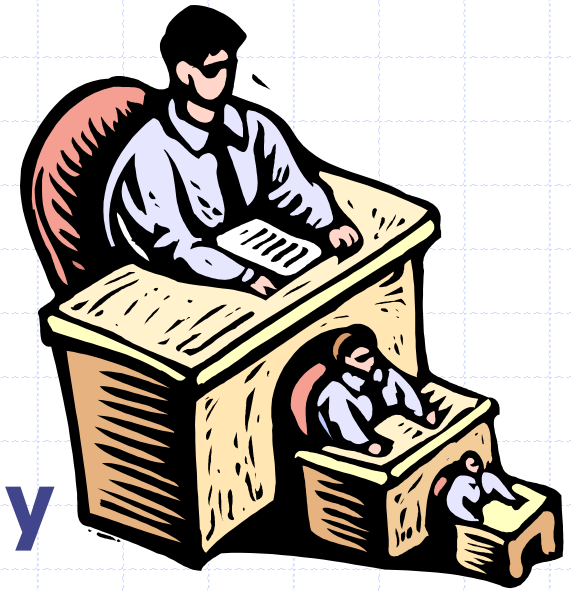
CS 2233-01

Data Structures and Algorithms

Chapter 5 Recursion



Instructor: Dr. Qingguo Wang
College of Computing & Technology
Lipscomb University

Recursion

- Recursion has two uses in computer science and software engineering, namely:
  - as a way of describing, defining, or specifying things.
  - as a way of designing solutions to problems (divide and conquer).

- It is also a programming technique.

Recursive Definition

- A recursive definition is one in which something is defined in terms of itself.
- All recursive definitions require two parts
  - **Base case**: values of the input variables for which we perform no recursive calls
  - **Recursive step**: the step that is defined in terms of itself
  - Any recursive call must make progress toward a base case
- Algorithm requiring looping can be defined iteratively or recursively.

Iterative Definition

- In general, we can define the factorial function in the following way:

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

- This is an ***iterative*** definition of the factorial function, because the definition only contains the algorithm parameters and not the algorithm itself

Recursive Definition

- We can also define the factorial function in the following way:
-

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

Iterative vs. Recursive

- **Iterative**

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 & \text{if } n>0 \end{cases}$$

Function does NOT calls itself

- **Recursive**

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

Function calls itself

Iterative vs Recursive Program

- Now that we know the difference between an iterative algorithm and a recursive algorithm, we will develop both an iterative and a recursive algorithm to calculate the factorial of a number.

- We will then compare the 2 algorithms.

Iterative Algorithm

```
factorial(n) {
    i = 1
    factN = 1
    loop (i <= n)
        factN = factN * i
        i = i + 1
    end loop
    return factN
}
```

The iterative solution is very straightforward. We simply loop through all the integers between 1 and n and multiply them together.

Recursive Algorithm

```
factorial(n) {
    if (n = 0)
        return 1
    else
        return n*factorial(n-1)
    end if
}
```

Note how much simpler the code for the recursive version of the algorithm is as compared with the iterative version

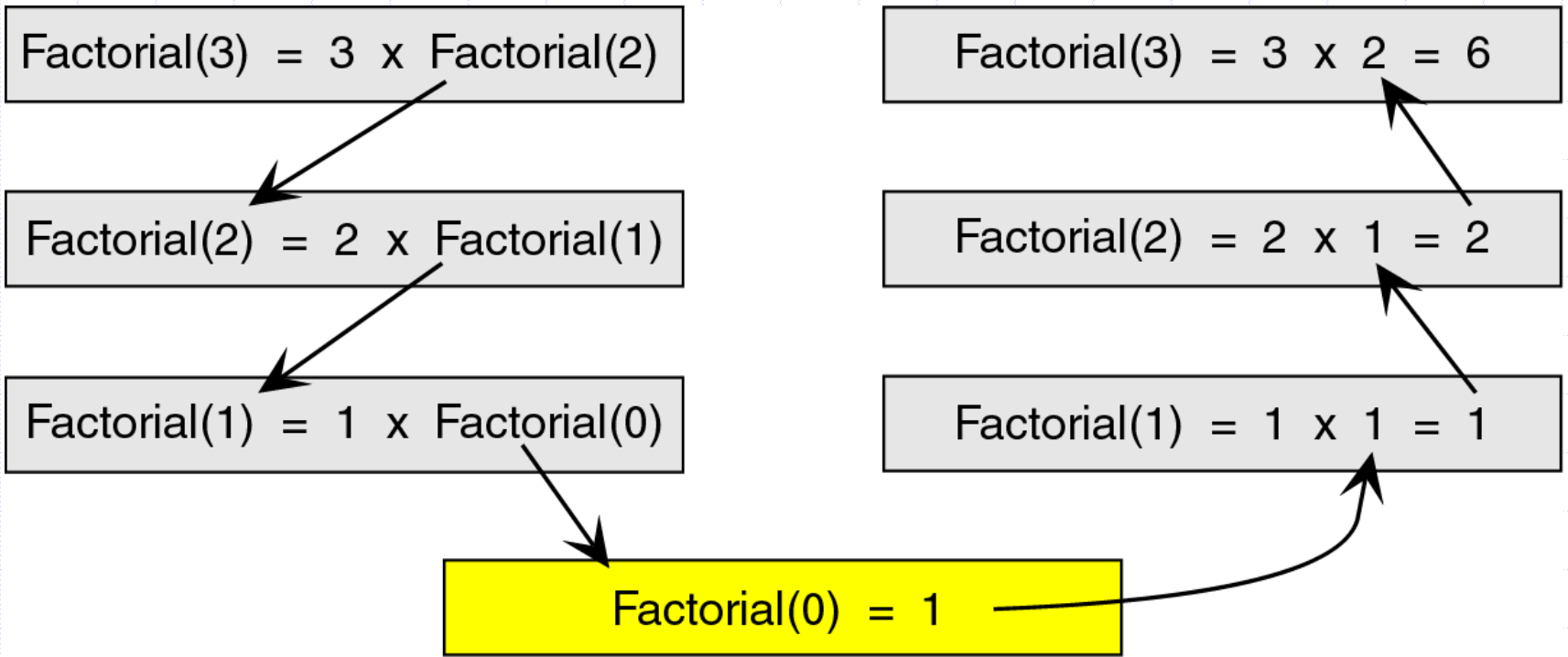
we have eliminated the loop and implemented the algorithm with 1 ‘if’ statement.

Content of a Recursive Method

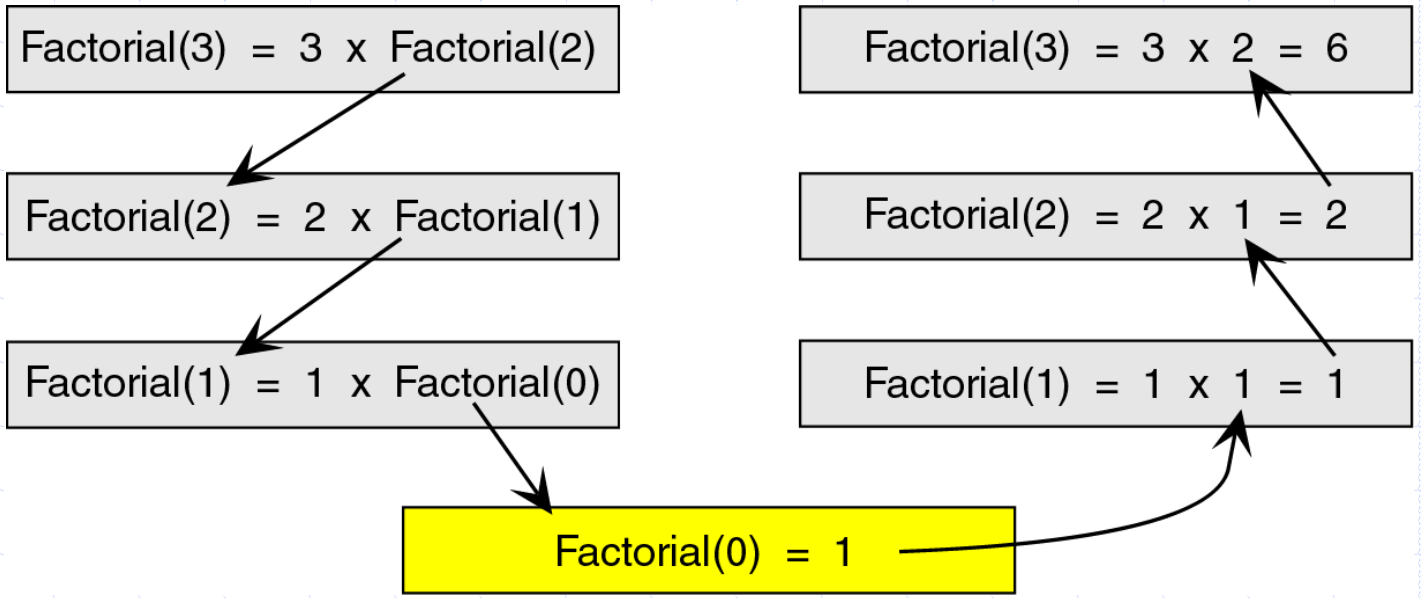
- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Recursion

- To see how the recursion works, let's break down the factorial function to solve factorial(3)



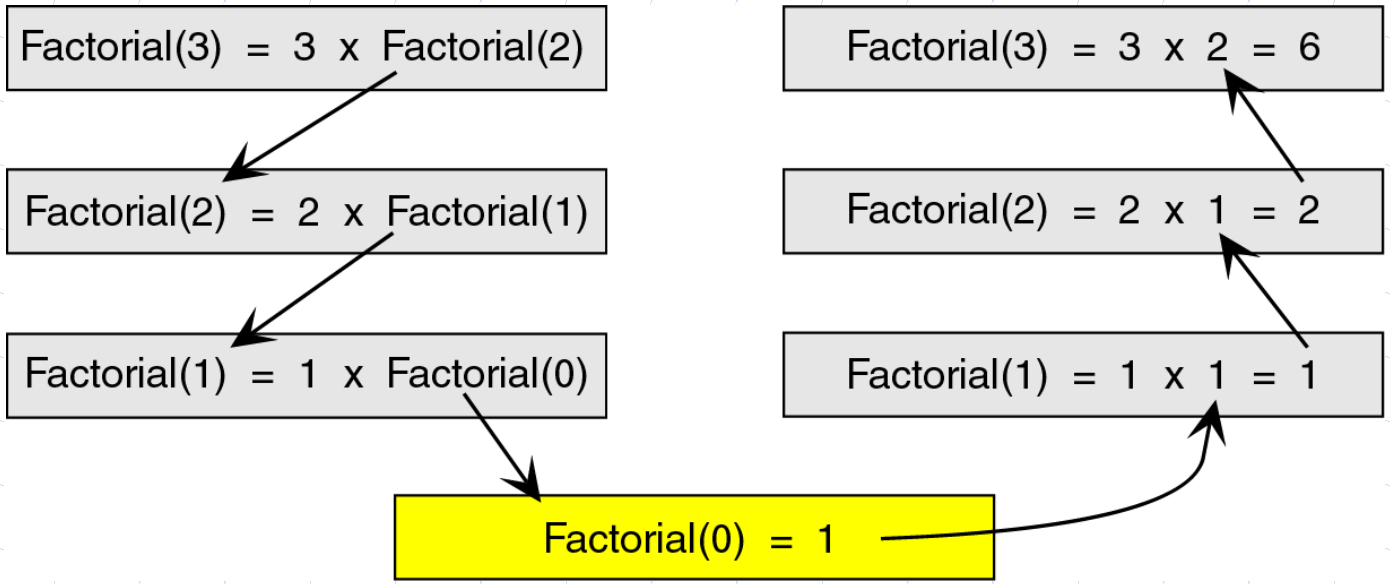
Breakdown




```
factorial(n) {
    if (n = 0)
        return 1
    else
        return n*factorial(n-1)
    end if
}
```

- Here, we see that we start at the top level, factorial(3), and simplify the problem into 3 x factorial(2).
- Now, we have a slightly less complicated problem in factorial(2), and we simplify this problem into 2 x factorial(1).

Breakdown (cont.)




```
factorial(n) {
    if (n = 0)
        return 1
    else
        return n*factorial(n-1)
    end if
}
```

- We continue this process until we are able to reach a problem that has a known solution.
- In this case, that known solution is factorial(0) = 1.
- The functions then return in reverse order to complete the solution.

Searching algorithm

- Unsorted array
- Sorted array

A Classic Recursive Algorithm: Binary Search

- ***Binary search*** is used to efficiently locate a target value within a sorted sequence of *n* elements stored in an array
- The most important of computer algorithms
- The reason that we so often store data in sorted order

Introduction to Binary Search

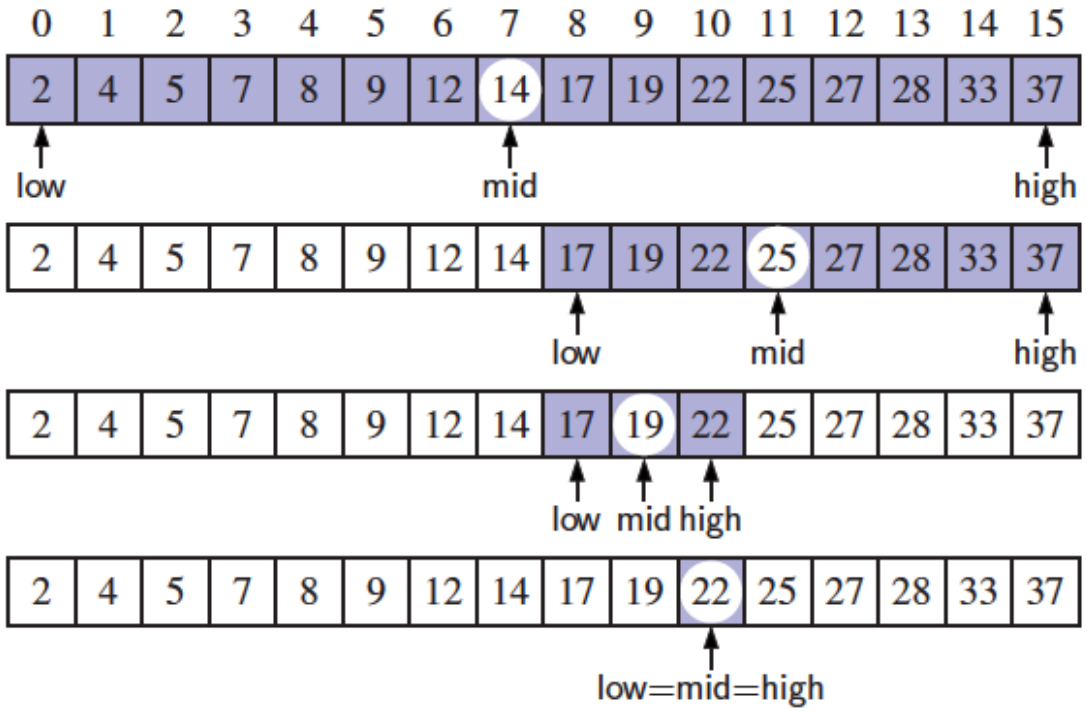
- When the sequence is ***sorted*** and ***indexable***, there is a more efficient algorithm
  - Initially, low = 0 and high = $n - 1$. We then compare the target value to the ***median candidate***

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor$$

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

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Recursive Implementation

Search for an integer in an ordered list

```
1  /**
2   * Returns true if the target value is found in the indicated portion of the data array.
3   * This search only considers the array portion from data[low] to data[high] inclusive.
4   */
5  public static boolean binarySearch(int[ ] data, int target, int low, int high) {
6    if (low > high)
7      return false;                                    // interval empty; no match
8    else {
9      int mid = (low + high) / 2;
10     if (target == data[mid])
11       return true;                                   // found a match
12     else if (target < data[mid])
13       return binarySearch(data, target, low, mid - 1);   // recur left of the middle
14     else
15       return binarySearch(data, target, mid + 1, high);  // recur right of the middle
16   }
17 }
```

Analyzing Binary Search

- The initial portion of the list is of size high – low + 1
- After one comparison, this becomes one of the following:

$$(\text{mid}-1)-\text{low}+1=\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor-\text{low}\leq\frac{\text{high}-\text{low}+1}{2}$$

$$\text{high}-(\text{mid}+1)+1=\text{high}-\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor\leq\frac{\text{high}-\text{low}+1}{2}.$$

- Thus, each recursive call divides the search region in half; hence, there can be at most **log n** levels
- Runs in O(log n) time.

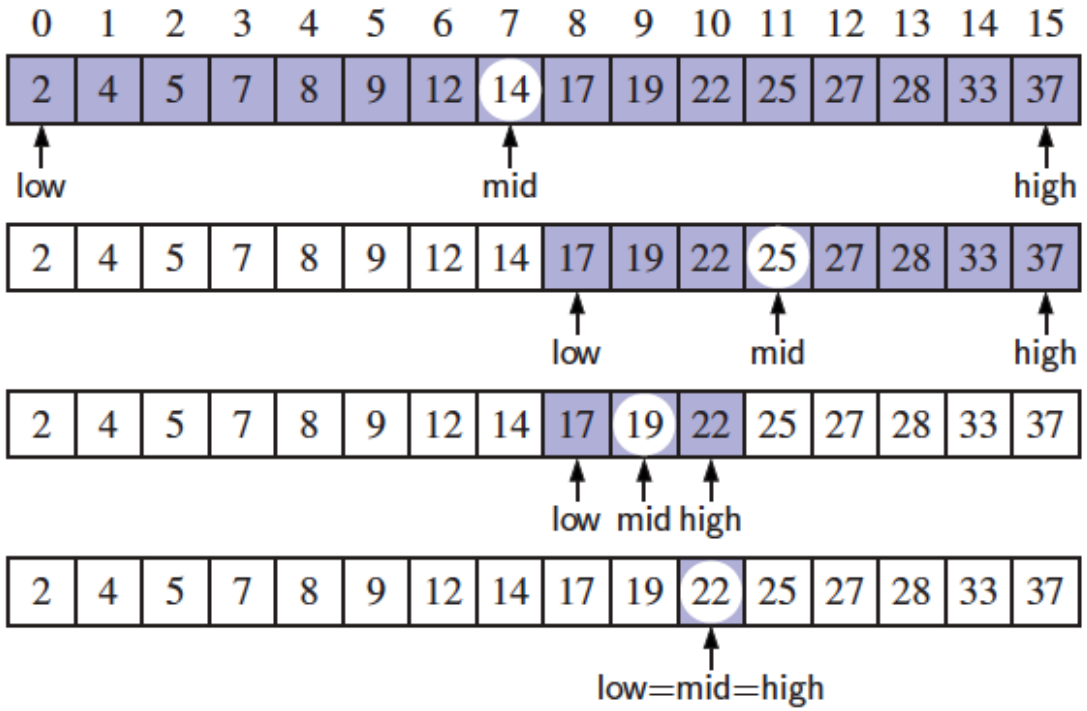
Recaps of Recursion

- Base case
- Recursive step
- Any recursive call must make progress toward a base case

```
factorial(n) {
    if (n = 0)
        return 1
    else
        return n*factorial(n-1)
    end if
}
```

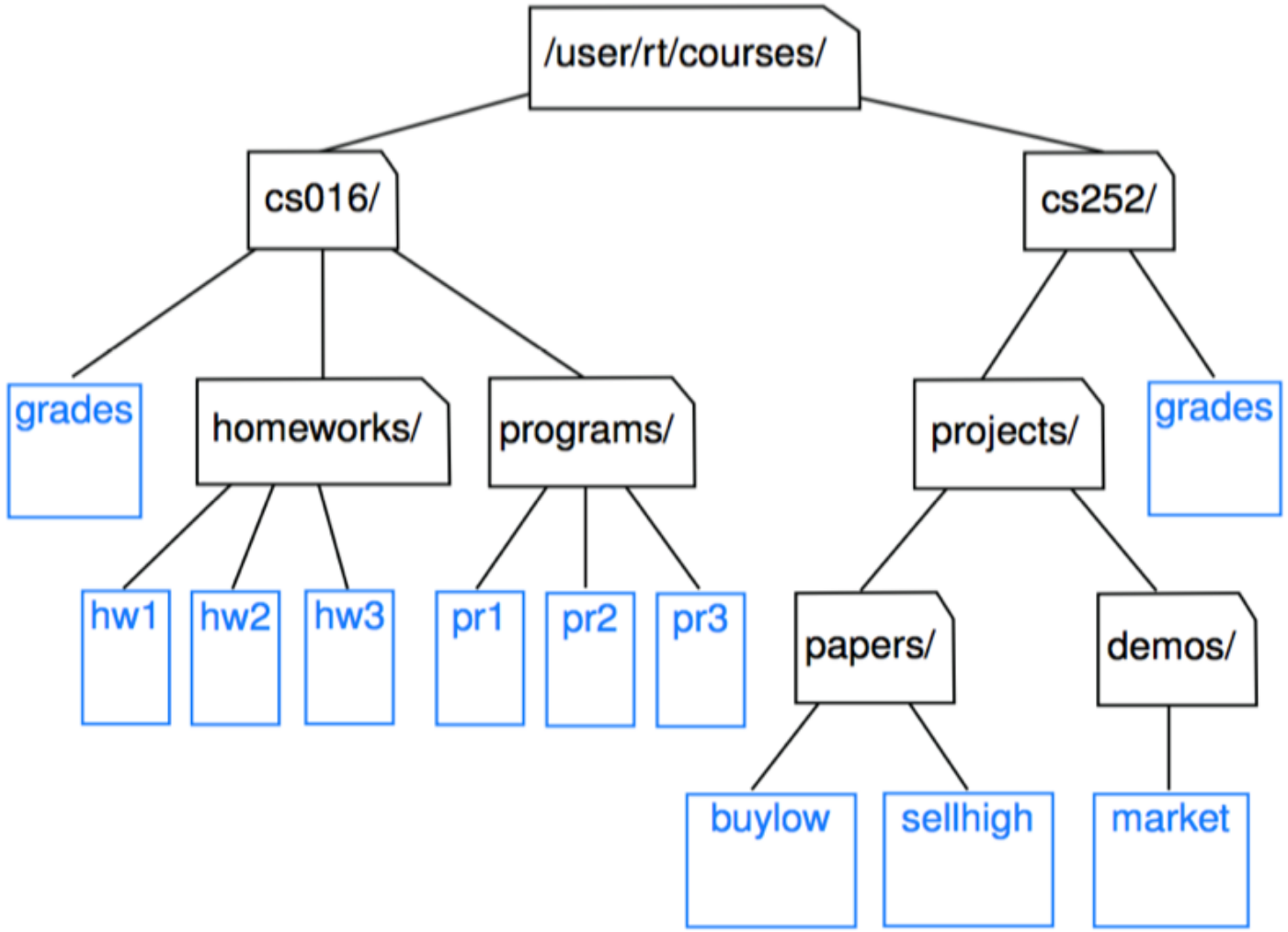
Recaps of Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Recursive File Systems

- Operating Systems define file-system directories (also called "folders") in a recursive way



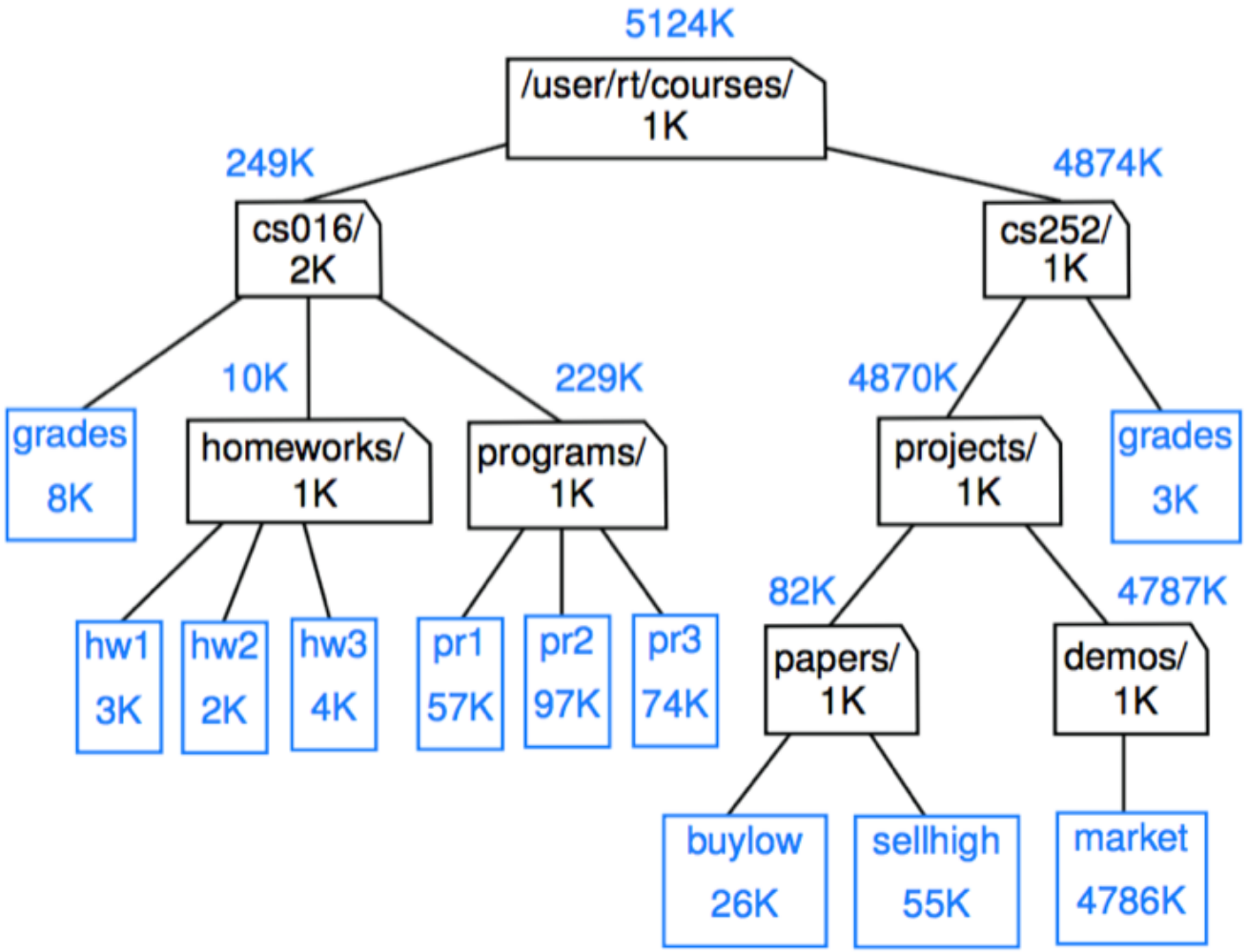
Recursive File Systems (cont.)

- Many common behaviors of an operating system, such as copying a directory or deleting a directory, are implemented with recursive algorithms.
- Consider one such algorithm: computing the total disk usage for all files and directories nested within a particular directory

Defining Disk Usage

- ***immediate*** disk space: space used by each entry
- ***cumulative*** disk space: space used by that entry and all nested features

File System Example



Defining Disk Usage

- ***immediate*** disk space: space used by each entry
- ***cumulative*** disk space: space used by that entry and all nested features
- The cumulative disk space for an entry can be computed with a simple recursive algorithm. It is equal to the immediate disk space used by the entry plus the sum of the cumulative disk space usage of any entries that are stored directly within the entry.

Recursive Algorithm

```
Algorithm DiskUsage(path):
    Input: A string designating a path to a file-system entry
    Output: The cumulative disk space used by that entry and any nested entries
    total = size(path)                          {immediate disk space used by the entry}
    if path represents a directory then
        for each child entry stored within directory path do
            total = total + DiskUsage(child)    {recursive call}
    return total
```

Implementation of the Algorithm

```
1  /**
2   * Calculates the total disk usage (in bytes) of the portion of the file system rooted
3   * at the given path, while printing a summary akin to the standard 'du' Unix tool.
4   */
5  public static long diskUsage(File root) {
6    long total = root.length( );                    // start with direct disk usage
7    if (root.isDirectory( )) {                      // and if this is a directory,
8      for (String childname : root.list( )) {       // then for each child
9        File child = new File(root, childname);     // compose full path to child
10       total += diskUsage(child);                  // add child's usage to total
11     }
12   }
13   System.out.println(total + "\t" + root);        // descriptive output
14   return total;                                   // return the grand total
15 }
```

Code Fragment 5.5: A recursive method for reporting disk usage of a file system.

Linear Recursion

- **Definition**: a recursive method that is designed so that each invocation of the body makes at most one new recursive call.
- **Examples**: factorial method, binary search algorithm
- **True or False**: linear recursion runs in O(n) time?

Linear Recursion

- **Definition**: a recursive method that is designed so that each invocation of the body makes at most one new recursive call.
- **Examples**: factorial method, binary search algorithm

- *linear recursion* terminology reflects the structure of the recursion trace, not the asymptotic analysis of the running time (e.g. binary search runs in $O(\log n)$ time).

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Summing the Elements of an Array Recursively

- We want to compute the sum of an array of n integers
- if $n = 0$ the sum is trivially 0,
- Otherwise it is the sum of the first $n - 1$ integers in the array plus the last value in the array

Recursive Algorithm for Summing an Array

```
1  /** Returns the sum of the first n integers of the given array. */
2  public static int linearSum(int[ ] data, int n) {
3    if (n == 0)
4      return 0;
5    else
6      return linearSum(data, n−1) + data[n−1];
7  }
```

Reversing an Array

Algorithm reverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as reverseArray(A, i, j), not reverseArray(A)

```
1  /** Reverses the contents of subarray data[low] through data[high] inclusive. */
2  public static void reverseArray(int[ ] data, int low, int high) {
3    if (low < high) {                          // if at least two elements in subarray
4      int temp = data[low];                    // swap data[low] and data[high]
5      data[low] = data[high];
6      data[high] = temp;
7      reverseArray(data, low + 1, high − 1);   // recur on the rest
8    }
9  }
```

Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

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

- This leads to an power function that runs in O(n) time (for we make n recursive calls)

Recursive Algorithm for Computing Powers

```
1  /** Computes the value of x raised to the nth power, for nonnegative integer n. */
2  public static double power(double x, int n) {
3    if (n == 0)
4      return 1;
5    else
6      return x * power(x, n-1);
7  }
```

Recursive Squaring

- We can do better than this, however
- We can derive a more efficient linearly recursive algorithm by using repeated squaring, for example,

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128$

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 \cdot x & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

Recursive Squaring Method

Algorithm Power(x, n):
  Input: A number x and integer n = 0
  Output: The value x^n
  if n = 0 **then**
    return 1
  if n is odd **then**
    y = Power(x, (n - 1)/ 2)
    return x · y ·y
  else
    y = Power(x, n/ 2)
    return y · y

Analysis

```
Algorithm Power(x, n):
    Input: A number x and integer n = 0
    Output: The value x^n
    if n = 0    then
        return 1
    if n is odd then
        y  = Power(x, (n - 1)/ 2)
        return x · y · y
    else
        y = Power(x, n/ 2)
        return y · y
```

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

Tail Recursion

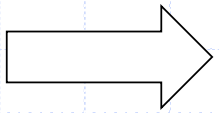
- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- For tail recursion, the return value of the recursive call (if any) is immediately returned by the enclosing recursion
  - Tail recursion example
    - reverseArray(A, i + 1, j - 1);
    - return;
  - This is not tail recursion
    - return n * factorial(n–1);
- Tail recursion can be easily converted to non-recursive methods.

Tail Recursion

```
Algorithm reverseArray(A, i,  j):
    Input: An array A and nonnegative integer indices i and  j
    if i <  j then
            Swap A[i] and A[ j]
            reverseArray(A, i + 1,  j – 1)
    return
```

Tail Recursion

```
Algorithm reverseArray(A, i,  j):
    Input: An array A and nonnegative integer indices i and  j
    if i <  j then
            Swap A[i] and A[ j]
            reverseArray(A, i + 1,  j – 1)
    return
```



```
Algorithm IterativeReverseArray(A, i, j ):
    Input: An array A and nonnegative integer indices i and j
    while i <  j do
        Swap A[i ] and A[ j ]
        i  = i + 1
        j  = j - 1
    return
```

Binary Recursion

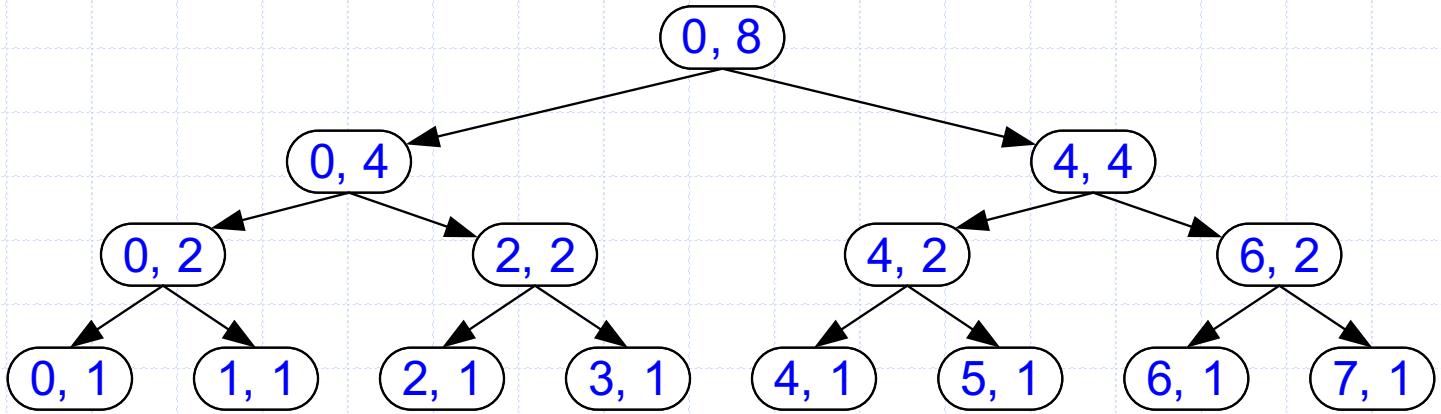
- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.
- Consider the problem of summing the n integers of an array
  - recursively compute the sum of the first half,
  - and the sum of the second half,
  - and add those sums together

Recursive Array Summation Method

- Problem: add all the numbers in an integer array A:

```
Algorithm BinarySum(A, i, n):
    Input: An array A and integers i and n
    Output: The sum of the n integers in A starting at index i
    if n = 1 then
      return A[i]
    return BinarySum(A, i, n/ 2) + BinarySum(A, i + n/ 2, n/ 2)
```

- **Example trace:**



Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- Recursive algorithm (first attempt):

```
Algorithm BinaryFib(k):
    Input: Nonnegative integer k
    Output: The kth Fibonacci number F_k
    if k <= 1 then
        return k
    else
        return BinaryFib(k − 1) + BinaryFib(k − 2)
```

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

Tree of Recursive Calls

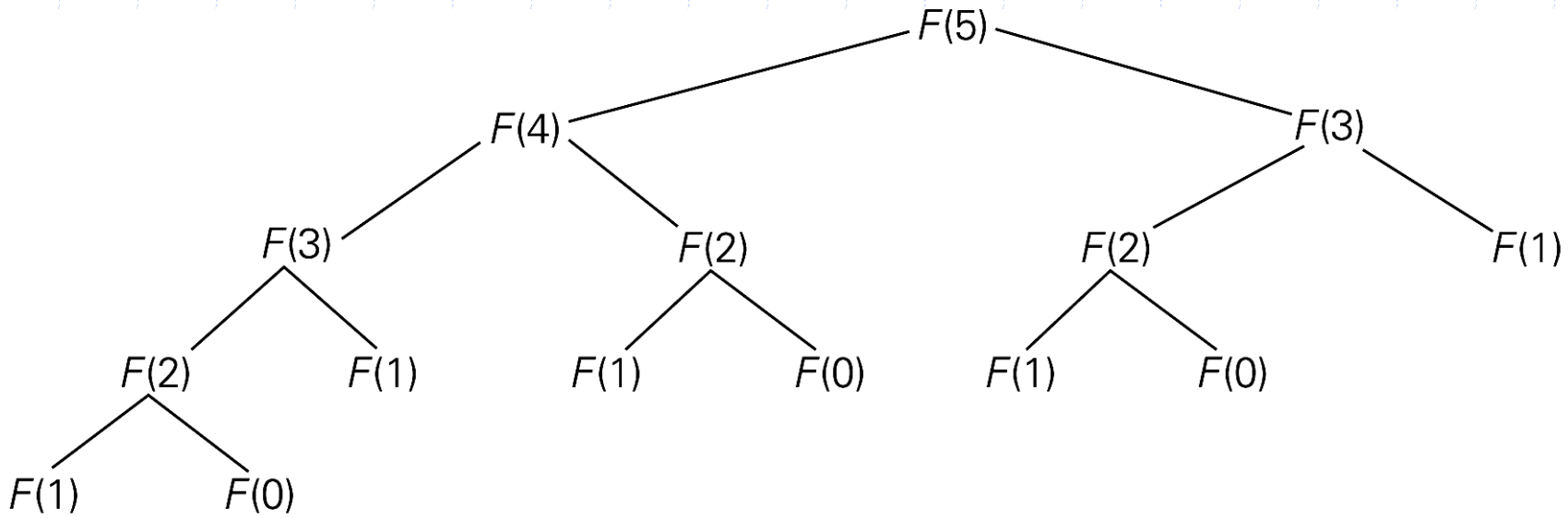


FIGURE 2.6 Tree of recursive calls for computing the 5th Fibonacci number by the definition-based algorithm

A Better Fibonacci Algorithm

- Use linear recursion instead

```
Algorithm LinearFibonacci(k):
    Input: A nonnegative integer k
    Output: Pair of Fibonacci numbers (F_k , F_{k−1})
    if k = 1 then
        return (k, 0)
    else
        (i, j) = LinearFibonacci(k − 1)
        return (i +j, i)
```

- LinearFibonacci makes k−1 recursive calls

Multiple Recursion

- Multiple recursion, e.g. DiskUsage:
  - makes potentially many recursive calls
  - not just one or two

```
Algorithm DiskUsage(path):
    Input: A string designating a path to a file-system entry
    Output: The cumulative disk space used by that entry and any nested entries
    total = size(path)                    {immediate disk space used by the entry}
    if path represents a directory then
        for each child entry stored within directory path do
            total = total + DiskUsage(child)          {recursive call}
    return total
```

Advantage of Recursion

- Recursion is a powerful problem-solving technique that often produces very clean, simple solutions to even the most complex problems.
- The program directly reflects the abstract solution strategy.
- Recursive solutions can be easier to understand and to describe than iterative solutions.

Typical Scenarios to Use Recursion

- Recursion works the best when the algorithm and/or data structure that is used naturally supports recursion.
- One such data structure is the tree (more to come).
- One such algorithm is the binary search algorithm that we discussed earlier in the course.

Disadvantage of programming recursively

- Programming recursively makes it easier to write inefficient ones.
- when we use recursion to solve problems we are interested exclusively with correctness, and not at all with efficiency. Consequently, our simple, elegant recursive algorithms may be inherently inefficient.

Overhead of Recursion

- Recursive solutions may involve extensive overhead because they use calls.
- When a call is made, it takes time to build a stackframe and push it onto the system stack.
- Conversely, when a return is executed, the stackframe must be popped from the stack and the local variables reset to their previous values – this also takes time.

Limitations of Recursion

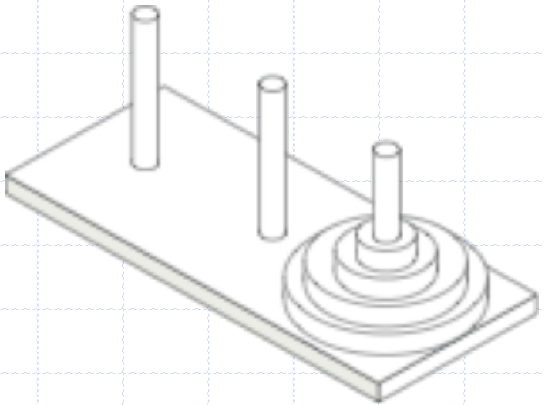
- In general, recursive algorithms run slower than their iterative counterparts.
- Also, every time we make a call, we must use some of the memory resources to make room for the stackframe.

Limitations of Recursion

- Therefore, if the recursion is deep, say, factorial(1000), we may run out of memory.
- Because of this, it is usually best to develop iterative algorithms when we are working with large numbers.

C-5.16 ***Towers of Hanoi*** puzzle

- We are given a platform with three pegs, *a*, *b*, and *c*, sticking out of it. On peg *a* is a stack of ***n*** disks, each larger than the next, so that the smallest is on the top and the largest is on the bottom. The puzzle is to move all the disks from peg *a* to peg *c*, moving one disk at a time, so that we never place a larger disk on top of a smaller one. Describe a recursive algorithm for solving the Towers of Hanoi puzzle for arbitrary *n*. (Hint: Consider first the subproblem of moving all but the *n* th disk from peg *a* to another peg using the third as "temporary storage.")



Next Week

- Chapter 6 Stacks, Queues, and Deques