Chapter 2: Recursion

- Properties of recursive solutions
- Examples
- Efficiency
Recursive Solutions

• Recursion is a programming pattern
  – function calls itself (on certain conditions)
• Solutions to some computing problems lend themselves naturally to recursion
  – solution is clearer
• Is a powerful problem-solving technique
  – breaks problem into smaller identical problems
  – alternative to iteration, which involves loops
Recursive Solutions

• Facts about a recursive solution
  – a recursive function calls itself
  – each recursive call solves an identical, but smaller, problem
  – the solution to at least one smaller problem— the base case—is known
  – eventually, one of the smaller problems must be the base case; reaching the base case enables the recursive calls to stop!
Recursive Solutions

• Four questions for constructing recursive solutions
  – How can you define the problem in terms of a smaller problem of the same type?
  – How does each recursive call diminish the size of the problem?
  – What instance of the problem can serve as the base case?
  – As the problem size diminishes, will you reach this base case?
Recursion Details

• Each function call (recursive or otherwise) pushes a new record on the *runtime* stack
  – contains arguments, locals, etc.
  – maintains function state
  – record popped on function return
  – introduces time and space overhead

• Box trace is visualize recursive call stack
Box Trace

• A systematic way to trace the actions of a recursive function
• Each box roughly corresponds to an activation record
• Contains function’s local environment at time of and as a result of the call to the function
A1: A Recursive Valued Function: The Factorial of n

• Problem -- Compute factorial of an integer n

• An iterative definition of factorial(n)

\[ \text{factorial}(n) = n \times (n - 1) \times (n - 2) \times \ldots \times 1 \]

for any integer \( n > 0 \)

\[ \text{factorial}(0) = 1 \]

• A recursive definition of \( \text{factorial}(n) \)

\[ \text{factorial}(n) = 1 \quad \text{if} \ n = 0 \]

\[ = n \times \text{factorial}(n-1) \quad \text{if} \ n > 0 \]

see C2-factorial.cpp
Box Trace

• A function’s local environment includes:
  – The function’s local variables
  – A copy of the actual value arguments
  – A return address in the calling routine
  – The value of the function itself

\[
\begin{align*}
n &= 3 \\
A: \text{fact}(n-1) &= ? \\
\text{return} &= ?
\end{align*}
\]
A2: A Recursive void Function: Writing a String Backward

• Problem
  – Given a string of characters, write it in reverse order

• Recursive solution
  – Each recursive step of the solution diminishes by 1 the length of the string to be written backward
  – Base case: write the empty string backward
A2: A Recursive void Function: Writing a String Backward

- Execution of writeBackward can be traced using the box trace
- Temporary cout statements can be used to debug a recursive method
A3: Fibonacci Sequence – Multiplying Rabbits

• Problem statement about rabbit growth
  – rabbits never die
  – a rabbit reaches sexual maturity exactly two months after birth, that is, at the beginning of its third month of life
  – rabbits are always born in male-female pairs. At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair
  – How many pairs of rabbits are alive in month \( n \)?
A3: Fibonacci Sequence – Multiplying Rabbits

• Recurrence relation
  \[ \text{rabbit}(n) = \text{rabbit}(n - 1) + \text{rabbit}(n - 2) \]

• Base cases
  \[ \text{rabbit}(2) = \text{rabbit}(1) = 1 \]

• Recursive definition
  \[ \begin{align*}
      \text{rabbit}(n) &= 1 & \text{if } n \text{ is 1 or 2} \\
      &= \text{rabbit}(n - 1) + \text{rabbit}(n - 2) & \text{if } n > 2
  \end{align*} \]

• Fibonacci sequence
  – The series of numbers \( \text{rabbit}(1), \text{rabbit}(2), \text{rabbit}(3), \) and so on; that is, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
A4: Organizing a Parade

• Problem statement
  – How many ways can you organize a parade of length $n$?
  – The parade will consist of bands and floats in a single line
  – One band cannot be placed immediately after another
A4: Organizing a Parade

• Let:
  – \( P(n) \) be the number of ways to organize a parade of length \( n \)
  – \( F(n) \) be the number of parades of length \( n \) that end with a float
  – \( B(n) \) be the number of parades of length \( n \) that end with a band

• Then
  – \( P(n) = F(n) + B(n) \)
A4: Organizing a Parade

• Number of acceptable parades of length $n$ that end with a float
  $- F(n) = P(n - 1)$

• Number of acceptable parades of length $n$ that end with a band
  $- B(n) = F(n - 1)$

• Number of acceptable parades of length $n$
  $- P(n) = P(n - 1) + P(n - 2)$
A4: Organizing a Parade

• Base cases
  
  \[ P(1) = 2 \] (The parades of length 1 are \textit{float} and \textit{band}.)

  \[ P(2) = 3 \] (The parades of length 2 are \textit{float-float}, \textit{band-float}, and \textit{float-band}.)

• Solution
  
  \[ P(1) = 2 \]

  \[ P(2) = 3 \]

  \[ P(n) = P(n - 1) + P(n - 2) \quad \text{for} \quad n > 2 \]
A5: Choosing k out of n Things

• Problem statement
  – How many different choices are possible for exploring k planets out of n planets in a system?
A5: Choosing k out of n Things

• Let $c(n, k)$ be the number of groups of $k$ planets chosen from $n$

• In terms of Planet X:
  – $c(n, k) = (\text{the number of groups of } k \text{ planets that include Planet } X) + (\text{the number of groups of } k \text{ planets that do not include Planet } X)$

• Num. of ways to choose $k$ of $n$ things is the sum
  – the number of ways to choose $k - 1$ out of $n - 1$ things
  – the number of ways to choose $k$ out of $n - 1$ things
  – $c(n, k) = c(n - 1, k - 1) + c(n - 1, k)$
A5: Choosing k out of n Things

• Base cases
  – there is one group of everything : \( c(k, k) = 1 \)
  – there is one group of nothing : \( c(n, 0) = 1 \)
  – Although k cannot exceed n here, we want our solution to be general
    \( c(n, k) = 0 \) if \( k > n \)

• Recursive solution
  \[
  c(n, k) = \begin{cases} 
  1 & \text{if } k = 0 \\
  1 & \text{if } k = n \\
  0 & \text{if } k > n \\
  c(n - 1, k - 1) + c(n - 1, k) & \text{if } 0 < k < n 
  \end{cases}
  \]
  see C2-kOfN.cpp
A5: Choosing k out of n Things

Figure 2-12 The recursive calls that \( c(4, 2) \) generates
A6: Finding Largest Item in an Array

• A recursive solution – `maxArray()`

  if (anArray has only one item)
  
  `maxArray(anArray)` is the item in anArray

  else if (anArray has more than one item)
  
  `maxArray(anArray)` is

  `MAX(maxArray(left half of anArray), maxArray(right half of anArray))`
A7: Binary Search

\[
\text{binarySearch}(\text{in anArray:ArrayType, in value:ItemType})
\]

\[
\text{if (anArray is of size 1)}
\]
\[
\text{Determine if anArray’s item is equal to value}
\]

\[
\text{else } \{
\]
\[
\text{Find the midpoint of anArray}
\]
\[
\text{Determine which half of anArray contains value}
\]
\[
\text{if (value is in the first half of anArray)}
\]
\[
\text{binarySearch(first half of anArray, value)}
\]
\[
\text{else}
\]
\[
\text{binarySearch(second half of anArray, value)}
\]
\[
\}
\]
A8: Finding $k^{th}$ Smallest Item in Array

- **Recursive solution**
  - select a ‘pivot’ item in the array
  - partitioning items in array about this pivot item
  - recursively apply strategy to one of the partitions
A8: Finding kth Smallest Item in Array

\[ \text{kSmall}(k, \text{anArray}, \text{first}, \text{last}) = \begin{cases} \text{kSmall}(k, \text{anArray}, \text{first}, \text{pivotIndex} - 1) & \text{if } k < \text{pivotIndex} - \text{first} + 1 \\ p & \text{if } k = \text{pivotIndex} - \text{first} + 1 \\ \text{kSmall}(k - (\text{pivotIndex} - \text{first} + 1), \text{anArray}, \text{pivotIndex} + 1, \text{last}) & \text{if } k > \text{pivotIndex} - \text{first} + 1 \end{cases} \]
A9: Organizing Data: The Towers of Hanoi

Figure 2-19a and b (a) The initial state; (b) move $n - 1$ disks from $A$ to $C$
A9: The Towers of Hanoi

Figure 2-19c and d (c) move one disk from A to B; (d) move $n - 1$ disks from C to B.
The Towers of Hanoi

```python
code here
```
Recursion and Efficiency

• Some recursive solutions are so inefficient that they should not be used
• Factors that contribute to the inefficiency of some recursive solutions
  – overhead associated with function calls
  – inherent inefficiency of some recursive algorithms
• Do not use a recursive solution if it is inefficient and there is a clear, efficient iterative solution
Summary

• Recursion solves a problem by solving a smaller problem of the same type
• Four questions:
  – How can you define the problem in terms of a smaller problem of the same type?
  – How does each recursive call diminish the size of the problem?
  – What instance(s) of the problem can serve as the base case?
  – As the problem size diminishes, will you reach a base case?
Summary

• To construct a recursive solution, assume a recursive call’s postcondition is true if its precondition is true
• The box trace can be used to trace the actions of a recursive method
• Recursion can be used to solve problems whose iterative solutions are difficult to conceptualize
Summary

• Some recursive solutions are much less efficient than a corresponding iterative solution due to their inherently inefficient algorithms and the overhead of function calls.

• If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so.