RECURSION IN DEPTH

After studying this chapter, you should be able to

- Discuss recursion as another form of repetition
- Do the following, given a recursive routine:
  - Determine whether the routine halts,
  - Determine the base case(s),
  - Determine the general case(s),
  - Determine what the routine does,
  - Determine whether the routine is correct and, if it is not, correct it
- Do the following, given a simple recursive problem:
  - Determine the base case(s),
  - Determine the general case(s),
  - Design and code the solution as a recursive void or value-returning function
- Verify a recursive routine, according to the three-question method
- Decide whether a recursive solution is appropriate for a problem
- Compare and contrast dynamic storage allocation and static storage allocation in relation to using recursion
- Explain how recursion works internally by showing the contents of the run-time stack
- Replace a recursive solution with iteration and/or the use of a stack
- Explain why recursion may or may not be a good choice to implement the solution of a problem

What Is Recursion?

- **Recursion:** A power programming technique in which a function calls itself in order to divide work into smaller portions
  - **Recursive call:** When the function being called is the same as the one making the call
- **Direct Recursion:** When a function calls itself
- **Indirect Recursion:** When a chain of function calls leads back to the function that started it

The Classic Example

- The factorial function, n!, is a classic example of recursion in mathematics
- 4! = 4 * 3 * 2 * 1 = 24
- The **recursive definition** is:
  - n! = 1 if n = 0
  - n! = n * (n-1)! if n > 0
  - **4!** = 4 * **3!** = 4 * 3 * **2!** = 4 * 3 * 2 * **1!** =
  - 4 * 3 * 2 * 1 * **0!** = 4 * 3 * 2 * 1 * 1 = **24**

Recursion Terms

- **Recursive definition:** When something is defined in terms of a smaller version of itself
- **Base case:** The case for which the solution can be defined non-recursively (n! = 1 if n = 0)
- **General (or recursive) case:** A case that is defined using recursion (n! = n * (n-1)! if n > 0)
- **Recursive algorithm:** An algorithm described in terms of base cases and general cases

Programming Recursively

- Recursion is a powerful tool when programming
- Recursive solutions are often simpler and more elegant than other solutions
- They allow programmers to solve small, divisible parts of a problem instead of having to wrangle the big picture
- We’ll discuss when to use recursion later in this chapter

Factorial Code Example

```
// Iterative solution
int Factorial(int number) {
  int fact = 1;
  for (int count = 2; count <= number; count++) {
    fact = fact * count;
  }
  return fact;
}
// Recursive solution
int Factorial(int number) {
  if (number == 0)  // Base case
    return 1;
  else  // General case
    return number * Factorial(number-1);
}
```

Recursive vs. Iterative Factorials

- The iterative solution uses a *looping construct* (the *for* loop) while the recursive solution uses a *branching construct* (the if-else statement)
- The iterative solution has two local variables, while the recursive solution only has the parameter
  - Sometimes the recursive solution use local variables, and sometimes they need several parameters

Verifying Recursive Functions

- Walking through the execution of the function is time consuming, tedious, and doesn’t tell us if it works for all possible values
- For example, try tracing the execution of Factorial(4)
- A more general solution is necessary for verifying recursive functions

The Three-Question Method

- **Base-Case Question:** Is there a non-recursive case in the method, and does it work correctly?
- **Smaller-Caller Question:** Does each recursive call involve a smaller case of the problem, eventually leading to the base case?
- **General-Case Question:** Assuming the recursive calls work correctly, does the rest of the function work correctly?
- You should be able to answer yes to all three!

Three Questions for Factorial

- Base-Case Question: The base case occurs when n = 0, and it returns 1 with no further recursion.
- Smaller-Caller Question: Each recursive call subtracts 1 from n. This will eventually reach 0.
- General-Case Question: Assuming Factorial(n-1) returns the correct answer, n * Factorial(n-1) corresponds with the mathematical formula.

Writing Recursive Functions

The general approach is:

- Get an exact definition of the problem
- Determine the size of the problem to be solved in each recursive call
- Identify the base cases
- Identify the general cases and the necessary recursive calls

Example: Searching in a List

- Problem: Check if a particular value is in a list.
- Recursive solution: Return (value is in the first position?) OR (value is in the rest of the list?)
  - How do we search the rest of the list? By recursively calling the ValueInList function!
- Function: bool ValueInList (list, value, index)
- Now consider the process above: What are the base cases and the general cases?

Example: Searching in a List (cont.)

- Base case: The item is at the given index. Return true.
- General case: The item is in the rest of the list. Recursively call ValueInList (list, value, index+1).
- Is that it? What if the item is not in the list?
- Base case: The end of the list has been reached. (index == length – 1) Return false.

Example: Searching in a List (cont.)

```
bool ValueInList(ListType list, int value,
int startIndex)
{
  if (list.info[startIndex] == value)
      return true;  // Base case 1
  else if (startIndex == list.length-1)
      return false; // Base case 2
  else return ValueInList(list, value,
startIndex+1);
}
```

Example: Searching in a List (cont.)

Does this function pass the Three Questions?

- Base Case: The value is found, and the function returns true. In the second case, the search reaches the end of the list and returns false.
- Smaller Caller: Each call increments the index.
- General Case: The base cases are simple enough to be obviously correct, so the general case will proceed while the value is not found and the rest of the list needs to be checked.

Simplifying Solutions with Recursion

- How many combinations of 5 books can be made from a group of 20 books? (“n choose k”)
- Size: Size of the group (n) and number of members in each group (k)
- Base Case: if members == 1, return group
- Base Case: if members == group, return 1
- General Case: Return Combinations (group-1, members-1) + Combinations (group-1, members)

Combinations Code

```
int Combinations(int group, int members)
// Pre: group and members are positive.
// Post: Function value = number of combinations of
// members size that can be constructed from the total
// group size.
{
  if (members == 1)
   return group;  // Base case 1
  else if (members == group)
    return 1;     // Base case 2
  else
    return (Combinations(group-1, members-1) +
            Combinations(group-1, members));
}
```

Combinations

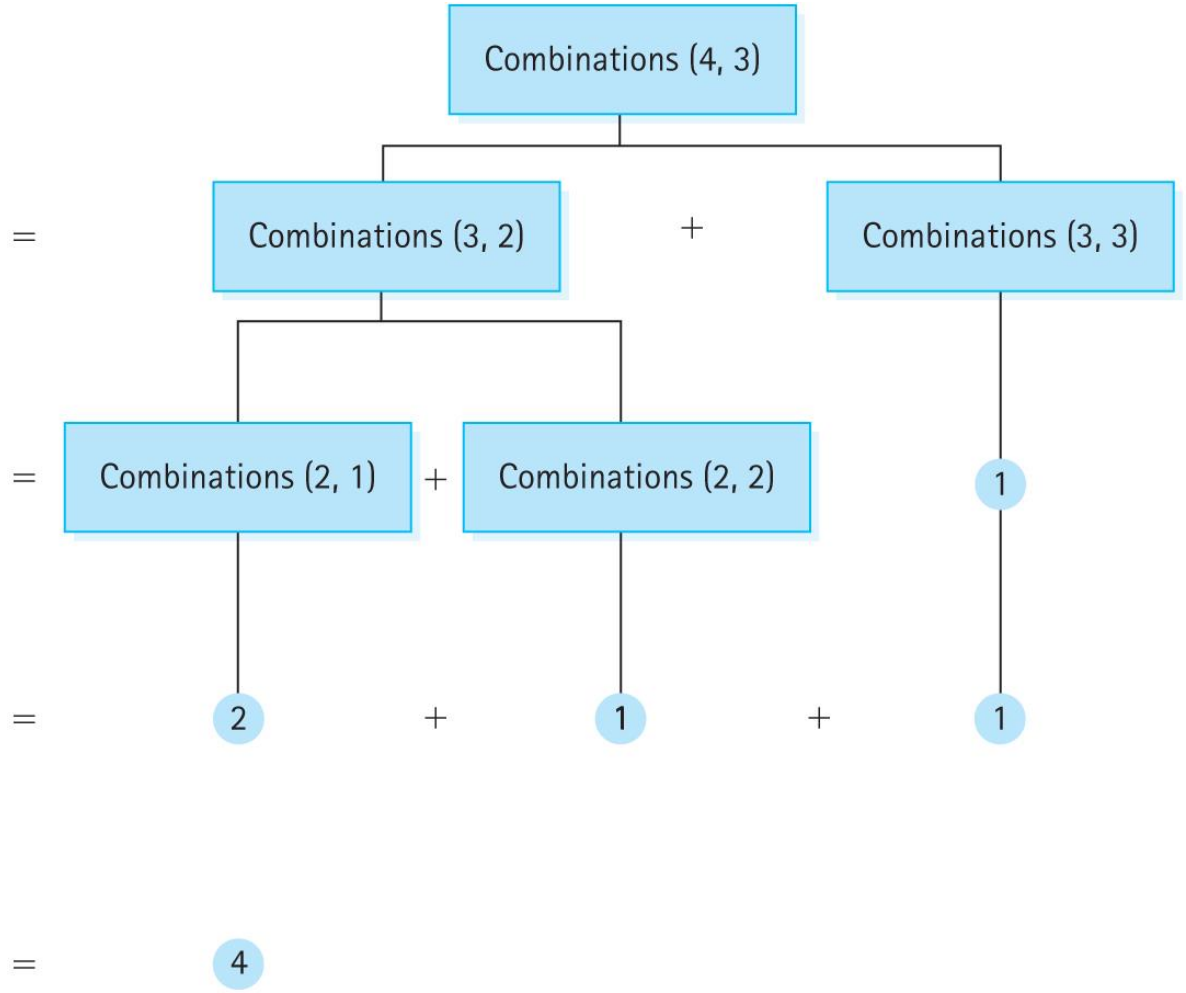


Figure 7.2 Calculating Combinations(4,3)

Recursive List Processing

- Another example of recursion: Printing the items of a linked list.
- This seems trivial. Instead, let’s try printing the list in reverse.
- Is this possible with iteration? Yes, if there are methods for iterating backwards through the list.
- Recursion doesn’t need that.

Problem Definition

- Problem: Print the elements of a list in reverse order.
- Size: The number of elements in the list.
- Base Case: If the list is empty, do nothing.
- General Case: RevPrint the list pointed to by listPtr->next, then print listPtr->info

RevPrint Code

```
// Prints the list in reverse order
void RevPrint(NodeType* listPtr){
  if (listPtr != NULL) {
    RevPrint(listPtr→next);
    std::cout << listPtr->info <<
      std::endl;
  }
}
```

RevPrint: Three Questions

- Base Case: Not explicit in the code. Nothing is done if the list pointer is NULL.
- Smaller-Caller: Each recursive call walks another step down the list, heading towards the final NULL pointer.
- General Case: Assuming the recursive call works correctly, the rest of the list will have been printed before the current value is.

Recursive Binary Search

- Chapter 4 showed an iterative binary search
- Recall that binary search has a list to search, a target value, and the indices to search between
- It is essentially a recursive algorithm: Each step of the binary search algorithm cuts the search area in half by changing the indices

Recursive Binary Search (cont.)

- Problem: Search the area for the target value
- Size: The number of items between the two indices
- Base Cases:
  - The target value is in the middle of the list
  - There is no more area to search (end > start)
- General Case: Search the lower or higher half of the list depending on the middle value

Recursive PutItem and DeleteItem

- Items at the very beginning or end of the list are edge cases for these two functions
- Recursion lets us program them as if we only have the edge cases: Either insert/remove the front of the list, or insert into an empty list
- Instead of being recursive themselves, write two recursive helper functions, Insert and Delete

Recursive List Insertion

- Problem: Insert an item into a sorted list
- Size: The number of items in the list
- Base Cases:
  - If the list is empty, insert the item
  - If the item is less than listPtr->info, insert the item as the first node
- General Case: Insert(listPtr->next, item)

Recursive Item Deletion

- Problem: Delete a target item from the list
- Size: The number of items in the list
- Base Case: If listPtr->info == item, delete the node pointed to by listPtr
- General Case: Delete(listPtr->next, item)

Recursive PutItem and DeleteItem

- Of course, both functions must update the list's links
- Insert and Delete need a *reference* to the list pointer in order to be able to update the list
  - **`NodeType<ItemType>*& listPtr`**
- Otherwise, any changes to listPtr will not be carried through to the caller

Recursive Insert

```
template<class ItemType>
void Insert(NodeType<ItemType>*& listPtr, ItemType item)
{
  if (listPtr == NULL || item < listPtr→info)
  {
    // Save current pointer.
    NodeType<ItemType>* tempPtr = listPtr;
    // Get a new node.
    listPtr = new NodeType<ItemType>;
    listPtr->info = item;
    listPtr->next = tempPtr;
  }
  else Insert(listPtr->next, item);
}
```

How Recursion Works

- Some programming languages do not support recursion. Why?
- To understand this, let's detour and talk about **binding**, or associating a memory address with a variable name.
- Binding can occur at different times.
  - **Static binding** occurs at compile time
  - **Dynamic binding** occurs at run time

Static Binding

- When a program is compiled, each variable is entered into a *symbol table* and bound to a memory address
- Each reference to the variable name is replaced by the memory address
- Function parameters and local variables are also bound to memory addresses at this time
- This is also called *static storage allocation*

Symbol Table

Three variables are declared: `int a, b, c;`

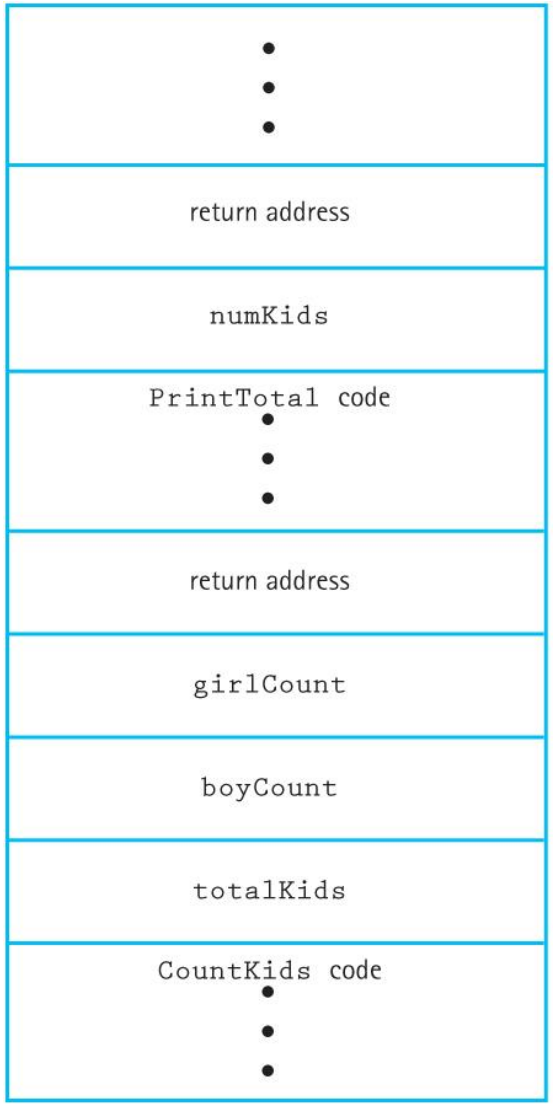
The compiler binds them in the symbol table:

Symbol	Address
a	0001
b	0002
c	0003

A statement such as: `a = b + c;`

Becomes: “Get value stored at 0002; Get the value stored at 0003; Store the sum in 0001”

Static Binding: Memory Layout



Main Function

Second Function

```
void PrintTotal (int numKids)
{ // no local variables
    . . .
}
```

Third Function

```
void CountKids (int girlCount,
     int boyCount)
{
     int totalKids;
     . . .
}
```

Figure 7.4 Static allocation of space for a program with three functions

Static Binding and Recursions

- Each parameter and local variable has a single, fixed memory address associated with it
- A recursive function call would overwrite the variables of the previous call
- Each recursive call needs memory to store its own variables
- Therefore languages that only have static storage allocation cannot support recursion

Dynamic Binding

- **Dynamic Storage Allocation:** Variable names are bound to memory addresses at run time.
- The compiler references variables by their offsets instead of by exact addresses. The correct addresses are calculated during execution using these offsets.
- But where are the variables and parameters for functions stored?

Activation Records

- Functions need to store local variables, parameters, and return address
  - Return address: Where execution returns to, once execution of the function has completed
- **Activation Record:** A record used at run time to store information about a function, such as its variables, parameters, return address, and register values
  - Also called a **stack frame**

Activation Records (cont.)

- Every function call creates a new activation record
- The activation record is deleted once the call is complete
- The code for creating and releasing activation records is inserted automatically by the compiler

Example: Factorial Stack Frame

Table 7.2 Run-Time Version of `Factorial` (Simplified)

What Your Source Code Says	What the Run-Time System Does
`int Factorial(int number)`	`// Function prologue`
`{`	`actRec = new ActivationRecordType;`
	`actRec->returnAddr = retAddr;`
	`actRec->number = number;`
	`// actRec->result is undefined`
`if (number == 0)`	`if (actRec->number == 0)`
`  return 1;`	`  actRec->result = 1;`
`else`	`else`
`  return number *`	`  actRec->result =`
`    Factorial(number - 1);`	`    actRec->number *`
	`    Factorial(actRec->number-1);`
`}`	`// Function epilogue`
	`returnValue = actRec->result;`
	`retAddr = actRec->returnAddr;`
	`delete actRec;`
	`Jump (goto) retAddr`

Table 7.2 Run-time Version of Factorial (Simplified)

Run-Time Stack

- Activation records are pushed onto the **run-time stack** when the function is called
- Each call pushes another record onto the stack
- This gives recursive function calls their own memory for storing variables
- It also clearly shows the order of recursive functions calls

Run-Time Stack of Factorial(4)

- Let’s walk through an example using Factorial
- At address 5200 is the statement:

```
answer = Factorial(4);
```

- This starts the recursive algorithm by pushing an activation record onto the stack
- The recursive call inside Factorial is at address 1010:

```
return number * Factorial(number-1);
```

Run-Time Stack of Factorial(4) (cont.)

Call Number	Number	Result	Return address
1	4	???	5200

The activation record contains the parameter (4), the result (unknown), and the return address (5200)

Run-Time Stack of Factorial(4) (cont.)

Call Number	Number	Result	Return address
1	4	???	5200
2	3	???	1010

- The recursive calls continue, since the base case hasn’t been reached (“number == 0”)

Run-Time Stack of Factorial(4) (cont.)

Call Number	Number	Result	Return address
1	4	???	5200
2	3	???	1010
3	2	???	1010

Run-Time Stack of Factorial(4) (cont.)

Call Number	Number	Result	Return address
1	4	???	5200
2	3	???	1010
3	2	???	1010
4	1	???	1010

Run-Time Stack of Factorial(4) (cont.)

Call Number	Number	Result	Return address
1	4	???	5200
2	3	???	1010
3	2	???	1010
4	1	???	1010
5	0	1	1010

- Here, the base case has been reached
- We start popping the run-time stack, propagating the result back to the original call

Run-Time Stack of Factorial(4) (cont.)

Call Number	Number	Result	Return address
1	4	???	5200
2	3	???	1010
3	2	???	1010
4	1	1	1010

- The result is calculated...

Run-Time Stack of Factorial(4) (cont.)

Call Number	Number	Result	Return address
1	4	24	5200

- Until (skipping a few steps) the final result is calculated (24) and returned to address 5200
- The result is stored at **`answer`** and the main method continues executing

Recursion Depth

- The number of recursive calls constitutes the **depth of recursion**
- Since recursion is another way of performing iterations, recursion depth can be thought of in terms of Big-O notation
- The number of iterations and the recursion depth are both based on the size of the problem

Tracing Recursive Insert

```
template<class ItemType>
void Insert(NodeType<ItemType>*& listPtr, ItemType item)
{
  if (listPtr == NULL || item < listPtr->info)
  {
    // Save current pointer.
    NodeType<ItemType>* tempPtr = listPtr;
    // Get new node.
    listPtr = new NodeType<ItemType>;
    listPtr->info = item;
    listPtr->next = tempPtr;
  }
  else Insert(listPtr->next, item);
}
```

Tracing Recursive Insert (cont.)

Call number	listPtr	item	tempPtr	returnAddr
1	010	11	???	R0

- listPtr points to the list [7, 9, 13, 20]
- Call: Insert(listPtr, 11)
- 11 > 7, so we drop into the else statement

Tracing Recursive Insert (cont.)

Call number	listPtr	item	tempPtr	returnAddr
1	010	11	???	R0
2	014	11	???	R1

- listPtr points to the list [9, 13, 20]
- 11 > 9, so continue to the next call

Tracing Recursive Insert (cont.)

Call number	listPtr	item	tempPtr	returnAddr
1	010	11	???	R0
2	014	11	???	R1
3	018	11	020	R2

- listPtr points to the list [9, 13, 20]
- 11 < 13, so the element is inserted
- The stack will then unwind and return to the original caller

Tracing Recursive Insert (cont.)

Before the recursive call, the list looked like:
Insert the image of the list at the bottom of page 465 here

During call 3, listPtr and tempPtr point to:
Insert the image in the middle of page 467 here

Recursive Quick Sort

- Quick Sort is an algorithm inspired by the idea that it is easier to sort two smaller lists than one large one: “divide and conquer.”
- This is a recursive strategy. The list is continually split into smaller lists, which are then passed to Quick Sort again.
- Recursion stops when the lists cannot be split, i.e., there is only one or zero elements left.

Quick Sort Algorithm

- If there is more than one element in the list:
  - Select a value to split the list on (splitVal)
  - Split the list such that:
    - values[first] to values[splitPoint-1] <= splitVal
    - values[splitPoint] = splitVal
    - values[splitPoint+1] to values[last] > splitVal
  - Quick sort the left half (the lower half)
  - Quick sort the right half (the higher half)

Recursive Quick Sort

- Problem: Sort an array-based list of values
- Size: values[first] to values[last]
- Base Case: If there are fewer than 2 items in values[first] to values[last], do nothing
- General Case: Split the array and quick sort the two halves

Quick Sort Code

```
template<class ItemType>
void QuickSort(ItemType values[], int first, int last)
{
  if (first < last)
  {
    int splitPoint;
    Split(values, first, last, splitPoint);
    // values[first]..values[splitPoint-1] <= splitVal
    // values[splitPoint] = splitVal
    // values[splitPoint+1]..values[last] > splitVal
    QuickSort(values, first, splitPoint-1);
    QuickSort(values, splitPoint+1, last);
  }
}
```

Quick Sort: Three Questions

- Base Case: Yes, when there’s at most one element left (first >= last) the function does nothing
- Smaller-Caller: Yes, Split divides the array into smaller segments, and those segments are then sorted
- General Case: Yes, assuming Split works, eventually the base case will be reached

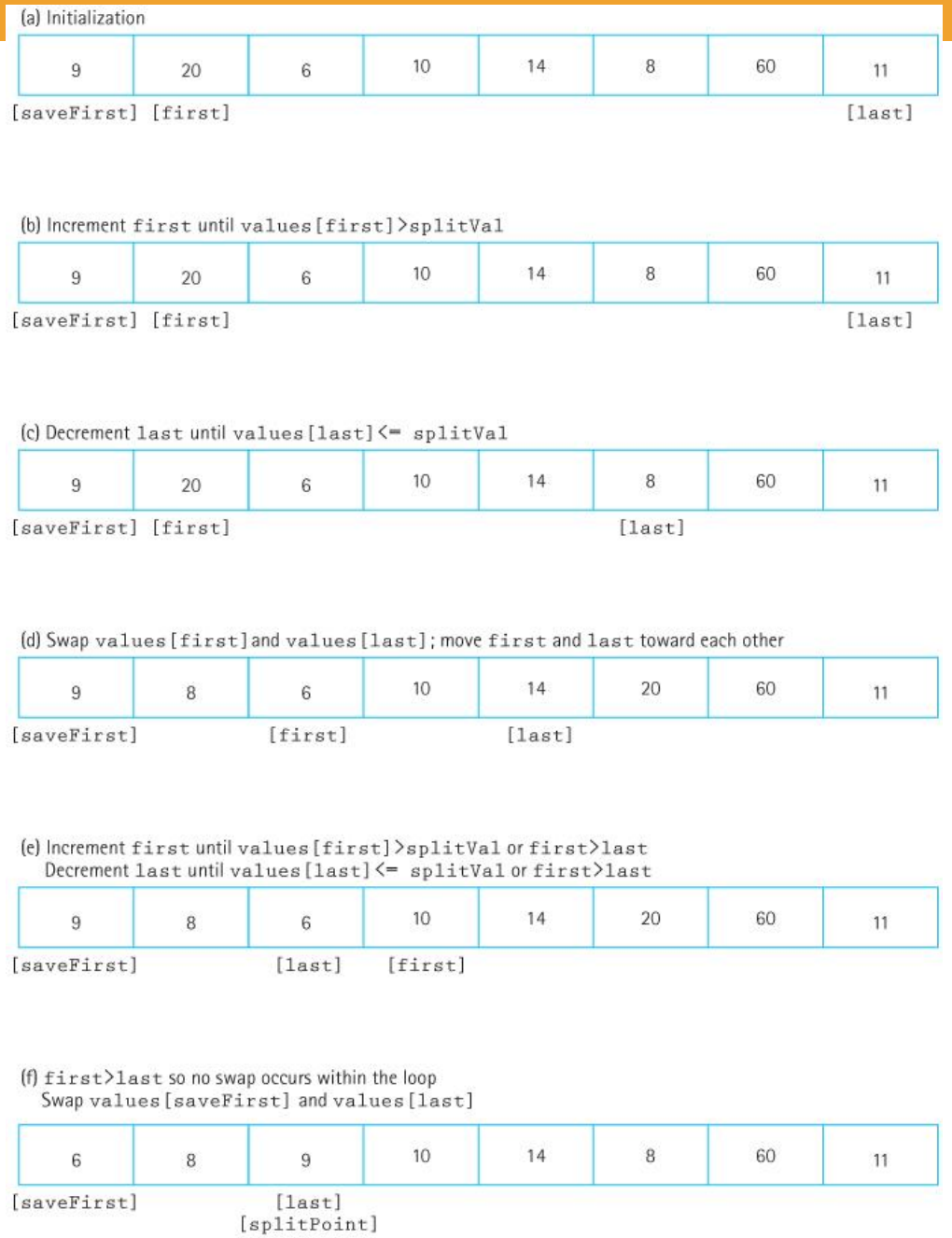
Quick Sort: Splitting

- A simple splitting algorithm walks through the array and swaps values that are out of place
- This is accomplished by using two indices, first and last, that start at front and back of the list
- Remember, we aren’t sorting, just making sure the values are on the correct side of the split
- This algorithm is correct, but may be slow

Quick Sort Splitting Algorithm

- Choose the split value.
- Advance first until it reaches a value that is greater than the split value.
- Move last backward through the list until it reaches a value that is less than the split value.
- Swap these two values and advance the two indices one space.
- Repeat until first and last meet.

Figure 7.7 Function split (a) Initialization (b) Increment first until values[first]>splitVal (c) Decrement last until values[last]<= splitVal (d) Swap values[first] and values[last]; move first and last toward each other (e) Increment first until values[first]>splitVal or first>last Decrement last until values[last]<=splitVal or first>last (f) first>last so no swap occurs within the loop Swap values[saveFirst] and values[last]



Choosing a Split Value

- The first value in the list is straightforward and typically works well
- Sometimes the list is already partially sorted, in which case choosing the middle value of the list is a better choice
- Different splitting strategies may benefit from different split values

Debugging Recursive Routines

- Recursive methods can have confusing and unique errors
- For example, it may recurse forever, causing the run-time stack to run out of space
  - This can also happen just from a method needing to recurse more times than the system can handle
- The Three-Question method is a helpful tool for thinking about and verifying recursive routines

Removing Recursion

- Sometimes recursion isn’t the best method
  - The language doesn’t support it
  - It uses too much space or time
  - An iterative solution is simpler (unusual but not impossible)
- Two techniques can be used to turn a recursive algorithm into a non-recursive one: iteration and stacking

Iteration

- **Tail Recursion:** When the recursive call is the last statement to be executed in a function
- When the recursive call returns, the calling function immediately returns the value
- This can be easily turned into a loop by using the base case conditions as loop conditions
- Some compilers automatically optimize tail recursion in this way

Stacking

- Recursive methods that aren’t tail recursive perform an action after receiving the result of the recursive calls
  - e.g., RevPrint prints the item after the rest of the list has been printed
- This can be emulated by using a stack as intermediate storage for results
  - e.g., Push each list value to a stack, then pop and print them until the stack is empty

Deciding When to Use Recursion

- **Efficiency:** Generally, recursive solutions need more computation time, memory, and overhead
- Each recursive call copies value parameters to the stack, for example
- Some recursive algorithms are inherently computationally inefficient, such as the Combinations algorithm earlier in this chapter

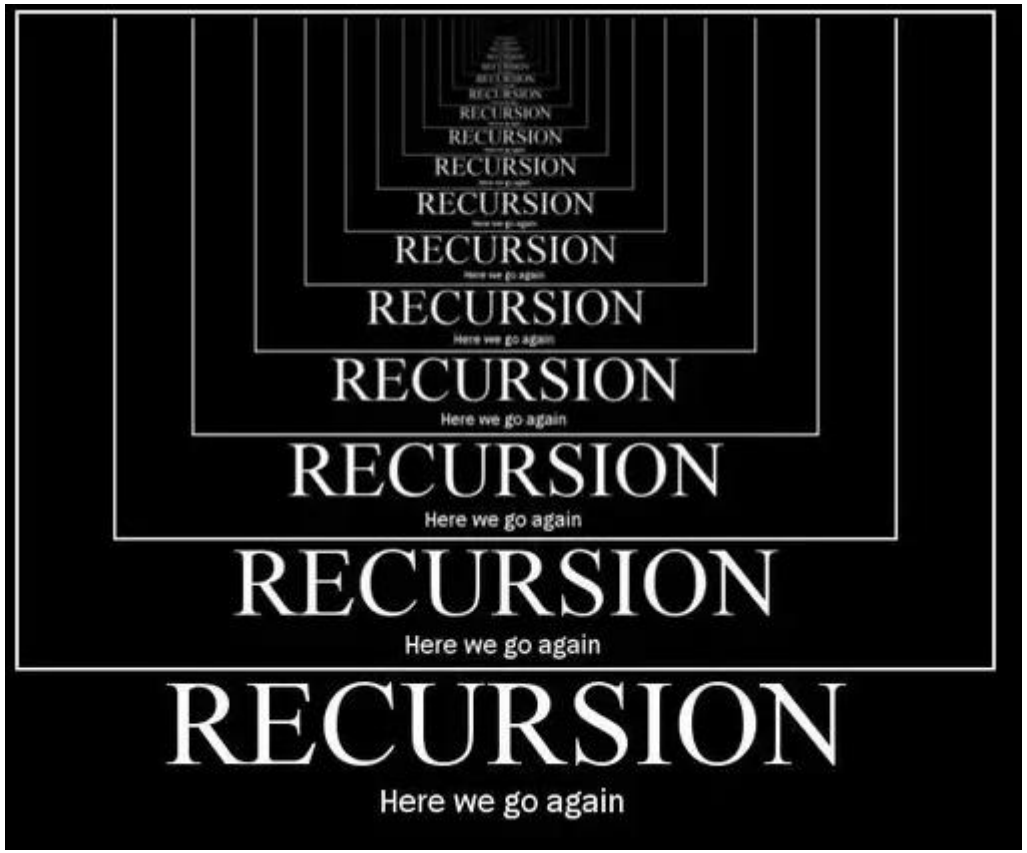
Deciding When to Use Recursion (cont.)

- **Clarity:** For many problems, the recursive solution is simpler and easier to write
  - Compare the two versions of RevPrint, for example
- Programmer efficiency should not be ignored
- Sometimes a tradeoff between development time and execution time is well worth it, especially in a team setting

When to Use Recursion

- The depth of recursion is relatively shallow: O(log*N*) is great, and even O(*N*) can be fine.
- The recursive version does about as much work as the non-recursive version.
  - e.g., Both versions of Binary Search are O(log*N*), while Combinations is O(2*N*) recursively vs. O(*N*) iteratively.
- The recursive version is shorter and simpler.

The End!


RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again