Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Recursion

**"To understand recursion,
you must first understand recursion."**

CS2013: Programming with Data Structures

What is Recursion??

- ***recursion:*** *a method or function which calls itself, where the solution to a problem depends on solving solutions to smaller instances (subsets) of the same problem.*
  - to say something "recurses", means the same computation recurs, or occurs repeatedly, as the problem is solved
  - usually the computation is performed on smaller and smaller subsets of the problem.
- powerful technique for breaking up complex computational problems into simpler smaller sub-problems.
- some computations may be difficult or impossible to solve without recursion.
  - can lead to very elegant solutions to problems which would be difficult to solve using only a loop.

Simple Mathematical Example

- Factorial (n!) is defined as the product of all numbers from 1 to n. If n = 0, then the factorial is defined as 1.
  - 5! = 5 * 4 * 3 * 2 * 1 = 120
  - 3! = 3 * 2 * 1 = 6

- We can define n! as a more formalized recursive definition:

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n \geq 1 \end{cases}$$

Base and Recursive Cases

- The previous definition of factorial includes the typical features of most recursive algorithms.
  - The ***base case(s):***
    - One or more special cases which end the recursion.
    - Does not make any subsequent recursive calls.
    - Without a base case, the recursion would never end.
    - In the previous definition, the base case occurs when n = 0.
  - The ***recursive case(s):***
    - Also known as the ***reduction step(s).***
    - The part of a recursive function which solves a part of the problem and reduces the problem by some factor.
    - The reduced problem is then passed to another call of the recursive function which will do some more work and further reduce the problem.
    - This step should work towards converging to the base case.

Factorial in Code

- Factorial can be implemented iteratively using a loop:

```
factorial1(n):
   p = 1

   for i from n to 2:
      p *= i

   return p
```

Factorial in Code

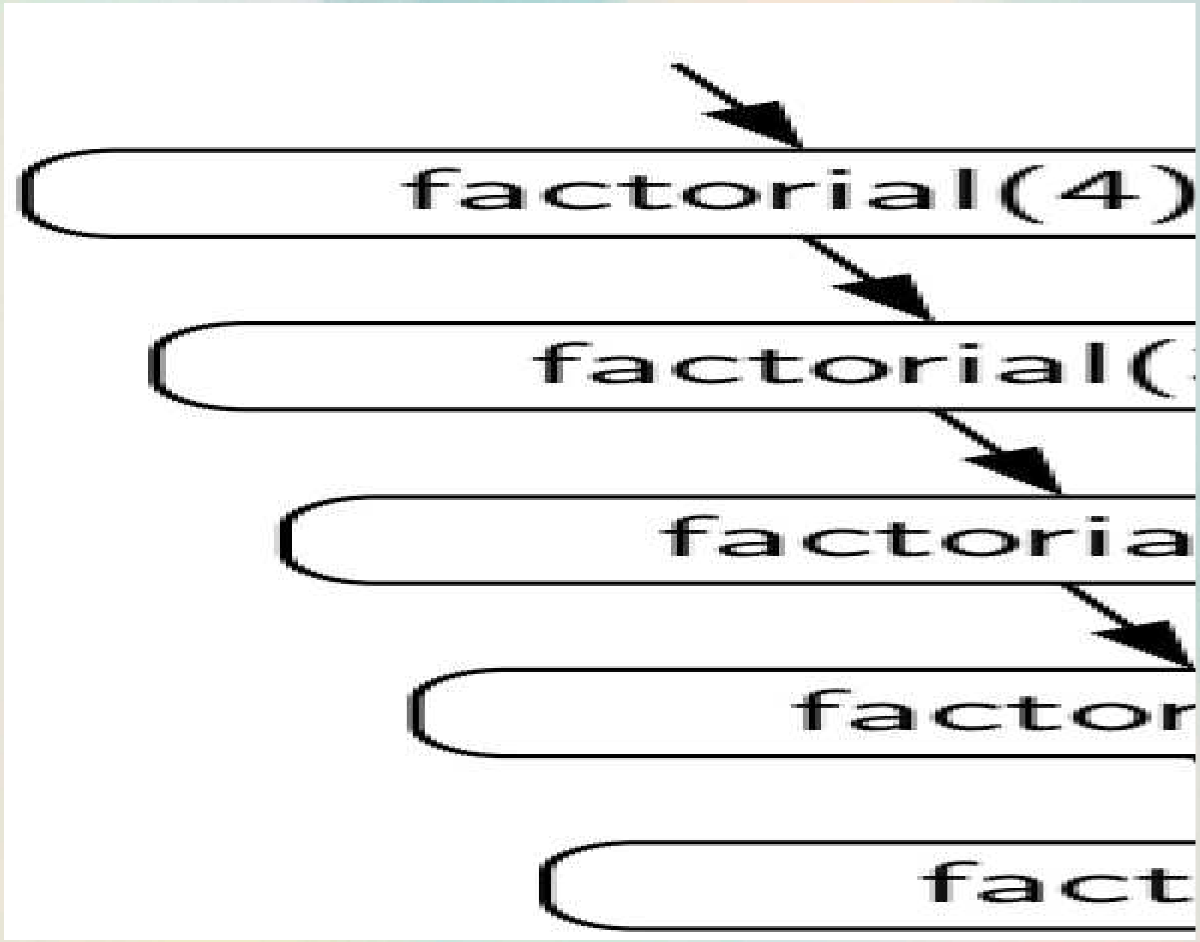
- Factorial can also be implemented using recursion:

```
factorial2(n):
   //base case
   if n = 0:
      return 1

   //recursive case
   return n * factorial2(n - 1)
```

- The base case returns 1.
- The recursive case reduces the problem by 1 giving a smaller ***sub-problem*** (n - 1)
  - The sub-problem has the same properties as the original problem
  - Call the recursive method again with the sub-problem as the argument to the method.
  - Eventually the subproblems will converge to the base case.

Factorial In Code

- The recursive factorial does not use any loops. The repetition is achieved by invoking the same method repeatedly.
- The process eventually stops, because the base case is reached.
- If you have a hard time thinking this through, sometimes it is helpful to do a ***recursive trace*** of the algorithm.
  - An example trace is shown to the right.



The Call Stack (Review)

- Recall how the ***Method Call Stack*** works:
  - When a method is called in Java, an activation record (activation frame) is created in memory.
    - These frames / records store all necessary information about the progress of that method such as local variables, parameters and the line of code currently being executed.
  - If a call to another method is nested inside of the current method, the execution of the current method is suspended and its frame will store the place where the method should resume at a later time.
  - A new frame is created for the new method invocation and the new method will execute until it is finished, or until it calls another method.
- This process is the same for all types of methods (recursive or non-recursive methods).
- Every time you call the same method, you create a brand new instance of that method in memory.

Information Processing Example

- This next example uses multiple methods for one recursive algorithm.
- Here, the general idea is that we have information being processed.
  - Receive more information if the buffer is not full.
  - Decode the information in the buffer.
  - Store the information from the buffer to a file.
  - Repeat until there is no more information.

Information Processing Example

```
receive(buffer):
   while buffer is empty:
      if more information:
         store the next character in the buffer
      else:
         exit program

   decode(buffer)

decode(buffer):
   decode information in buffer
   store(buffer)

store(buffer):
   transfer information in buffer to file
   receive(buffer)
```

Direct vs Indirect Recursion

- ***direct recursion:*** when a recursive method directly invokes itself.
  - i.e. Method A calls Method A over and over.
  - Example: Factorial

- ***indirect recursion:*** when there might be other methods involved in the recursive process.
  - i.e. Method A calls Method B, Method B calls Method A, and the cycle continues.
  - i.e. Method A invokes Method B, Method B calls Method C, Method C calls Method A, and the cycle continues
  - Example: Information Processing

Example: Drawing a Ruler

- Let's look at a more complex example, drawing an English ruler.

- Each inch of the ruler has a tick followed by a numeric label.
  - **major tick length**: the length of the tick mark that represents a whole inch.
  - **minor tick length**: ticks of various lengths placed at intervals of ½ inch ¼ inch and so on.
    - As the size of the interval decreases by half, the tick length decreases by 1.

Example: Drawing a Ruler

```
---- 0          ----- 0         --- 0
-               -               -
--              --              --
-               -               -
---             ---             --- 1
-               -               -
--              --              --
-               -               -
---- 1          ----            --- 2
-               -               -
--              --              --
-               -               -
---             ---             --- 3
-               -
--              --
-               -
---- 2          ----- 1

(a)             (b)             (c)
```

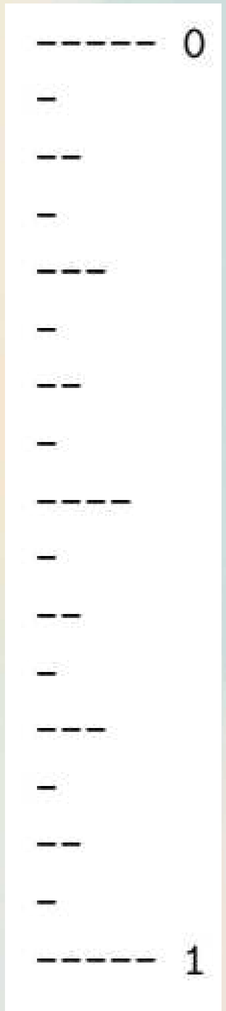
Figure 5.2: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

Example: Drawing a Ruler

- This pattern is an example of a ***fractal***, that is, a shape that has a self-recursive structure at various levels of magnification.

- Look at figure 5.2(b) and observe the following:
  - Ignore the 0 and 1 lines
  - The central tick (½ in) has a length of 4.
  - The two patterns of ticks above and below the central tick are identical
    - Each of these patterns has its own central tick with a length of 3.

- In general, an interval with a central tick length L >= 1 is composed of:
  - an interval with a central tick length L -1
  - A single tick of length L
  - An interval with a central tick length L -1

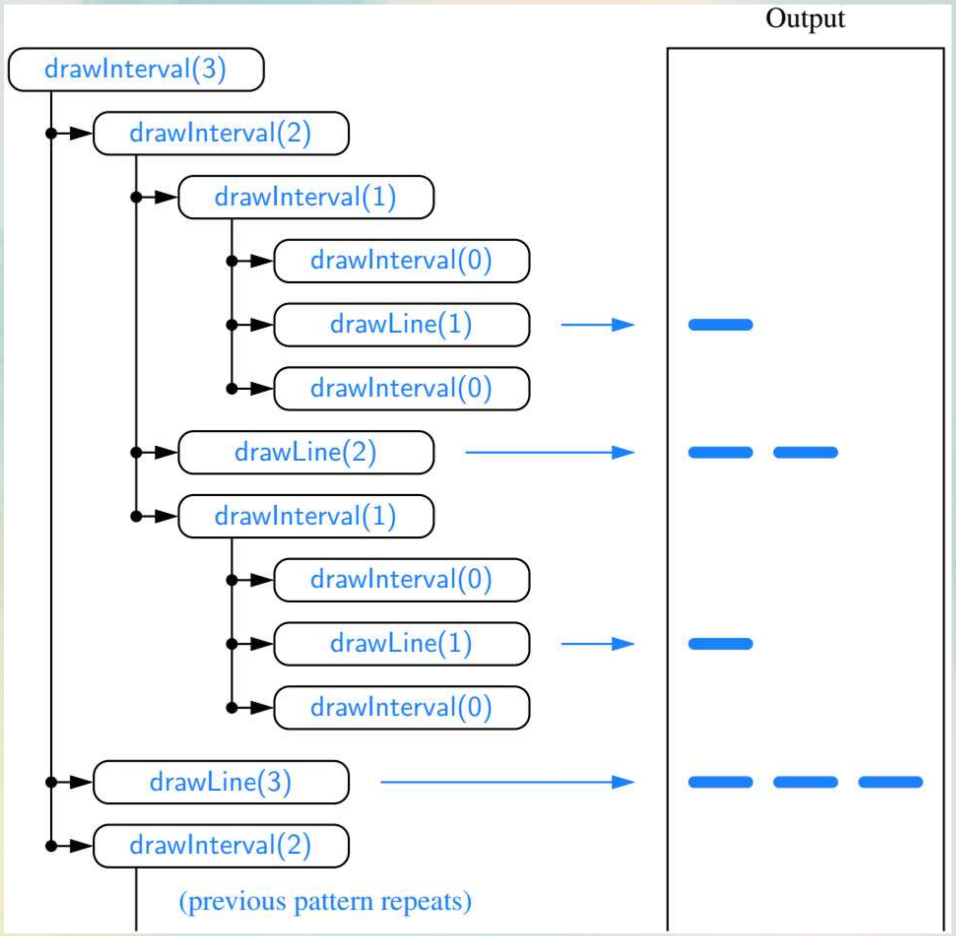


```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

(b)

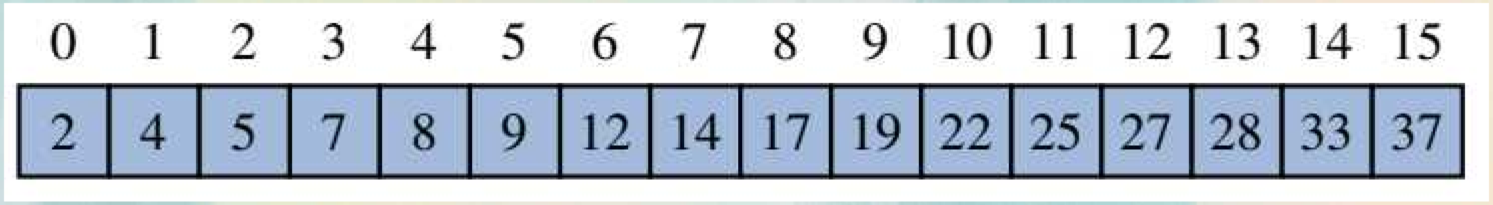
Example: Drawing a Ruler

```
drawRuler(nInches, majorLength):
   drawLine(majorLength, 0)
   for i from 1 to nInches:
      drawInterval(majorLength - 1)
      drawLine(majorLength, i)

drawInterval(centralLength):
   if centralLength >= 1:
      drawInterval(centralLength - 1)
      drawLine(centralLength)
      drawInterval(centralLength - 1)

drawLine(tickLength, tickLabel):
   for i from 0 to tickLength - 1:
      print("-")
   if tickLabel >= 0:
      print(" " + tickLabel)
   print newline

drawLine(tickLength):
   drawLine(tickLength, -1)
```



Example: Binary Search

- ***binary search***: is an efficient search algorithm for locating a particular element in a **sorted** list.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

- This algorithm starts by finding the middle index of all of the elements and then considers three cases:
  - If the middle element is equal to the item we are search for, we found our item and we are done.
  - If the item we want to find is less than the middle element, search on the first half of the list.
  - If the item we want to find is greater than the middle element, search on the second half of the list.

```
list: an array or other list type data structure
key: the item we are searching for
low: the lowest index of the range we are searching
high: the highest index of the range we are searching

binarySearch(list, key, low, high):
   if low > high:
      return false
   else
      mid = (low + high) / 2
      if (key == list[mid]):
         return true
      else if key < list[mid]:
         return binarySearch(list, key, low, mid - 1)
      else if key > list[mid]:
         return binarySearch(list, key, mid + 1, high)
```

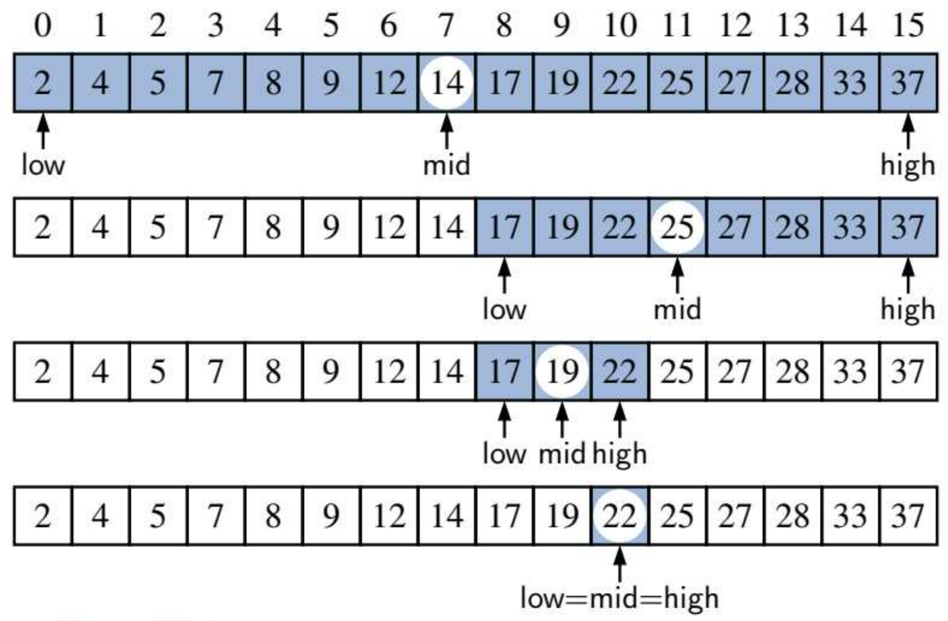
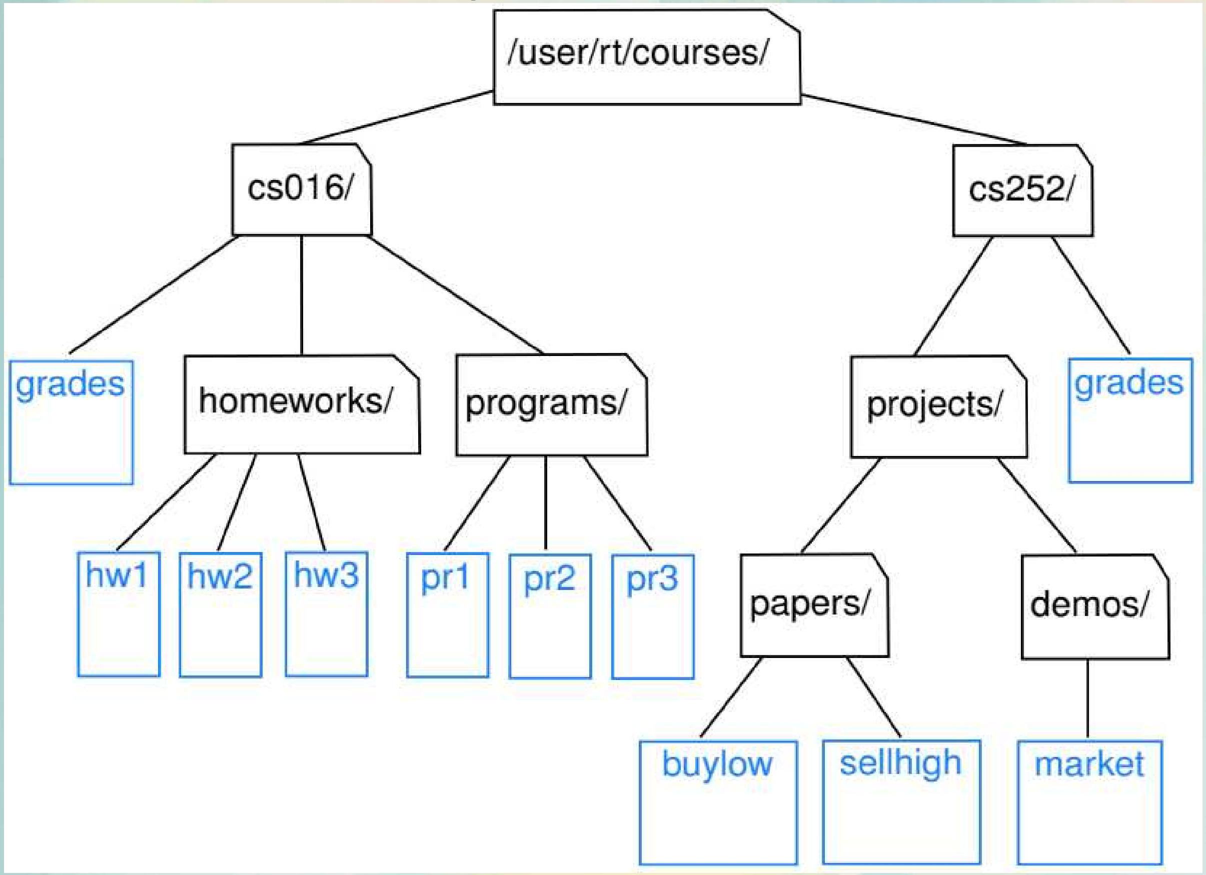


Figure 5.5: Example of a binary search for target value 22

Example: Compute Directory Size

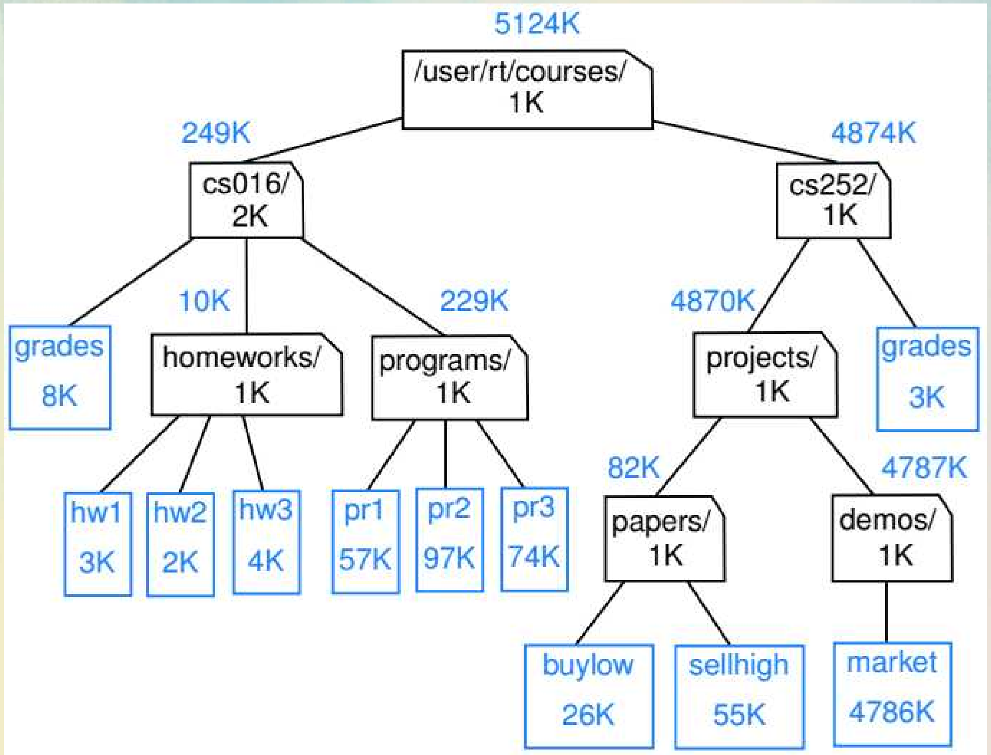
- Modern file systems define directories in a recursive way.
- Normally there is a top level directory which can contain any number of files and other directories, which also contain more files and directories and so on.
- Directory structures are inherently recursive, so a lot of the file system operations are defined recursively.



Example: Compute Directory Size

- Computing the size of a directory can be very easily implemented using recursion. The size of a directory is the sum of all the sizes of the files and directories in that directory.

```
path: the path to the file / folder

folderSize(path):
   if path is a file:
      return size of file
   else:
      for each item in folder
         total += folderSize(item)

    return total
```



Linear Recursion

Linear Recursion

- ***linear recursion***: a recursive method is designed such that each invocation of the body makes at most one new recursive call.
  - NOTE: linear here refers to the structure of the recursion trace, not the Big-O analysis of the algorithm. (we will discuss Big-O at a later time).
- The Factorial and Binary Search examples are both examples of linear recursion.
  - Factorial is obvious that it is linear.
  - Binary search does have two recursive branches, only one branch is chosen during a particular execution of the body of the method.
- An easy way to tell if a recursion is linear, is to hand trace and see if the trace appears as a single sequence of method calls.

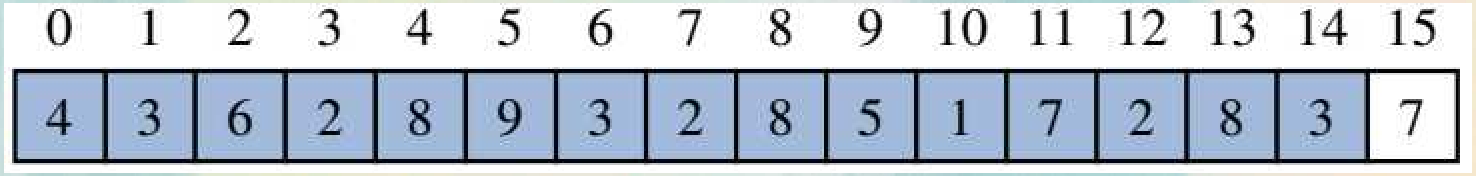
Linear Recursion: Summing an Array

- Linear recursion is useful for processing a sequence like a Java array or list.

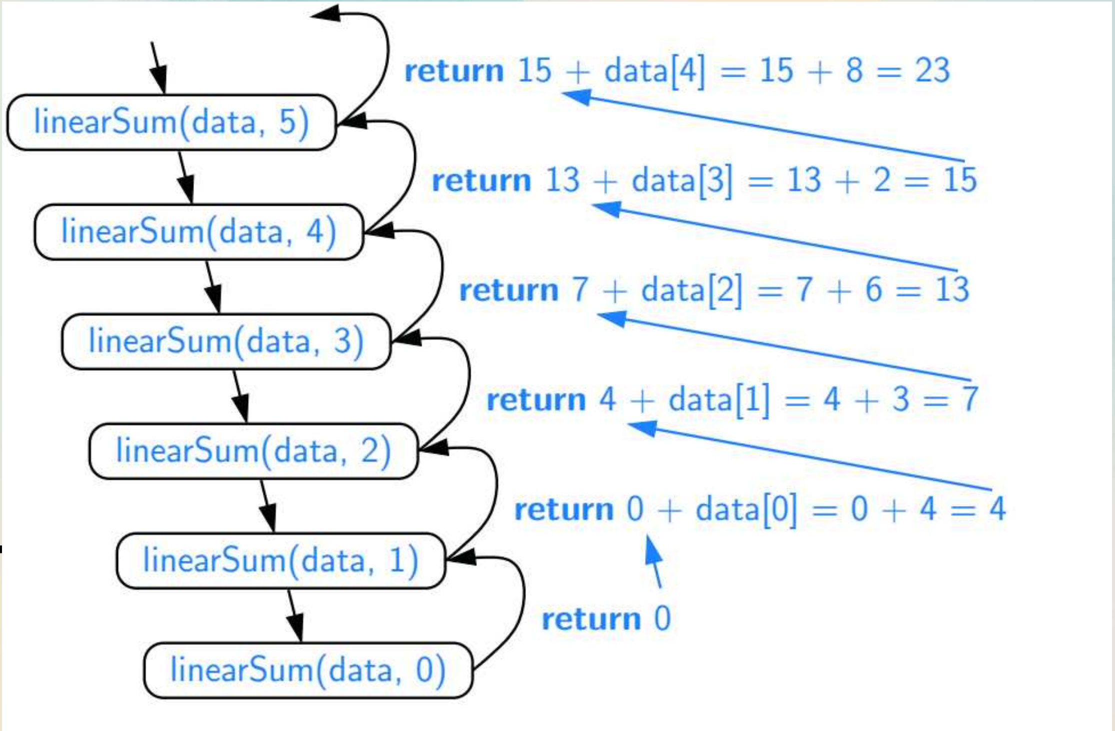
- Example: Compute the sum of an array of *n* integers.
  - NOTE: *n* here is the number of elements in the array you want to sum, NOT the index.
  - Observe that if n = 0, the sum is 0 (there are no values in the array, or we have reached the end of the array.)

Linear Recursion: Summing an Array

- Compute the sum of a sequence recursively by adding the last number to the sum of the first n-1 elements.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	6	2	8	9	3	2	8	5	1	7	2	8	3	7



return 15 + data[4] = 15 + 8 = 23
linearSum(data, 5)
return 13 + data[3] = 13 + 2 = 15
linearSum(data, 4)
return 7 + data[2] = 7 + 6 = 13
linearSum(data, 3)
return 4 + data[1] = 4 + 3 = 7
linearSum(data, 2)
return 0 + data[0] = 0 + 4 = 4
linearSum(data, 1)
return 0
linearSum(data, 0)

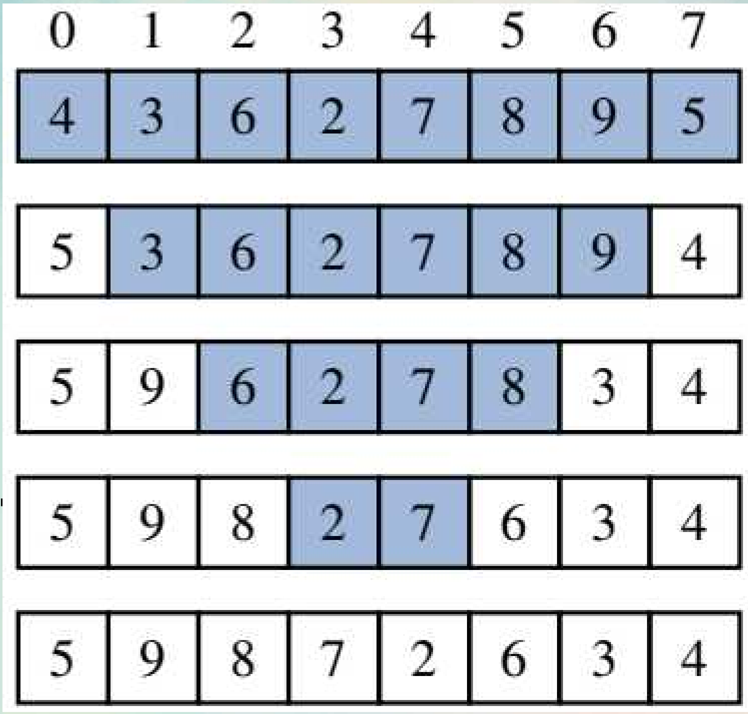
Figure 5.10: Recursion trace for an execution of linearSum(data, 5) with input parameter data = 4, 3, 6, 2, 8.

```
list: a list of the numbers
n: the number of elements to sum

linearSum(list, n):
   if n = 0:
      return 0
   else
      return linearSum(list, n-1) + data[n-1]
```

Linear Recursion: Reversing a Sequence

- Reverse the *n* elements of an array so the first element becomes the last, the second becomes the second to last and so on.
- Observation: This can be achieved by swapping the corresponding elements at both ends.

```
list: a list elements
low: an integer that is the starting index
high: an integer that is the ending index

reverseArray(list, low, high):
   if low >= high
      return
   else
      temp = data[low]
      data[low] = data[high]
      data[high] = temp
      reverseArray(list, low + 1, high - 1)
```



Linear Recursion: Computing Powers

- The ***power function*** (x^n) can be computed using a recursive algorithm.

- A simple recursive definition can be derived from the fact that $x^n = x \quad x^{n-1}$ for n>0:

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

```
x: the base of the power.
n: the exponent of the power.

power(x, n):
   if n == 0:
      return 1
   else:
      return x * power(x, n-1)
```

Binary Recursion

Binary Recursion

- ***binary recursion***: when a method always makes two recursive calls.
  - drawing the English Ruler was one example.

- Another example would be to revisit the summation of an array:
  - computing the sum of 1 or 0 values is trivial
  - with two or more values we can recursively compute the sum of the first half, the sum of the second half, and then add those results together.

Binary Recursion: Binary Sum

- The following is an implementation of the previous array summation, using binary recursion:

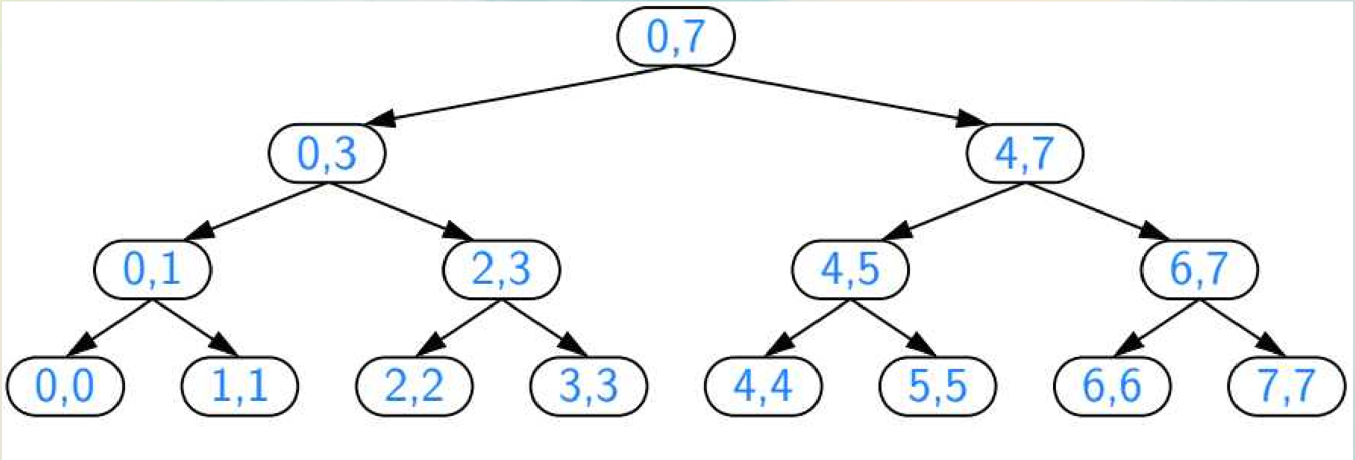


Figure 5.13: Recursion trace for the execution of binarySum(data, 0, 7).

```
list: a list of numbers to sum.
low: the start position in the list.
high: the end position in the list

binarySum(list, low, high):
   if low > high:
      return 0
   else if low == high:
      return data[low]
   else:
      mid = (low + high) / 2
      return binarySum(data, low, mid) + binarySum(data, mid+1, high)
```

Multiple Recursion

Multiple Recursion

- ***multiple recursion:*** a method may make more than two recursive calls
  - An example of this was size of a file system since the number of recursive calls during one invocation is equal to the number of entries in a given directory.

- Another example of multiple recursion is in solving combinatorial puzzles an example of which is a summation puzzle.

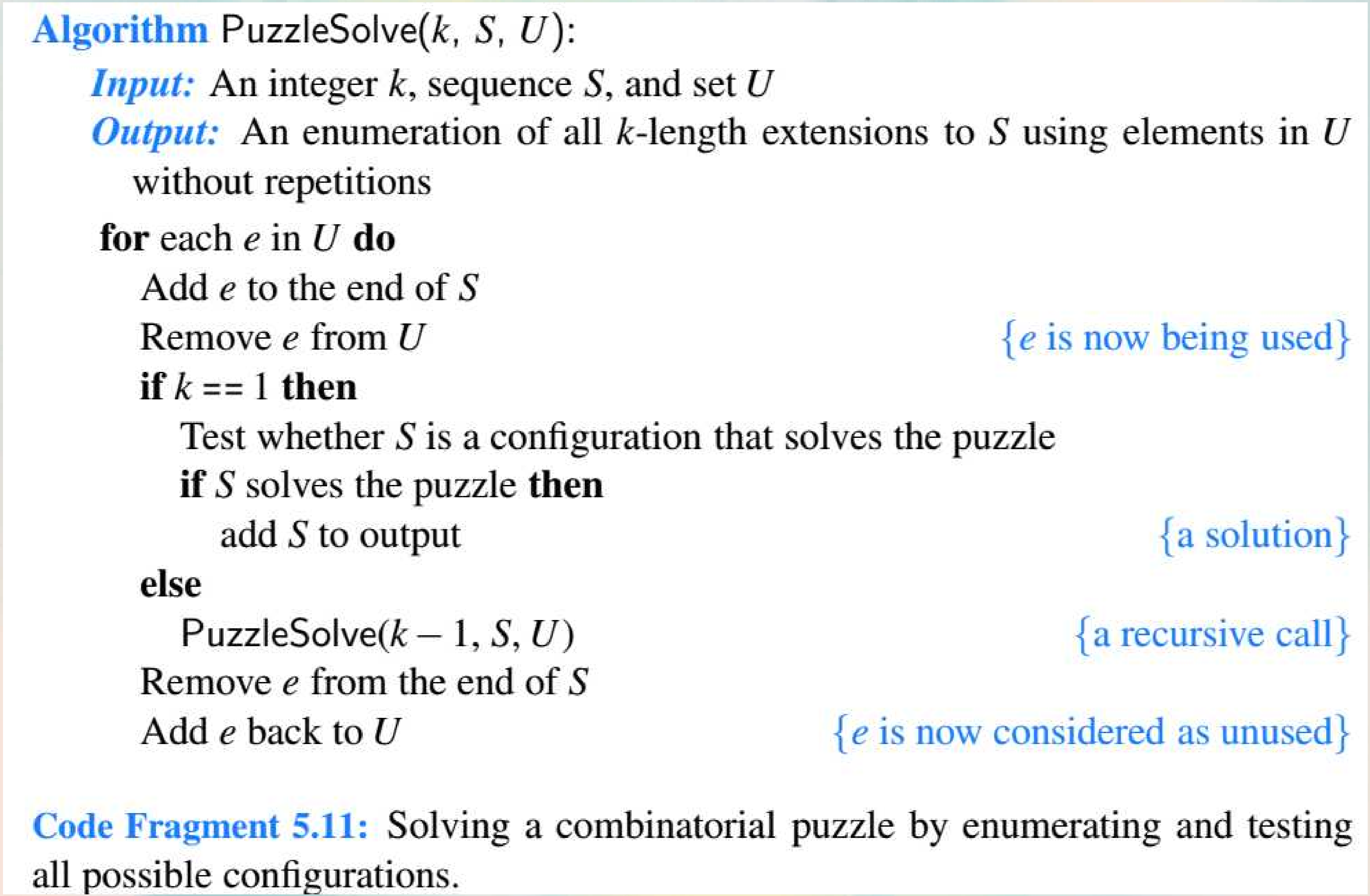
- Summation puzzles are puzzles where a digit is assigned to each letter and the sum of the resulting numbers must equal the value on the right.

– pot + pan = bib	143 + 109 = 252
– dog + cat = pig	452 + 310 = 762
– boy + girl = baby	491 + 3750 = 4241

Multiple Recursion

- The following shows a generalized way to deal with these types of combinatorial problems.
- The algorithm enumerates and tests all k-length sequences, without repetitions, chosen from a given set U.
- The algorithm builds a sequence of k elements using the following steps:
  - Recursively generate sequences of k - 1 elements.
  - Append to each sequence an element not already contained in it.
- U is a set that tracks elements not contained in the current sequence, i.e. element *e* will be in set U if e has not been added to the sequence yet.
- For summation puzzles U = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} and each position corresponds to a given letter.

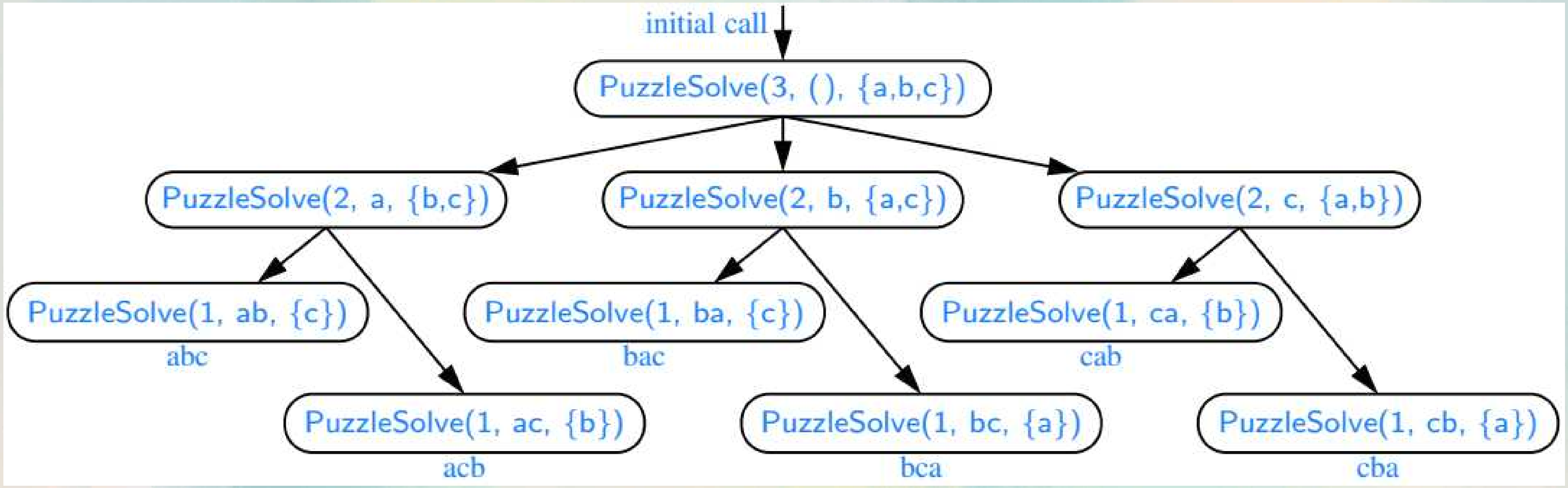
Multiple Recursion



```
Algorithm PuzzleSolve($k$, $S$, $U$):
    Input: An integer $k$, sequence $S$, and set $U$
    Output: An enumeration of all $k$-length extensions to $S$ using elements in $U$
      without repetitions
  for each $e$ in $U$ do
    Add $e$ to the end of $S$
    Remove $e$ from $U$                                   {$e$ is now being used}
    if $k == 1$ then
      Test whether $S$ is a configuration that solves the puzzle
      if $S$ solves the puzzle then
        add $S$ to output                                 {a solution}
    else
      PuzzleSolve($k-1$, $S$, $U$)                        {a recursive call}
    Remove $e$ from the end of $S$
    Add $e$ back to $U$                                   {$e$ is now considered as unused}
```

Code Fragment 5.11: Solving a combinatorial puzzle by enumerating and testing all possible configurations.

Multiple Recursion



Multiple Recursion: String Permutations

- A permutation is the rearrangement of the letters in the string.
- Example: the string "eat" has six permutations (including the original string)

"eat"	"ate"
"eta"	"tea"
"aet"	"tae"

- Now, how could we solve this recursively?

Multiple Recursion: String Permutations

- First generate all permutations starting with the letter 'e', then those that start with 'a', then those that start with 't'

- If we start with 'e' as the first letter, we need to generate all permutation of the substring "at"
  - this is also done with recursion and the permutations are "at" and "ta"
  - for each permutation of that substring add the letter 'e' at the beginning

    "**e**at"
    "**e**ta"
  - We can continue this for all permutations

Multiple Recursion: String Permutations

```
prefix: an empty string initially.
str: the string you want to permutate.

permutate(prefix, str):
   if (str is empty):
      print prefix
   else:
      for all characters c in str:
         permutate(prefix + c, str - c)

         //str - c = str with the current character removed
```

Designing Recursive Algorithms

Designing Recursive Algorithms

- **Test for base cases:**
  - Always identify and test for base cases.
  - A recursive algorithm always has 1 or more base cases and these are the stopping points of the algorithm.
  - Define the cases so that each recursive call will reduce and converge to one of the base cases.
  - These cases **should not** have a recursive call in them.
- **Recursive cases:**
  - Any other case in a recursive algorithm is a recursive case and should include a recursive call.
  - Each recursive call should reduce the problem in some way.
  - You may have multiple recursive cases, or multiple recursive calls within the same case.

- Work out some examples by hand and see how you should design each subproblem.

Recursive Helper Methods

- Sometimes you will need to redefine the original problem to allow for similar looking subproblems.

- This may involve creating a recursive helper method.
  - This is a method which has a parameter list designed to accommodate any information that needs to be passed to subsequent recursive calls.
  - This method is usually private, and a public version of this method with a simpler parameter list is usually visible to the client.

Recursive Helper Methods

- The string permutation example can be rewritten with a helper method:

```
public permutate(str):
   permutate("", str)


private permutate(prefix, str):
   if (str is empty):
      print prefix
   else:
      for all characters c in str:
         permutate(prefix + c, str - c)

         //str - c = str with the current character removed
```

Recursive Helper Methods

- The binarySearch example can also be rewritten to use a recursive helper method:

```
public binarySearch(list, key):
   binarySearch(list, key, 0, list.length - 1)

private binarySearch(list, key, low, high):
   if low > high:
      return false
   else
      mid = (low + high) / 2
      if (key == list[mid]):
         return true
      else if key < data[mid]:
         return binarySearch(data, key, low, mid - 1)
      else if key > data[mid]:
         return binarySearch(data, key, mid + 1, high)
```

Infinite Recursion

- If the recursive function does not sufficiently reduce the problem to the base case, the function will not end **(infinite recursion)**.

- Successive recursive calls cause the Stack to fill up to capacity with method calls.
  - the program will crash.
  - **StackOverflowError**

Towers of Hanoi

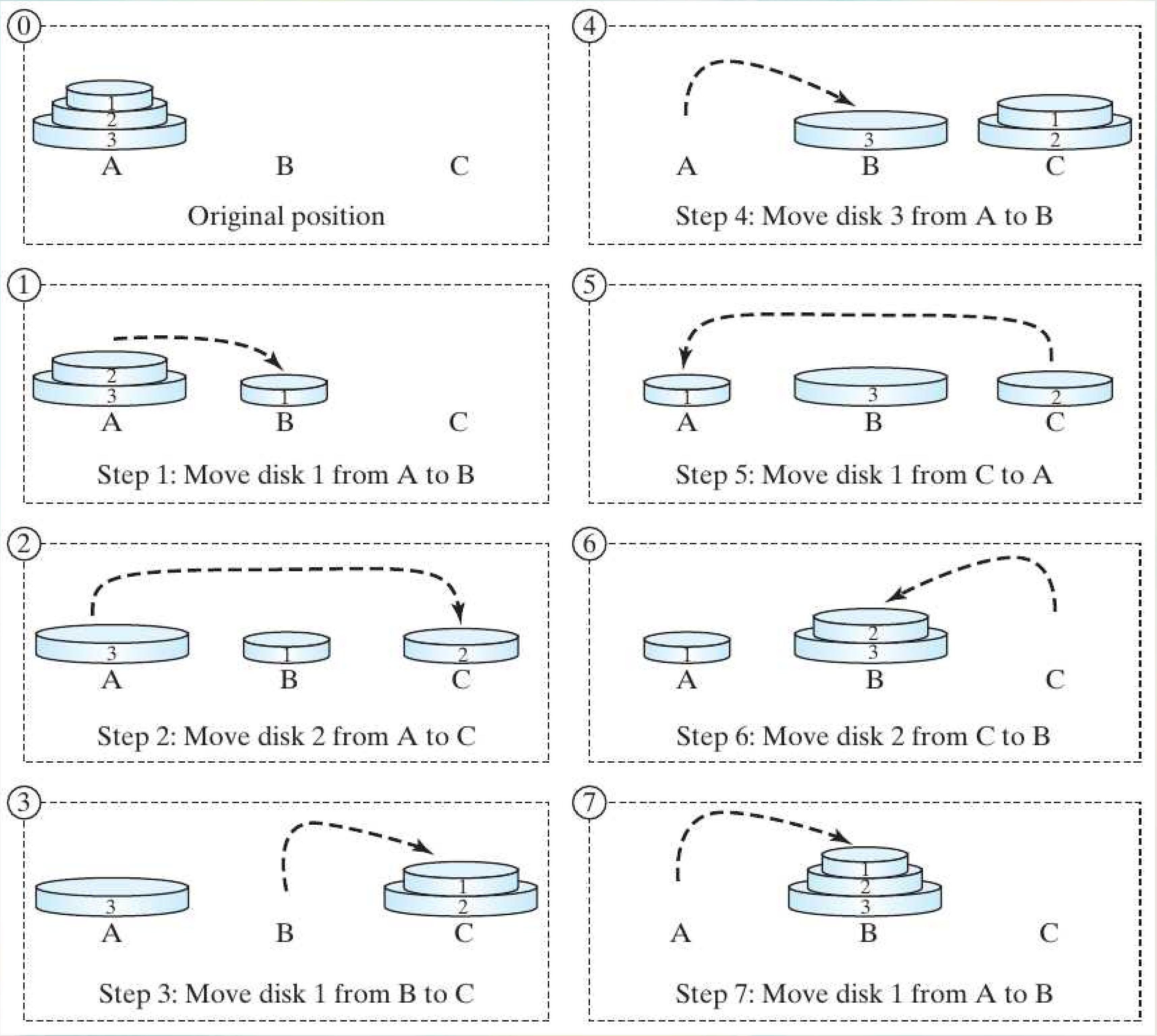
Towers of Hanoi

- A classic problem that can be solved very easily using recursion, but is difficult to solve without.

- The Problem: Given a number of disks of distinct sizes and given three "towers" we want to move the disks from one tower to another following these rules:
  - There are ***n*** disks labeled 1, 2, 3, … , *n* and three towers labeled A, B, and C.
  - No disk can be on top of a smaller disk at any time
  - All disks are initially placed on tower A.
  - Only one disk can be moved at a time, and it must be the smallest disk on a tower.

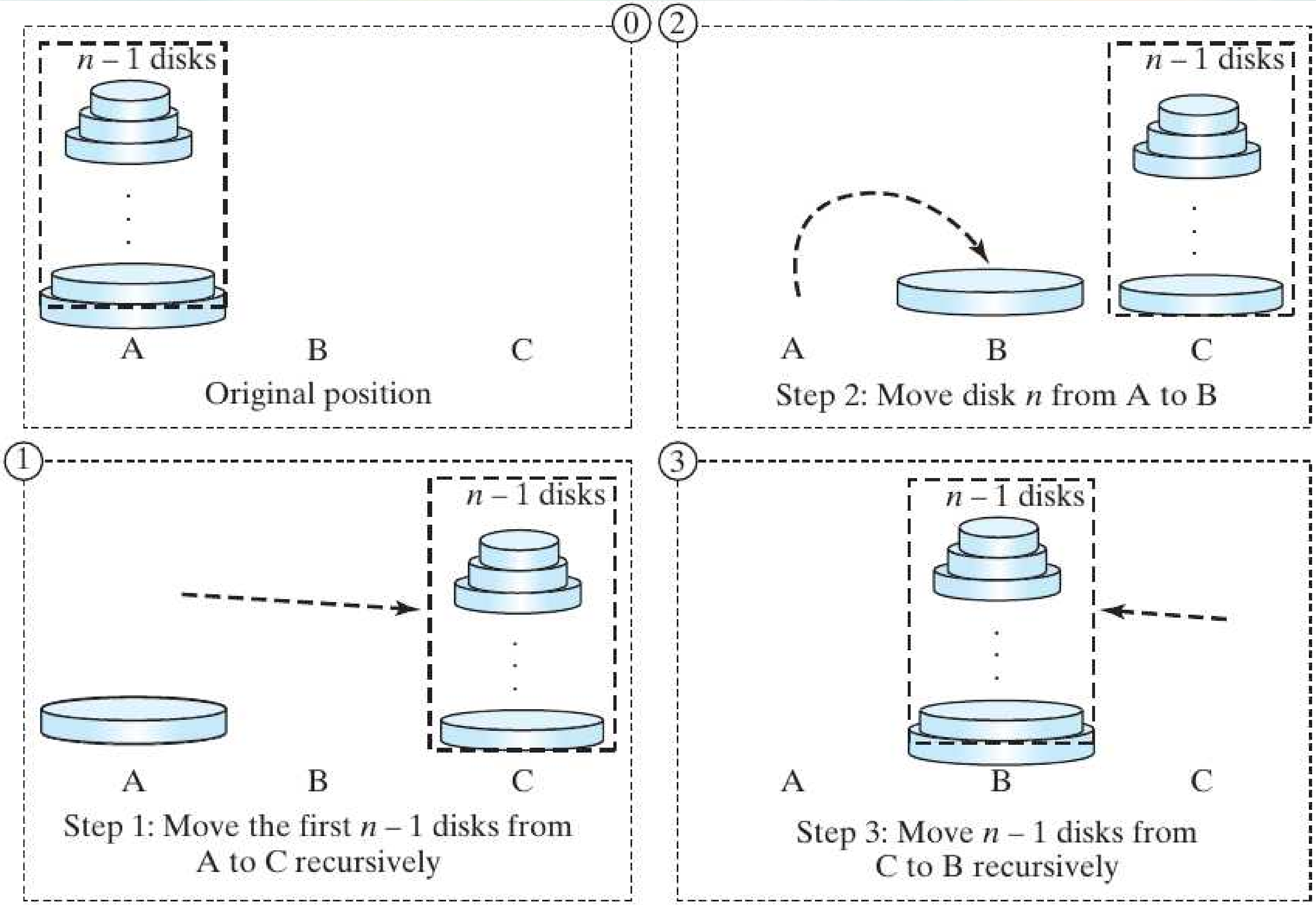
- The Objective: move all the disks from Tower A to Tower B with the assistance of Tower C.

- Example: The following slide shows the moves required when there are 3 disks.


0
A
B
C
Original position
1
A
B
C
Step 1: Move disk 1 from A to B
2
A
B
C
Step 2: Move disk 2 from A to C
3
A
B
C
Step 3: Move disk 1 from B to C
4
A
B
C
Step 4: Move disk 3 from A to B
5
A
B
C
Step 5: Move disk 1 from C to A
6
A
B
C
Step 6: Move disk 2 from C to B
7
A
B
C
Step 7: Move disk 1 from A to B

Towers of Hanoi

- The base case:
  - when n = 1, you can just move the disk from A to B
- When n > 1, the original problem can be split into three subproblems:
  - Move the first n - 1 disks from A to C using tower B
  - Move disk n from A to B
  - Move n - 1 disks from C to B using tower A


0
$n-1$ disks
A
B
C
Original position
2
$n-1$ disks
A
B
C
Step 2: Move disk n from A to B
1
$n-1$ disks
A
B
C
Step 1: Move the first $n-1$ disks from A to C recursively
3
$n-1$ disks
A
B
C
Step 3: Move $n-1$ disks from C to B recursively

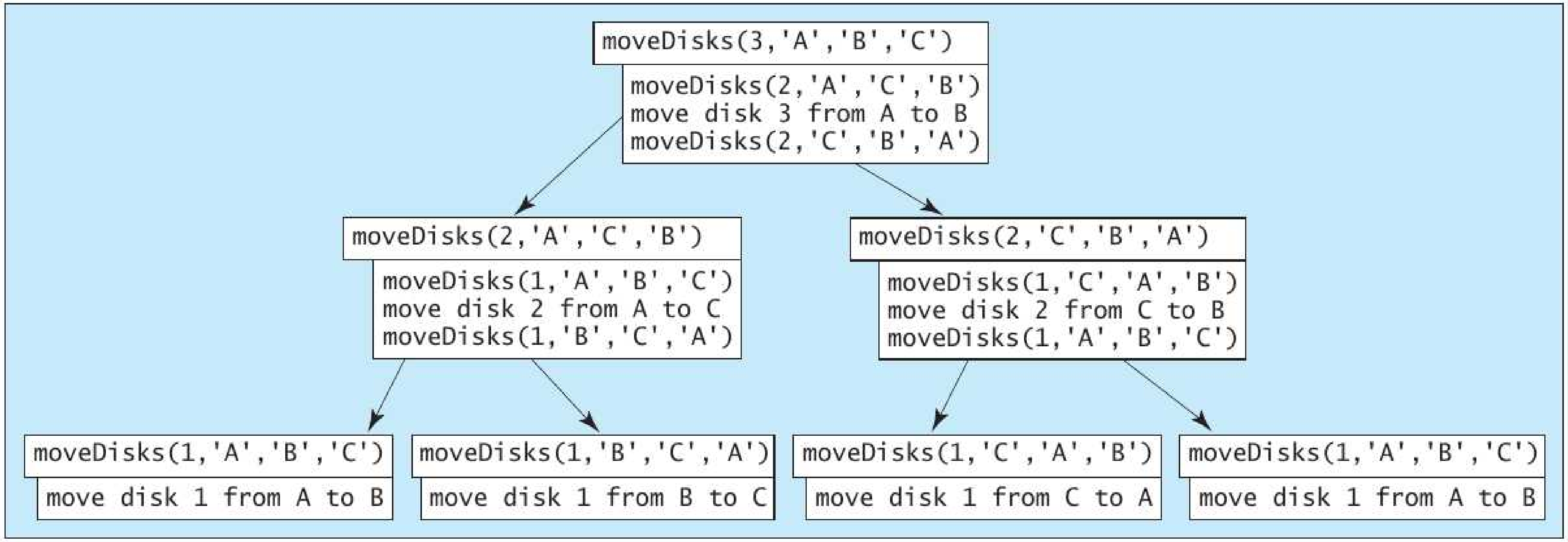
The Towers of Hanoi problem can be decomposed into three subproblems.

Towers of Hanoi

- The algorithm for the method which moves n disks from the **`fromTower`** to the **`toTower`** with the assistance of the **`auxTower`** is as follows:

```
void moveDisks(int n, char fromTower, char toTower, char auxTower) {
  if (n == 1) //The base case
    Move disk 1 from the fromTower to the toTower;
  else {
    moveDisks(n - 1, fromTower, auxTower, toTower);
    Move disk n from the fromTower to the toTower;
    moveDisks(n - 1, auxTower, toTower, fromTower);
  }
}
```

Towers of Hanoi



Backtracking and N-Queens

Backtracking

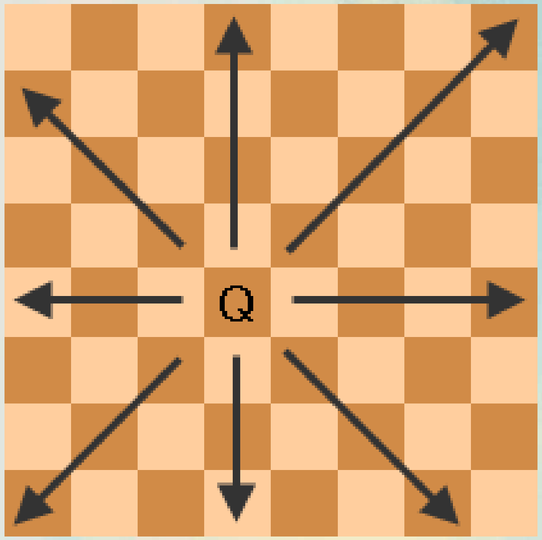
- problem solving technique that builds up partial solutions that get increasingly closer to the goal
- if a partial solution cannot be completed then that solution is abandoned and other possibilities are examined
- backtracking examples:
  - crossword puzzles
  - maze solvers
  - solutions to systems constrained by rules

Backtracking

- To use backtracking with a problem, there need to be two characteristic properties:
  - 1. a procedure to examine a partial solution and determine whether to:
    - accept it as an actual solution
    - abandon the partial solution (since it could never be a valid solution)
    - continue extending the current solution
  - 2. a procedure to extend a partial solution, generating one or more solutions that come closer to the goal.

N-Queens

- Problem:
  - A classic recursive example where the goal is to position n Queen chess pieces on a chess board in such a way that no queen can attack another.
  - A queen can attack another queen if the two queens are on the same row, column, or diagonal.

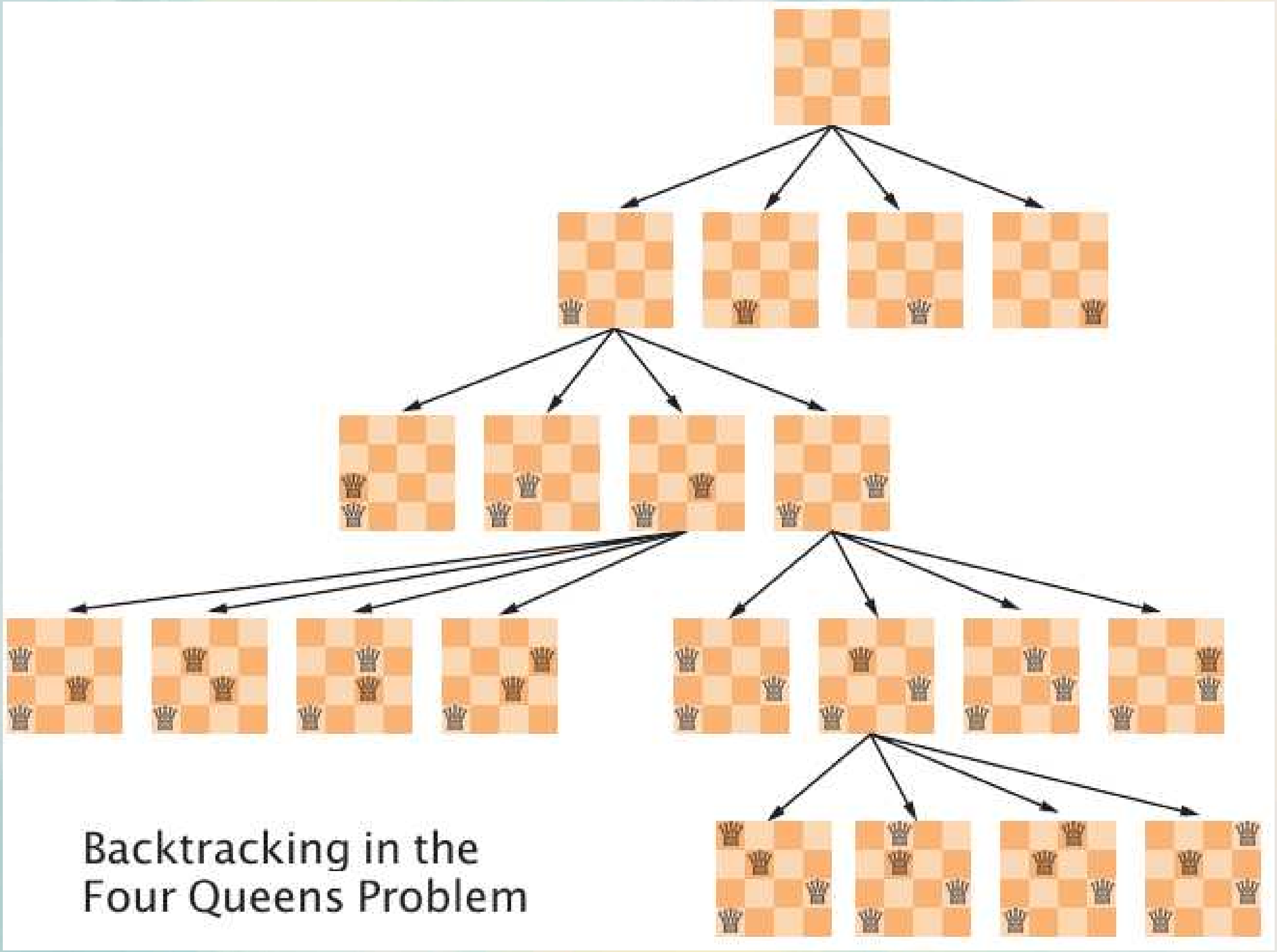


N-Queens

- A partial solution is easy to find:
  - if two queens attack another reject it
  - otherwise if eight queens have been placed on the board accept the solution.
  - otherwise continue placing more queens and checking solutions

- To make the solution more efficient, we will just place a queen in the next row after the row where the last queen was placed.

N-Queens



N-Queens

```
board: a 2D board, should be an n x n board
row: the current row number

public nqueens(board, row):
   if board has all queens:
      print board
   else:
      for each column in current row:
         place a queen in current column
         if no conflict with other queens:
            nqueens(board, row + 1)
         remove queen // backtracking part

   if row = 0:
      print "no solution"
```

Tail Recursion

Tail Recursion

- Recursive methods are either:
  - ***tail recursive:*** a recursive method which has the recursive call as the last statement in the method.
  - non-tail recursive: a recursive method that is not tail recursive.

- Was our previous factorial example tail recursive?
  - NO! when coming back to the method from the previous recursive call, there is still the pending multiplication operation.
  - Factorial is non-tail recursive.

Tail Recursion

- Is this method tail recursive?

```
public static void tailOrNonTail(int i) {
    if (i > 0) {
        System.out.print(i + " ");
        tailOrNonTail(i – 1);
    }
}
```

- What about this one?

```
public static void test(int i) {
    if (i > 0) {
        test(i - 1);
        System.out.print(i + " ");
        test(i - 1);
    }
}
```

Tail Recursion

- In many programming languages, compilers can convert tail recursion into iterative code (code using loops)

- This is more efficient since it does not create new instances on the call stack and since, in many cases, it can avoid redundant calculations.

- This is called ***Tail-Call Optimization***, or ***TCO***.

- Java compilers may have this feature in the future, but not at the current time

Tail Recursion

- Tail recursion may be desirable: because the method ends when the last recursive call ends, there is no need to store the intermediate calls in the stack

- Tail Recursive methods are very easy to convert to iterative methods

```
public static void tail(int i) {
   if (i > 0) {
     System.out.print(i + " ");
     tail(i – 1);
   }
}
```

$\Rightarrow$

```
public static void loop(int i) {
   while (i > 0) {
      System.out.println(i + " ");
      i--;
   }
}
```

Tail Recursion

- a nontail-recursive method can sometimes be converted to a tail-recursive method by using auxiliary parameters.
  - these parameters contain the result.

- incorporate pending operations into auxiliary parameters so the recursive call no longer has a pending operation.

- can define a new auxiliary recursive method which may overload the original method.

Tail Recursion

- Here is an example of Factorial rewritten to use tail recursion.

```
public static long factorial(int n) {
   return factorial(n, 1);
}

private static long factorial(int n, int result) {
   if (n == 0) {
      return result;
   }
   else {
      return factorial(n - 1, n * result);
   }
}
```

Thinking Recursively

- Recursion is everywhere, you just have to start thinking recursively to see it.
- For example: something as simple as drinking coffee can be "solved recursively"

```
public static void drinkCoffee(Cup cup) {
  if (!cup.isEmpty()) {
    cup.takeOneSip(); //Take a sip
    drinkCoffee(cup); //Recurse
  }
  else {
    cup.wash(); //You are finished.
  }
}
```