DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 5: Recursion

Objectives

Looking ahead – in this chapter, we'll consider

- Recursive Definitions
- Function Calls and Recursive Implementation
- Anatomy of a Recursive Call
- Tail Recursion
- Nontail Recursion
- Indirect Recursion
- Nested Recursion
- Excessive Recursion
- Backtracking

Recursive Definitions

- It is a basic rule in defining new ideas that they not be defined circularly
- However, it turns out that many programming constructs are defined in terms of themselves
- Fortunately, the formal basis for these definitions is such that no violations of the rules occurs
- These definitions are called ***recursive definitions*** and are often used to define infinite sets
- This is because an exhaustive enumeration of such a set is impossible, so some others means to define it is needed

Recursive Definitions (continued)

- There are two parts to a recursive definition
  - The ***anchor*** or ***ground case*** (also sometimes called the ***base case***) which establishes the basis for all the other elements of the set
  - The ***inductive clause*** which establishes rules for the creation of new elements in the set
- Using this, we can define the set of natural numbers as follows:

  1. $0 \in \mathbf{N}$ (anchor)
  2. if $n \in \mathbf{N}$, then $(n + 1) \in \mathbf{N}$ (inductive clause)
  3. there are no other objects in the set **N**

- There may be other definitions; an alternative to the previous definition is shown on page 170

Recursive Definitions (continued)

- We can use recursive definitions in two ways:
  - To define new elements in the set in question
  - To demonstrate that a particular item belongs in a set
- Generally, the second use is demonstrated by repeated application of the inductive clause until the problem is reduced to the base case
- This is often the case when we want to define functions and sequences of numbers
- However this can have undesirable consequences
- For example, to determine 3! (3 factorial) using a recursive definition, we have to work back to 0!

Recursive Definitions (continued)

- This results from the recursive definition of the factorial function:

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

- So $3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 \cdot 0! = 3 \cdot 2 \cdot 1 \cdot 1 = 6$
- This is cumbersome and computationally inefficient
- It would be helpful to find a formula that is equivalent to the recursive one without referring to previous values
- For factorials, we can use $n! = \prod_{i=1}^{n} i$
- In general, however, this is frequently non-trivial and often quite difficult to achieve

Recursive Definitions (continued)

- These discussions and examples have been on a theoretical basis
- From the standpoint of computer science, recursion occurs frequently in language definitions as well as programming
- Fortunately, the translation from specification to code is fairly straightforward; consider a factorial function in C++:

```
unsigned int factorial (unsigned int n){
   if (n == 0)
      return 1;
   else return n * factorial (n - 1);
}
```

Recursive Definitions (continued)

- Although the code is simple, the underlying ideas supporting its operation are quite involved
- Fortunately, most modern programming languages incorporate mechanisms to support the use of recursion, making it transparent to the user
- Typically, recursion is supported through use of the runtime stack
- So to get a clearer understanding of recursion, we will look at how function calls are processed

Anatomy of a Recursive Call

- To gain further insight into the behavior of recursion, let's dissect a recursive function and analyze its behavior
- The function we will look at is defined in the text, and can be used to raise a number *x* to a non-negative integer power *n*:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

- We can also represent this function using C++ code, shown on the next slide

Anatomy of a Recursive Call (continued)

```
/* 102 */  double power (double x, unsigned int n) {
/* 103 */   if (n == 0)
/* 104 */      return 1.0;
            else
/* 105 */      return x * power(x,n-1);
          }
```

- Using the definition, the calculation of x^4 would be calculated as follows: $x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0))) = x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x)) = x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x$
- Notice how repeated application of the inductive step leads to the anchor

Anatomy of a Recursive Call (continued)

- This produces the result of x^0, which is 1, and returns this value to the previous call
- That call, which had been suspended, then resumes to calculate $x \cdot 1$, producing x
- Each succeeding return then takes the previous result and uses it in turn to produce the final result
- The sequence of recursive calls and returns looks like:

call 1	$x^4 = x \cdot x^3$	$= x \cdot x \cdot x \cdot x$
call 2	$x^3 = x \cdot x^2$	$= x \cdot x \cdot x$
call 3	$x^2 = x \cdot x^1$	$= x \cdot x$
call 4	$x^1 = x \cdot x^0$	$= x \cdot 1$
call 5	$x^0 = 1$	

Anatomy of a Recursive Call (continued)

- Now, the sequence of calls is kept track of on the runtime stack, which stores the return address of the function call
- This is used to remember where to resume execution after the function has completed
- The function `power()` in the earlier slide is called by the following statement:

```
/* 136 */    y = power(5.6,2);
```

- The sequence of calls and returns generated by this call, along with the address of the calling statement, the return address, and the arguments, are shown in Figure 5.2

Recursion - rules

Rule 1: Provide exit from recursion. (focus on base cases – some times, more than one base case is needed.)

Rule 2: Make sure that the successive recursive calls progress towards the base case.

Rule 3: Assume that recursive calls work.

Rule 4: Compound interest rule: Avoid redundant calls.

Recursion – simple examples

1) Compute n!

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

2) Compute $1 + 2 + \ldots + n$

$f(n) = n + f(n - 1)$

3) Compute the n-th Fibonacci number

$f(n) = f(n - 1) + f(n - 2)$

A faster way to compute x^n

Given x and n, we want to compute x^n.

Obvious iterative solution:

```
int exp_it(int x, int n) {
    int temp = x;
    for (int j= 1; j < n; ++j)
     temp*= x;
    return temp;
  }
```

The number of multiplications performed is n – 1.

We will see that a recursive algorithm provides a much faster solution.

Faster algorithm is crucial for RSA encryption algorithm.

Idea behind the algorithm

Rule of exponents:

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

- base case n = 1 → return x
- even n → call exp(x*x , n/2) and return the result.
- odd n → call exp(x*x, (n – 1)/2) and multiply the result of call by x and return the product.

In this example, recursion makes algorithm faster

Remove negative items from the list

Example:

List: -3, 4, 5, -2, 11 becomes 4, 5, 11

We will write this one recursively.

Remove negative items from the list

Example:

List: -3, 4, 5, -2, 11 becomes 4, 5, 11

We will write this one recursively.

```
void remove_negative() {
// removes all the negative items from a list
// Example input: -4 5 6 -2 8; output: 5 6 8
  if (head == NULL) return;
  else if (head->key >= 0) {
        List nList = List(head->next);
        nList.remove_negative();
         head->next = nList.head;
     }
  else {
         List nList = List(head->next);
        nList.remove_negative();
        head = nList.head;
     }
  }
```

Excessive Recursion

- Recursive algorithms tend to exhibit simplicity in their implementation and are typically easy to read and follow
- However, this straightforwardness does have some drawbacks
- Generally, as the number of function calls increases, a program suffers from some performance decrease
- Also, the amount of stack space required increases dramatically with the amount of recursion that occurs
- This can lead to program crashes if the stack runs out of memory
- More frequently, though, is the increased execution time leading to poor program performance

Excessive Recursion (continued)

- As an example of this, consider the Fibonacci numbers
- They are first mentioned in connection with Sanskrit poetry as far back as 200 BCE
- Leonardo Pisano Bigollo (also known as Fibonacci), introduced them to the western world in his book *Liber Abaci* in 1202 CE
- The first few terms of the sequence are 0, 1, 1, 2, 3, 5, 8, … and can be generated using the function:

$$Fib(n) = \begin{cases} n & \text{if } n < 2 \\ Fib(n-2) + Fib(n-1) & \text{otherwise} \end{cases}$$

Excessive Recursion (continued)

- This tells us that that any Fibonacci number after the first two (0 and 1) is defined as the sum of the two previous numbers
- However, as we move further on in the sequence, the amount of calculation necessary to generate successive terms becomes excessive
- This is because every calculation ultimately has to rely on the base case for computing the values, since no intermediate values are remembered
- The following algorithm implements this definition; again, notice the simplicity of the code that belies the underlying inefficiency

Excessive Recursion (continued)

```
unsigned long Fib(unsigned long n) {
    if (n < 2)
        return n;
// else
        return Fib(n-2) + Fib(n-1);
}
```

- If we use this to compute `Fib(6)` (which is 8), the algorithm starts by calculating `Fib(4)` + `Fib(5)`
- The algorithm then needs to calculate `Fib(4)` = `Fib(2)` + `Fib(3)`, and finally the first term of that is `Fib(2)` = `Fib(0)` + `Fib(1)` = 0 + 1 = 1

Excessive Recursion (continued)

- The entire process can be represented using a tree to show the calculations:

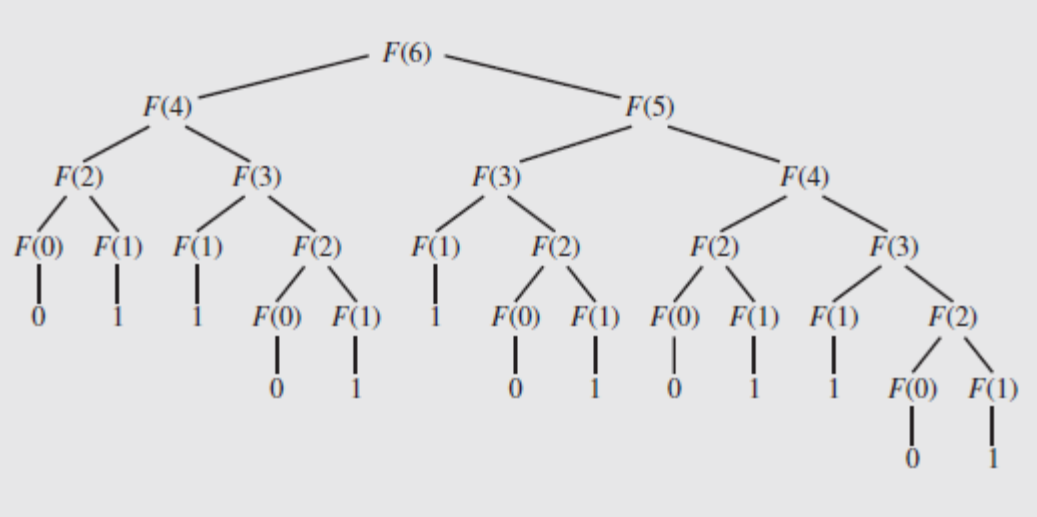

Fig. 5.8 The tree of calls for Fib(6).

- Counting the branches, it takes 25 calls to `Fib()` to calculate `Fib(6)`

Excessive Recursion (continued)

- The total number of additions required to calculate the n^{th} number can be shown to be Fib(n + 1) – 1
- With two calls per addition, and the first call taken into account, the total number of calls is 2 · Fib(n + 1) – 1
- Values of this are shown in the following table:

n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

Fig. 5.9 Number of addition operations and number of recursive calls to calculate Fibonacci numbers.

Backtracking

- ***Backtracking*** is an approach to problem solving that uses a systematic search among possible pathways to a solution
- As each path is examined, if it is determined the pathway isn't viable, it is discarded and the algorithm returns to the prior branch so that a different path can be explored
- Thus, the algorithm must be able to return to the previous position, and ensure that all pathways are examined
- Backtracking is used in a number of applications, including artificial intelligence, compiling, and optimization problems
- One classic application of this technique is known as ***The Eight Queens Problem***

Backtracking (continued)

- In this problem, we try to place eight queens on a chessboard in such a way that no two queens attack each other

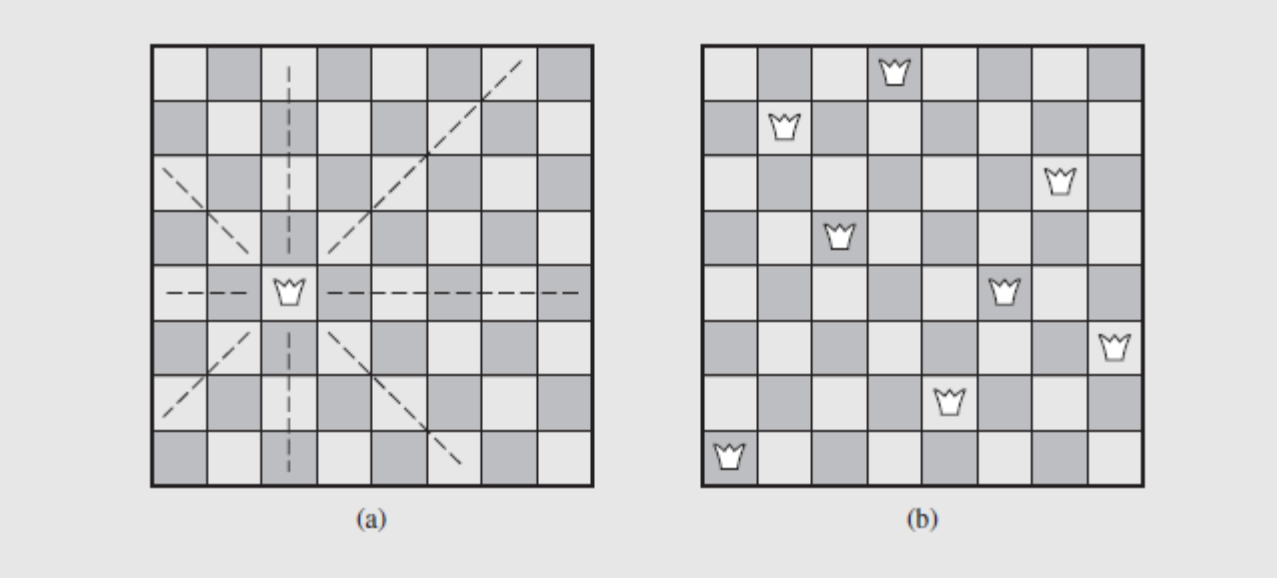

Fig. 5.11 The eight queens problem

Backtracking (continued)

- The approach to solving this is to place one queen at a time, trying to make sure that the queens do not check each other
- If at any point a queen cannot be successfully placed, the algorithm backtracks to the placement of the previous queen
- This is then moved and the next queen is tried again
- If no successful arrangement is found, the algorithm backtracks further, adjusting the previous queen's predecessor, etc.
- A pseudocode representation of the backtracking algorithm is shown in the next slide; the process is described in detail on pages 192 – 197, along with a C++ implementation

Backtracking (continued)

```
putQueen(row)
    for every position col on the same row
       if position col is available
          place the next queen in position col;
          if (row < 8)
             putQueen(row+1);
          else success;
          remove the queen from position col;
```

- This algorithm will find all solutions, although some are symmetrical

Concluding Remarks

- The foregoing discussion has provided us with some insight into the use of recursion as a programming tool
- While there are no specific rules that require we use or avoid recursion in any particular situation, we can develop some general guidelines
- For example, recursion is generally less efficient than the iterative equivalent
- However, if the difference in execution times is fairly small, other factors such as clarity, simplicity, and readability may be taken into account
- Recursion often is more faithful to the algorithm's logic

Concluding Remarks (continued)

- The task of converting recursive algorithms into their iterative equivalents can often be difficult to perform
- As we saw with nontail recursion, we frequently have to explicitly implement stack handling to handle the runtime stack processing incorporated into the recursive form
- Again, this may require analysis and judgment by the programmer to determine the best course of action
- The text suggests a couple of situations where iterative versions are preferable to recursive ones
- First, real-time systems, with their stringent time requirements, benefit by the faster response of iterative code

Concluding Remarks (continued)

- Another situation occurs in programs that are repeatedly executed, such as compilers
- However, even these cases may be changed if the hardware or operating environment of the algorithm supports processing that speeds the recursive algorithm (consider a hardware supported stack)
- Sometimes the best way to decide which version to use relies simply on coding both forms and testing them
- This is especially true in cases involving tail recursion, where the recursive version may be faster, and with nontail recursion where use a stack cannot be eliminated

Concluding Remarks (continued)

- One place where recursion must be examined carefully is when excessive, repeated calculations occur to obtain results
- The discussion of the Fibonacci sequence illustrated this concern
- Often, drawing a call tree such as figure 5.8 can be helpful
- Trees with a large number of levels can threaten stack overflow problems
- On the other hand, shallow, “bushy” trees may indicate a suitable recursive candidate, provided the number of repetitions is reasonable