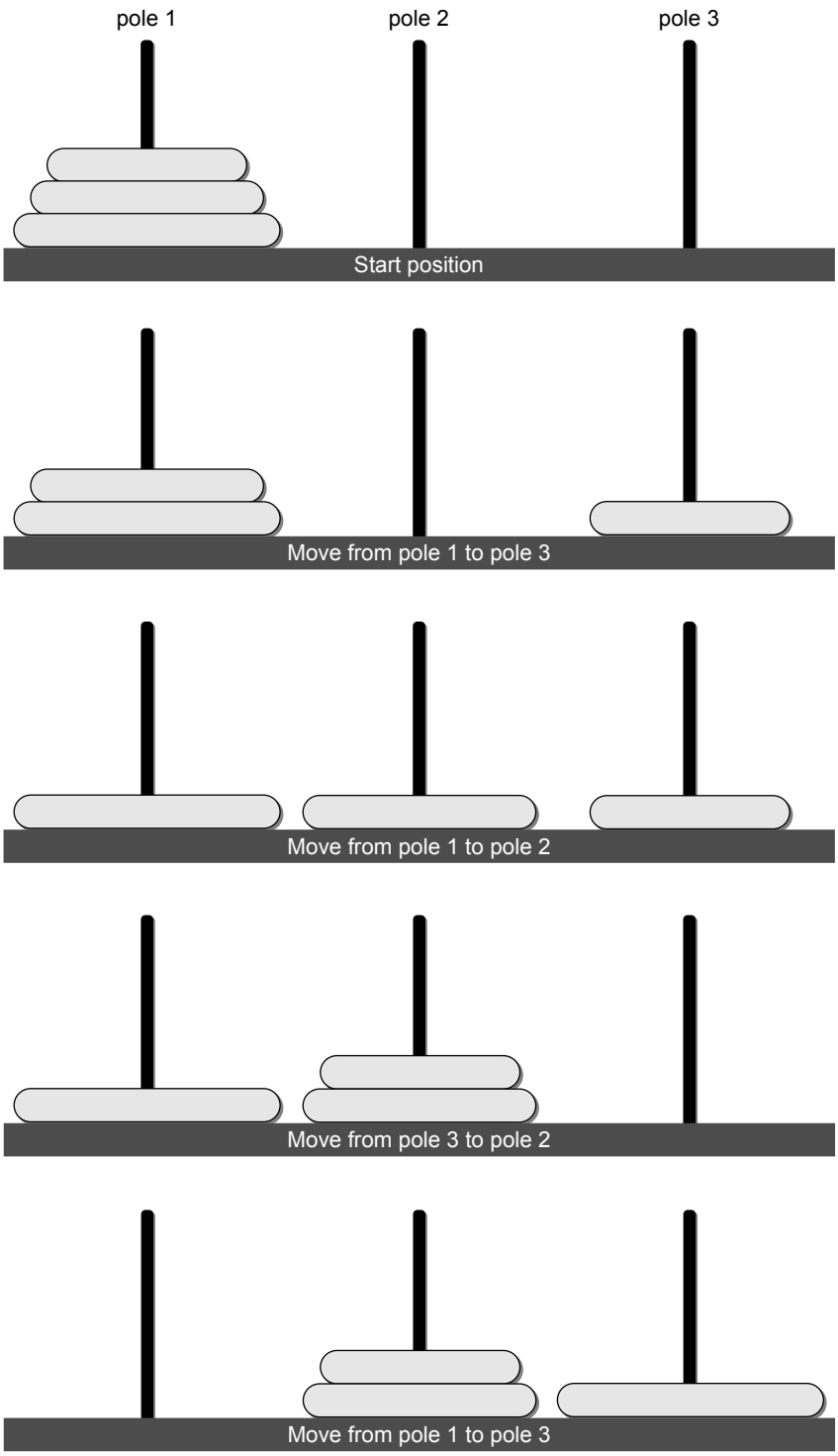


Figure 10.14: The first four moves in solving the Towers of Hanoi puzzle with three disks.

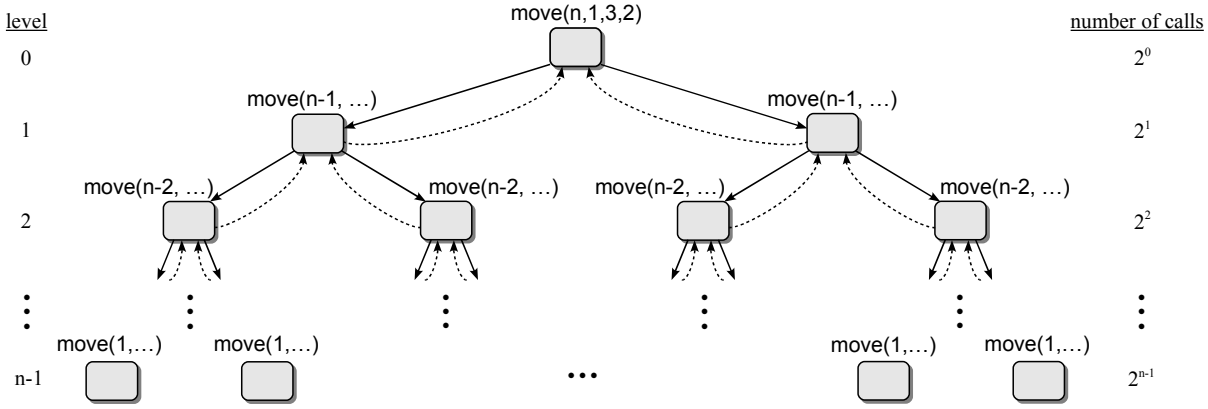


Figure 10.15: The recursive call tree for the Towers of Hanoi puzzle with n disks.

10.4.3 Exponential Operation

Some of the recursive examples we have seen are actually slower than an equivalent iterative version such as computing a Fibonacci number. Those were introduced to provide simple examples of recursion. Other problems such as solving the Towers of Hanoi puzzle can only be done using a recursive algorithm. There are some problems, however, in which the recursive version is more efficient than the iterative version. One such example is the exponential operation, which raises a number to a power.

By definition, the exponential operation x^n can be expressed as x multiplied by itself n times ($x * x * x \cdots x$). For example, $y = 2^8$ would be computed as:

```
2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
```

Of course, in Python this can be done using the exponential operator:

```
y = 2 ** 8
```

But how is this operation actually performed in Python? A basic implementation would use an iterative loop:

```
def exp1( x, n ):
  y = 1
  for i in range( n ):
    y *= x
  return y
```

This implementation requires linear time, which is relatively slow if we are raising a value to a large power. For example, suppose we want to compute 2^{31285}. The basic implementation requires 31,285 iterations to compute this value, but each iteration performs a multiplication, which itself is time consuming when compared to other operations. Fortunately, there is a faster way to raise a value to an integer

power. Instead of computing 2^8 as `2 * 2 * 2 * 2 * 2 * 2 * 2 * 2`, we can reduce the number of multiplications if we computed $(2 * 2)^4$ instead. Better yet, what if we just computed $16 * 16$? This is the idea behind a recursive definition for raising a value to an integer power. (The expression $n/2$ is integer division in which the real result is truncated.)

$$x^n = \begin{cases} 1, & \text{if } n = 0 \\ (x * x)^{n/2}, & \text{if } n \text{ is even} \\ x * (x * x)^{n/2}, & \text{if } n \text{ is odd} \end{cases}$$

Listing 10.8 provides a recursive function for raising x to the integer value of n. Since two of the expressions compute $(x * x)^{n/2}$, we go ahead and compute this value as the `result` on line 5 and then determine if n is even or odd. If n is even, then the result is returned; otherwise, we have to first multiply the result by x to include the odd factor. The run time analysis of `exp()` is left as an exercise.

Listing 10.8 The recursive implementation of `exp()`.

```
1 # A recursive implementation for computing x ** n where n is an integer.
2 def exp( x, n ):
3   if n == 0 :
4     return 1
5   result = exp( x * x, n // 2 )
6   if n % 2 == 0 :
7     return result
8   else :
9     return x * result
```

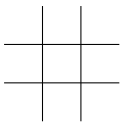
10.4.4 Playing Tic-Tac-Toe

In this technological age, it's very likely you have played a computer game in which you are competing against the computer. For example, you may have played the game of checkers, chess, or something as simple as tic-tac-toe. In any such game, when it's the computer's turn to play, the computer must make a decision as to what play to make. Depending on the game and your level of expertise, you may sometimes think the computer is a genius or that there is some kind of magic going on behind the scenes.

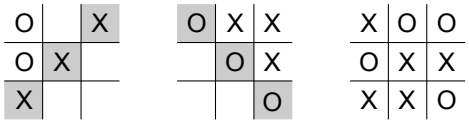
So, how does the computer make its decision? One simple technique the game programmer can apply is the use of a ***game tree***. A game tree provides the sequence of all possible moves that can be made in the game for both the computer and the human opponent. When the computer has to make a move, it can evaluate the game tree and determine its best move. The best move in this case is one that allows the computer to win before its human opponent in the fewest possible moves. Thus, when playing against a computer, it's not that the computer is highly intelligent, but that the computer can evaluate every possible move from the current point to the end of the game and choose the best move. Humans simply

cannot visualize or evaluate this amount of information and instead must rely on experience in attempting to make the best moves.

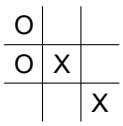
Consider the game of tic-tac-toe in which two players use a board containing nine squares organized into three rows of three columns:



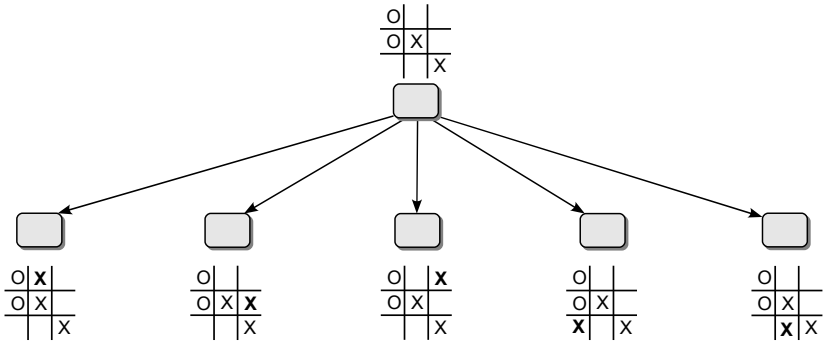
The two players take turns placing tokens of `X`s and `O`s in the squares. One player is assigned the `X`s while the other is assigned the `O`s. Play continues until all of the squares are filled, resulting in a draw, or one of the players wins by aligning three identical pieces vertically, diagonally, or horizontally. The following diagrams show three different game boards, two resulting in wins and one resulting in a draw:



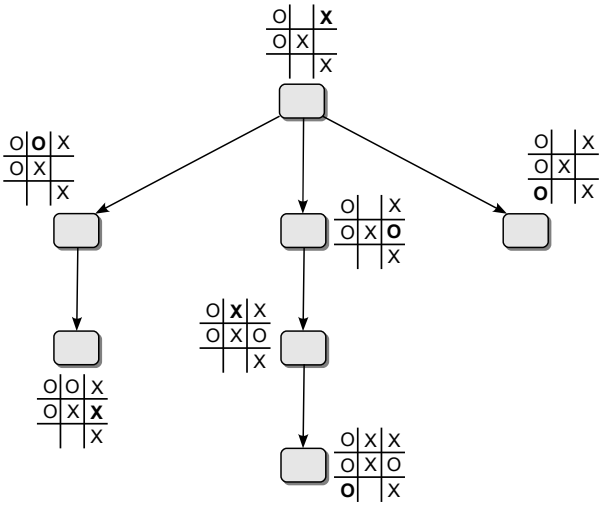
Suppose you are playing a game of tic-tac-toe in which four moves have been played as follows:



and now it's `X`'s turn to play, which happens to be the computer. The computer needs to evaluate all of its possible moves to determine the best move to make, which it can do by evaluating the game tree starting from this point in the game. It can use recursion and build a recursive call tree to represent all possible moves in the game for both itself and its opponent. During the recursion, the tokens are placed on the board for both the computer and its opponent as if they were both actually playing. As the recursion unwinds, the tokens are picked to return the game to its current state. This game tree shows the five possible moves the computer can make at this point:



The computer would need to evaluate all of these moves to determine which would be the best. The decision would be based on which move would allow it to win before its opponent. The next figure shows the part of the game tree that is constructed while evaluating the placement of an `X` in the upper-right square.



Upon evaluating this portion of the tree, the computer would soon learn it could win in two additional moves if its opponent placed their token in the upper-middle square. Following the middle branch from the top, the computer would learn that if its opponent placed their token in the middle-right square instead, it could not win in two more moves. But the opponent could win in three moves in this situation. Finally, it would be determined that the opponent could win in the next move by playing in the lower-left square if the computer made this play. While that's bad, this is only one possible move the computer could make. It still has to evaluate the other possible moves to determine if one is better. Eventually, the computer would determine that the best move would be to play in the lower-left square. This would be based on the fact it could win on the next move by playing in either of two different places before its opponent could win.

Using recursion to build a game tree can make for very interesting games in which a human competes against the computer. We leave as an exercise the implementation of a function to find the best move for the computer in playing tic-tac-toe.

10.5 Application: The Eight-Queens Problem

In Chapter 7, we explored the concept of backtracking and its use in solving certain problems such as that of finding a path through a maze. In that problem, we saw that backtracking allows us to move forward searching for a solution and, when necessary, to back up one or more steps in order to try other options. Backtracking solutions require the use of a stack in order to remember the current solution and to remove the latter parts of that solution when it's necessary to back up and try other options.

In this chapter, we have discovered that function calls and recursion are implemented internally using a run time stack. Thus, the solution to any problem that requires the use of a stack can be implemented using recursion. In this section, we explore the well-known puzzle and classic recursion example known as the Eight-Queens problem. The task is to place eight queens onto a chessboard such that no queen can attack another queen.

In the game of chess, a square board is used consisting of 64 squares arranged in eight rows of eight columns. Each player has a collection of playing pieces that move and attack in fixed ways. The queen can move and attack any playing piece of the opponent by moving any number of spaces horizontally, vertically, or diagonally, as illustrated in Figure 10.16.

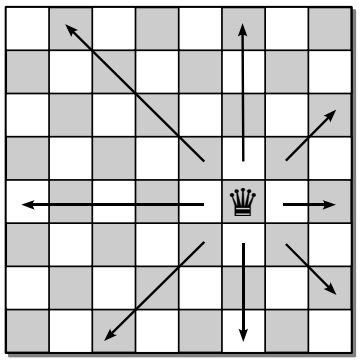


Figure 10.16: Legal moves of the queen in the game of chess.

For the eight-queens problem, we use a standard chessboard and eight queens. The objective is to place the eight queens onto the chessboard in such a way that each queen is safe from attack by the other queens. There are 92 solutions to this problem, two of which are shown in Figure 10.17.

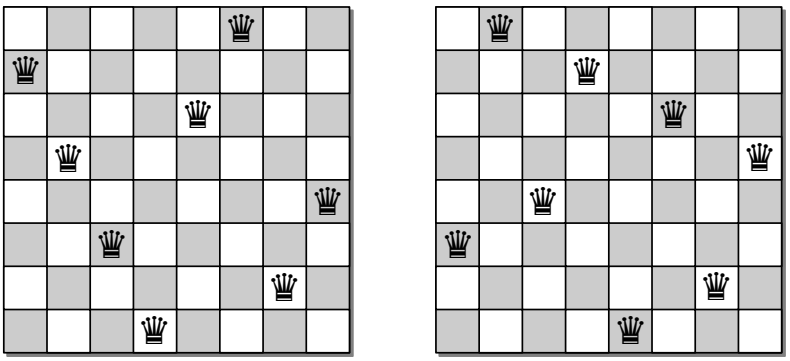
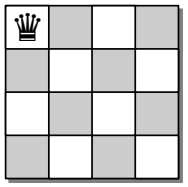


Figure 10.17: Two solutions for the eight-queens problem.

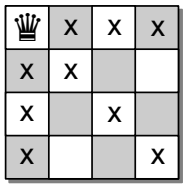
10.5.1 Solving for Four-Queens

To develop an algorithm for this problem, we can first study a smaller instance of the problem by using just four queens and a 4×4 board. How would you go about solving this smaller problem? You may attempt to randomly place the queens on the board until you find a solution that may work for this smaller case. But when attempting to solve the original eight-queens problem, this approach may lead to chaos.

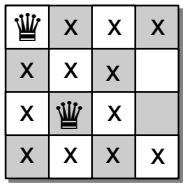
Consider a more organized approach to solving this problem. Since no two queens can occupy the same column, we can proceed one column at a time and attempt to position a queen in each column. We start by placing a queen in the upper-left square or position $(0, 0)$ using the 2-D array notation:



With this move, we now eliminate all squares horizontally, vertically, and diagonally from this position for the placement of additional queens since these positions are guarded by the queen we just placed.



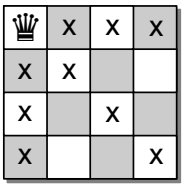
With the first queen placed in the first column, we now move to the second column. The first open position in this column where a queen can be placed without being attacked by the first queen we placed is at position $(2, 1)$. We can place a queen in this position and mark those squares that this queen guards, removing yet more positions for the placement of additional queens.



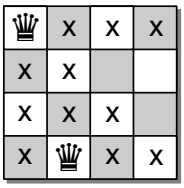
We are now ready to position a queen in the third column. But you may notice there are no open positions in the third column. Thus, the placement of the first two queens will not result in a valid solution. At this point, you may be tempted to remove all of the existing queens and start over. But that would be a drastic move. Instead, we can employ the backtracking strategy as introduced in Chapter 7, in

which we first return to the second column and try alternate positions for that queen before possibly having to return all the way back to the first column.

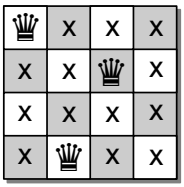
The next step is to return to the second column and pick up the queen we placed at position $(2, 1)$ and remove the markers that were used to indicate the squares that queen was guarding.



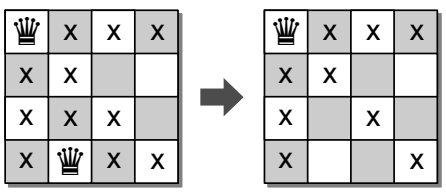
We then place the queen at the next available square within the same column $(3, 1)$ and mark the appropriate squares guarded from this position, as shown here:



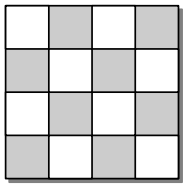
Now we can return to the third column and try again. This time, we place a queen at position $(1, 2)$, but this results in no open squares in the fourth column.



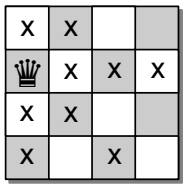
We could try other squares in the same column, but none are open. Thus, we must pick up the queen from the third column and again backtrack to try other combinations. We return to the second column and pick up the queen we placed earlier at position $(3, 1)$ so we can try other squares within this column.



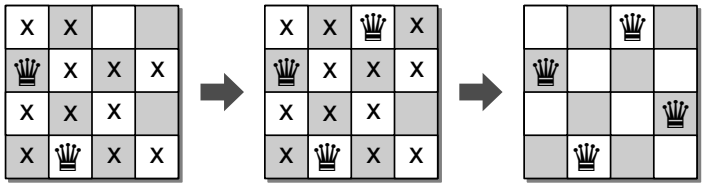
But there are no more open squares in the second column to try, so we must back up even further, returning to the first column. When returning to a column, the first step is always to pick up the queen previously placed in that column.



After picking up the queen in the first column, we place it in the next position $(1, 0)$ within that column.



We can now repeat the process and attempt to find open positions in each of the remaining columns. These final steps, which are illustrated here, results in a solution to the four-queens problem.



Having found a solution for the four-queens problem, we can use the same approach to solve the eight-queens problem. The only difference between the two is that there is likely to be more backtracking required to find a solution. In addition, while the four-queens problem has two solutions, the eight-queens problem has 92 solutions.

The original problem definition only considered finding a solution for a normal 8×8 chessboard. A more general problem, however, is known as the n-queens problem, which allows for the placement of n queens on a board of size $n \times n$ where $n > 3$. The same backtracking technique described earlier can be used with the n-queens problem, although finding a solution to larger-sized boards can be quite time consuming. We leave the analysis of the time-complexity as an exercise.

10.5.2 Designing a Solution

Given the description of the eight-queens problem and the high-level overview of how to find a solution to the four-queens problem, we now consider an implementation for solving this classic example of recursion and backtracking.

The Board Definition

The implementation will consist of two parts: a game board for placing the queens and a recursive function for finding a solution. We begin with the definition of the NQueens Board ADT to represent the board and the placement of the queens.

Define NQueens Board ADT

The *n-queens board* is used for positioning queens on a square board for use in solving the n-queens problem. The board consists of $n \times n$ squares arranged in rows and columns, with each square identified by indices in the range $[0 \ldots n)$.

- `QueensBoard(n)`: Creates an $n \times n$ empty board.
- `size()`: Returns the size of the board.
- `numQueens()`: Returns the number of queens currently positioned on the board.
- `unguarded(row, col)`: Returns a boolean value indicating if the given square is currently unguarded.
- `placeQueen(row,col)`: Places a queen on the board at position (`row`, `col`).
- `removeQueen(row,col)`: Removes the queen from position (`row`, `col`).
- `reset()`: Resets the board to its original state by removing all queens currently placed on the board.
- `draw()`: Prints the board in a readable format using characters to represent the squares containing the queens and the empty squares.

Using the ADT

Given the ADT definition, we can now design a recursive function for solving the n-queens problem. The function in Listing 10.9 takes an instance of the NQueens Board ADT and the current column in which we are attempting to place a queen. When called for the first time, an index value of 0 should be passed to the function.

The function begins by testing if a solution has been found that is one of three base cases. If no solution has been found, then we must loop through the rows in the current column to find an unguarded square. If one is found, a queen is placed at that position (line 10) and a recursive call is made in an attempt to place a queen in the next column. Upon return of the recursive call, we must check to see if a solution was found with the queen placed in the square at position `(row,col)`. If a solution was found, another base case is reached and the function returns `True`. If no solution was found, then the queen in the current column must be picked up (line 18) and another attempt made to place the queen within this column. If all unguarded squares within the current column have been exhausted, then there is no solution to the problem using the configuration of the queens from the previous columns. In this case, which represents the last base case, we must backtrack and allow the previous instance of the recursive function to try other squares within the previous column.

Listing 10.9 The recursive function for solving the n-queens problem.

```
def solveNQueens( board, col ):
   # A solution was found if n-queens have been placed on the board.
  if board.numQueens() == board.size() :
    return True
  else :
     # Find the next unguarded square within this column.
    for row in range( board.size() ):
      if board.unguarded( row, col ):
         # Place a queen in that square.
        board.placeQueen( row, col )
         # Continue placing queens in the following columns.
        if board.solveNQueens( board, col+1 ) :
           # We are finished if a solution was found.
          return True
        else :
           # No solution was found with the queen in this square, so it
           # has to be removed from the board.
          board.removeQueen( row, col )

    # If the loop terminates, no queen can be placed within this column.
    return False
```

Implementing the ADT

Having provided the recursive function for solving the n-queens problem, we leave the implementation of the NQueens Board ADT as an exercise. In this section, however, we discuss possible data structures for representing the actual board.

The most obvious choice is a 2-D array of size $n \times n$. The elements of the array can contain boolean values with `True` indicating the placement of the queens. To determine if a given square is unguarded, loops can be used to iterate over all of the squares to which a queen can move from that position. If a queen is found in any of the squares searched during the loop iterations, then we know the square is currently guarded by at least one queen. The placement and removal of the queens is also quite easy to implement using the 2-D array structure.

As an alternative, we can actually use a 1-D array consisting of n elements. Consider the illustration in Figure 10.18 on the next page, which shows the abstract view of an 8×8 board at the top and a 1-D array at the bottom used to represent the board. Each element of the 1-D array corresponds to one column on the board. The elements of the 1-D array will contain row indices indicating the positions of the queens on the board. Since only one queen can be placed within a given column, we need only keep track of the row containing the queen in the column. When determining if a square is unguarded, we can iterate through the row and column indices for the preceding columns on the board from which the given square can be attacked by a queen. Instead of searching for a `True` value within the elements of a 2-D array, we need only determine if the elements of the 1-D array contain one of the row indices being examined. Consider the illustration in

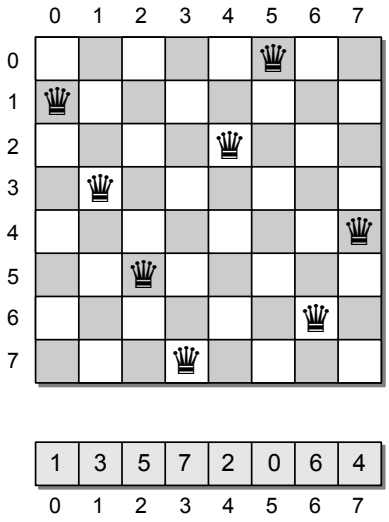


Figure 10.18: Representing an 8×8 board using a 1-D array.

Figure 10.19, in which three queens have been placed and we need to determine if the square at position $(1, 3)$ is unguarded.

When searching horizontally backward, we examine the elements of the 1-D array looking for an index equal to that of the current row. If one is found, then there is a queen already positioned on the current row as is the case in this example. If a queen was not found on the current row, then we would have to search diagonally to the upper left and to the lower left. In these two cases, we search the squares indicated by the arrows and examine the row indices of each and compare them to the entries in the 1-D array. If any of the indices match, then a queen is currently guarding the position and it is not a legal move.

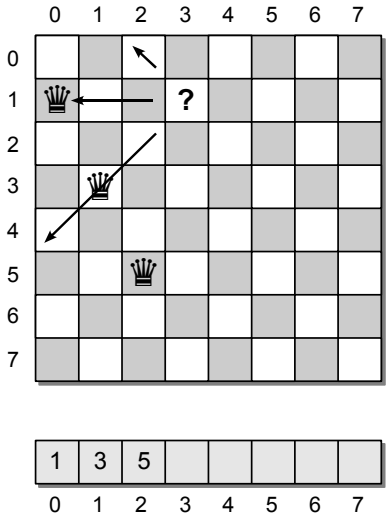


Figure 10.19: Determining if a square is unguarded using a 1-D array.

Exercises

10.1 Draw the recursive call tree for the `printRev()` function from Section 10.1 when called with a value of 5.

10.2 Determine the worst case run time of the recursive factorial function.

10.3 Determine the worst case run time of the recursive Fibonacci function.

10.4 Show or prove that the `printList()` function requires linear time.

10.5 Does the recursive implementation of the binary search algorithm from Listing 10.6 exhibit tail recursion? If not, why not?

10.6 Determine the worst case run time of the recursive exponential function `exp()`.

10.7 Determine the worst case run time of the backtracking solution for the n-queens problem.

10.8 Design and implement an iterative version of the factorial function.

10.9 Design and implement a recursive function for determining whether a string is a palindrome. A ***palindrome*** is a string of characters that is the same as the string of characters in reverse.

10.10 Design and implement a recursive function for computing the greatest common divisor of two integer values.

10.11 Design and implement a program that prints Pascal's triangle:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
```

using a recursive implementation of the binomial coefficients function:

$$a(n, r) = \frac{n!}{r!(n-r)!}$$

10.12 Implement the NQueens Board ADT using the indicated data structure to represent the chess board.

(a) 2-D array

(b) 1-D array

Programming Projects

10.1 Design and implement a program to solve the n-queens problem. Your program should prompt the user for the size of the board, search for a solution, and print the resulting board if a solution was found.

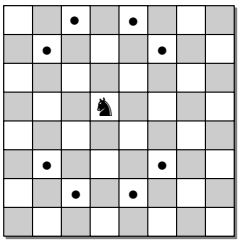
10.2 Instead of finding a single solution to the n-queens problem, we can compute the total number of solutions for a given value of n. Modify the `solveNQueens()` function from Listing 10.9 to count the number of solutions for a given value of n instead of simply determining if a solution exists. Test your program on the following values of n (the number of solutions for the given board size is indicated in parentheses).

(a) 4 (2)
(b) 8 (92)
(c) 9 (352)
(d) 10 (724)
(e) 11 (2680)
(f) 12 (14200)

10.3 Implement a new version of the maze solving program from Chapter 7 to use recursion instead of a software stack.

10.4 Design and implement a program to play tic-tac-toe against the computer using a recursive function to build a game tree for deciding the computer's next move.

10.5 The ***Knight's tour*** problem is another chessboard puzzle in which the objective is to find a sequence of moves by the knight in which it visits every square on the board exactly once. The legal moves of a knight are shown in the diagram to the right. Design and implement a program that uses a recursive backtracking algorithm to solve the knight's tour. Your program should extract from the user a starting position for the knight and produce a list of moves that solves the knight's tour.



10.6 The ***knapsack problem*** is a classic problem in computer science. You are given a knapsack and a collection of items of different weights and your job is to try to fit some combination of the items into the knapsack to obtain a target weight. All of the items do not have to fit in the knapsack, but the total weight cannot exceed the target weight. For example, suppose we want to fill the knapsack to a maximum weight of 30 pounds from a collection of seven items where the weights of the seven items are 2, 5, 6, 9, 12, 14, and 20. For a small number of items, it's rather easy to solve this problem. One such solution, for example, would be to include the items that have weights 2, 5, 9, and 14. But what if we had several thousand items of varying weights and need to fit them within a large knapsack? Design and implement a recursive algorithm for solving this problem.