Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Recursion

- One way to describe repetition within a computer program is the use of loops
- An entirely different way to achieve repetition is through a process known as **recursion**
- Recursion is a programming technique in which a method (function) calls itself
- Recursion is a technique by which a method makes one or more calls to itself *during execution*, or by which a **data structure relies upon smaller instances of the very same type** of structure in its representation
- In computing, recursion provides an elegant and powerful alternative for performing repetitive tasks
- When one invocation of the method makes a recursive call, that invocation is suspended until the recursive call completes

Recursion

- Examples of the use of recursion
  - The ***factorial function*** (commonly denoted as n!) is a classic mathematical function that has a natural recursive definition.
  - An ***English ruler*** has a recursive pattern that is a simple example of a fractal structure.
  - ***Binary search*** is among the most important computer algorithms. It allows us to efficiently locate a desired value in a data set with upwards of billions of entries.
  - The ***file system*** for a computer has a recursive structure in which directories can be nested arbitrarily deeply within other directories. Recursive algorithms are widely used to explore and manage these file systems.

Recursion

- The key features common to all recursive routines:
  - It calls itself.
  - When it calls itself, it does so to solve a smaller problem
  - There's some version of the problem that is simple enough that the routine can solve it, and return, without calling itself
- In each successive call of a recursive method to itself, the ***argument becomes smaller*** reflecting the fact that the ***problem has become "smaller" or easier***
- When the argument or range reaches a certain minimum size, a condition is triggered and the method returns without calling itself
- Calling a method involves certain overhead
  - *Control must be transferred* from the location of the call *to the beginning of the method*
  - In addition, ***the arguments*** to the method and the address to which the method should return must be ***pushed onto an internal stack*** so that the method can access the argument values and know where to return
  - if there is a large amount of data, leading to ***stack overflow***
- Recursion is usually used because it **simplifies a problem conceptually**, not because it's inherently more efficient

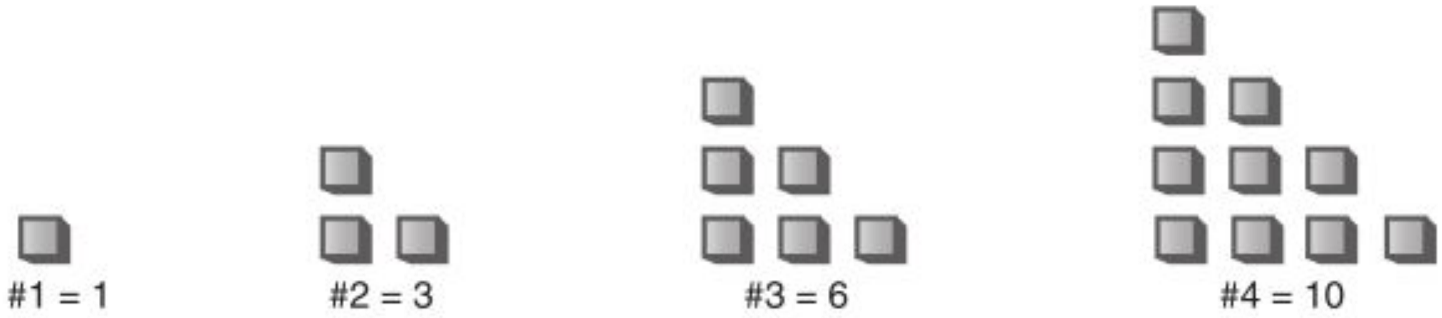
Mathematical Induction

- Recursion is the programming equivalent of mathematical induction
- Mathematical induction is a way of defining something in terms of itself
- Using induction, we could define the triangular numbers mathematically by saying

$$tri(n) = 1 \qquad \text{if } n = 1$$

$$tri(n) = n + tri(n-1) \qquad \text{if } n > 1$$

- Trangular number is the series of numbers 1, 3, 6, 10, 15, 21, …, where the *n^{th} term in the series is obtained by adding n to the previous term*



The Factorial Function

- The factorial of a positive integer ***n***, denoted ***n!***, is defined as the product of the integers from 1 to n
- If n = 0, then n! is defined as 1 by convention
- More formally, for any integer n ≥ 0,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

- The number of ways in which ***n*** distinct items can be arranged into a sequence, that is, the number of permutations of ***n*** items = n!
    - For example, the three characters a, b, and c can be arranged in 3! = 3 · 2 · 1 = 6 ways: abc, acb, bac, bca, cab, and cba
- More generally, for a positive integer n, we can define ***n!*** to be ***n · (n–1)!***

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

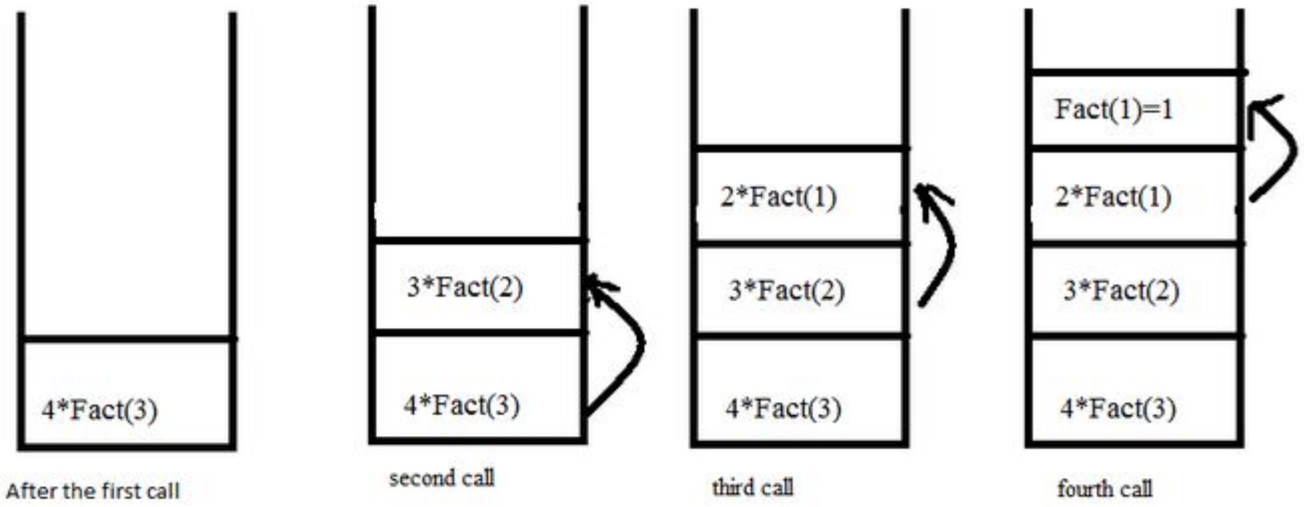
A Recursive Implementation of the Factorial Function

- This method does not use any explicit loops
- Repetition is achieved through repeated recursive invocations of the method
- The process is finite because each time the method is invoked, its argument is smaller by one, and when a base case is reached, no further recursive calls are made

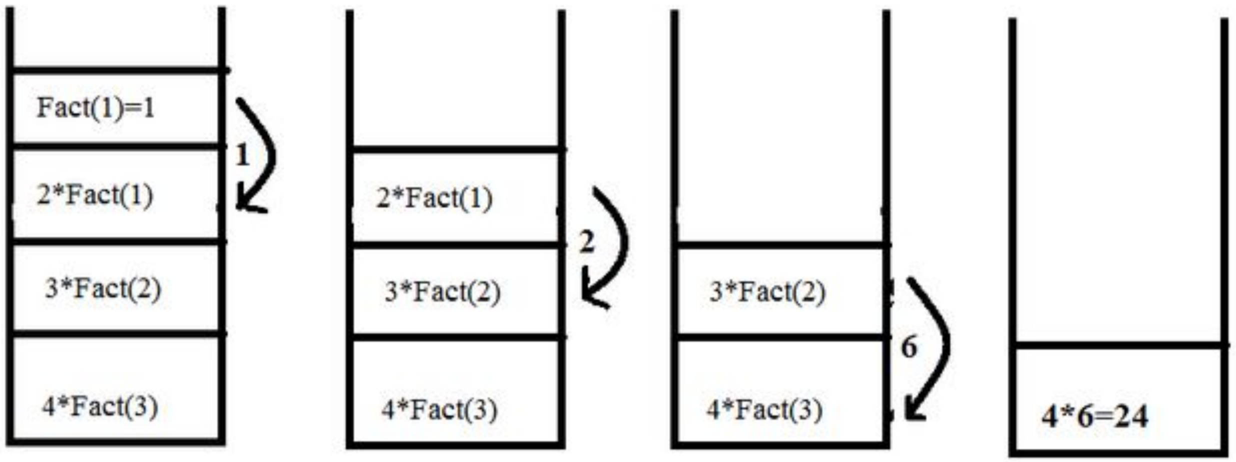
```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );    // argument must be nonnegative
  else if (n == 0)
    return 1;                                 // base case
  else
    return n * factorial(n-1);                // recursive case
}
```

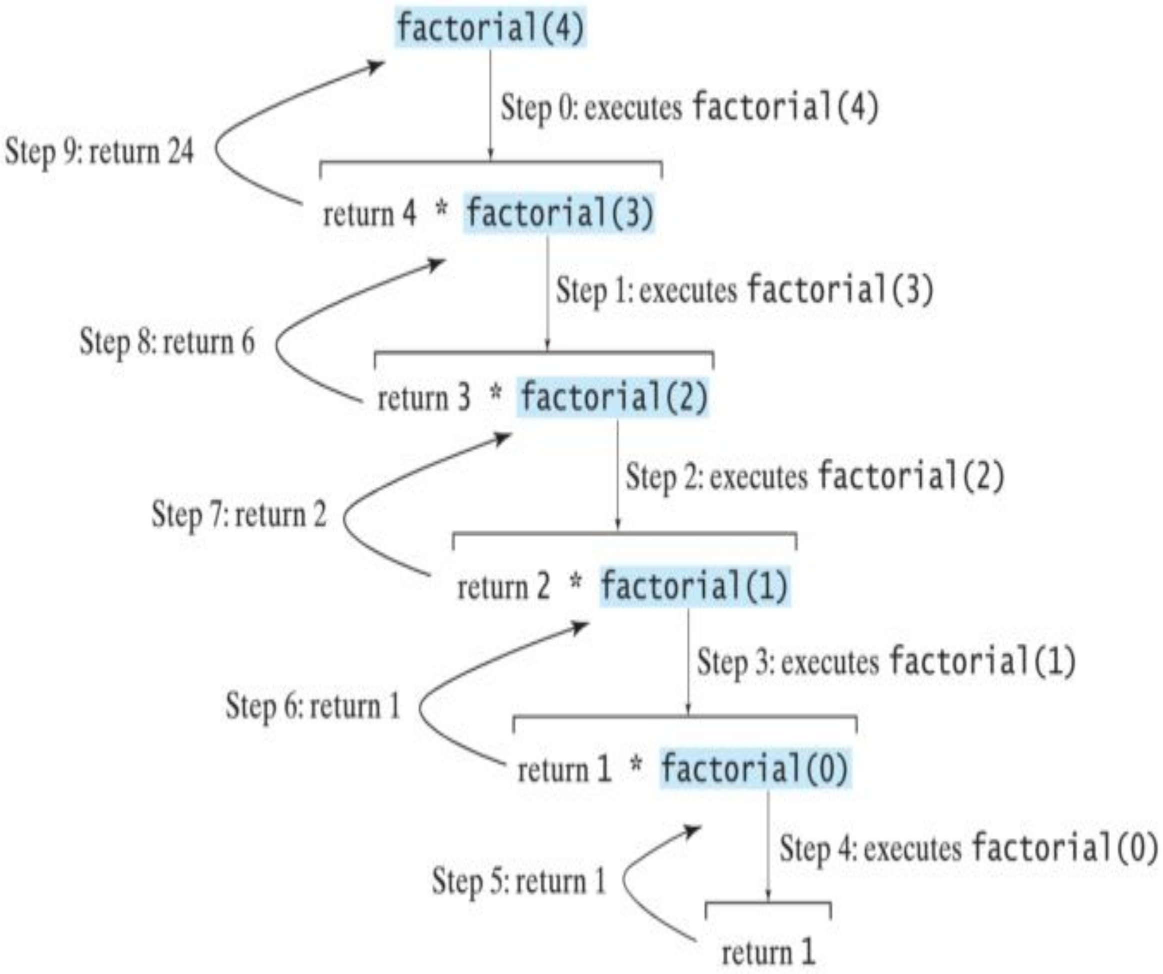
Factorial(4) Function recursion call trace

When function call happens previous variables gets stored in stack



Returning values from base case to caller function





Binary Search

- Binary search is an efficiently *locate a target value within a* *sorted sequence* of n elements stored in an array
- When the sequence is unsorted, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set
- This algorithm is known as **linear search**, **or** **sequential search**, and it runs in **O(n) time**
- When the sequence is sorted and indexable, there is a more efficient algorithm binary search, and it runs in **O(logn) time**
- This is a significant improvement, given that if ***n*** is 1 billion, ***logn*** is only 30
- The Guess-a-Number Game
  - Range 1-100
  - In this game, a friend asks you to guess a number she's thinking of between 1 and 100. When you guess a number, she'll tell you one of three things: Your guess is larger than the number she's thinking of, it's smaller, or you guessed correctly.

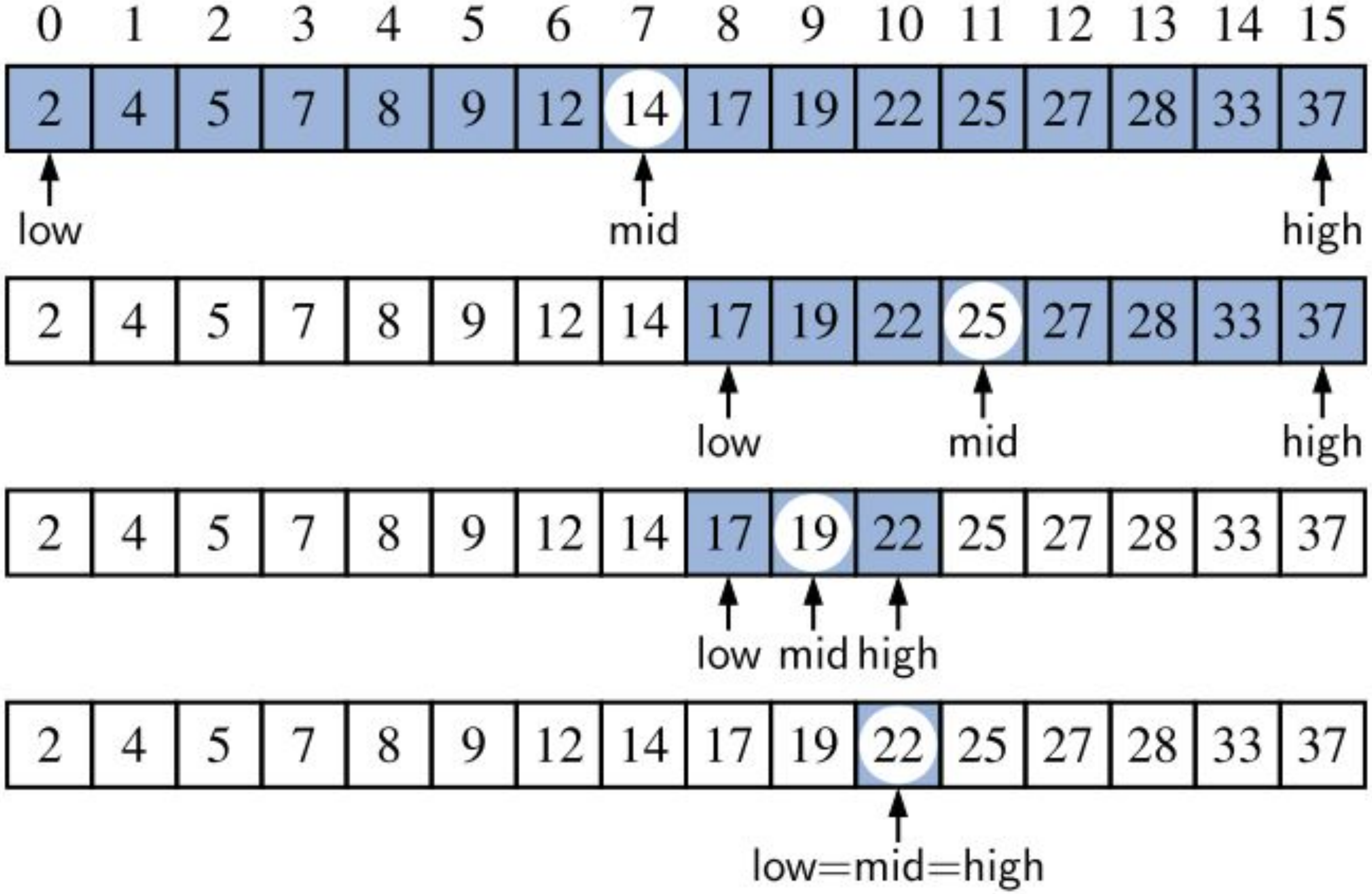
Binary Search

- Binary search is a variant of the children's game Guess-a-Number Game with "high-low" principle
- The algorithm maintains two parameters, *low* and *high*, such that all the candidate elements have index at least low and at most high
- Initially, *low = 0 and high = n−1*
- Then compare the target value to the median candidate, that is, the element with index $mid = \lfloor (low+high)/2 \rfloor$
- We consider three cases
  - If the target equals the median candidate, then we have found the item
  - If the target is less than the median candidate, then we recur on the first half of the sequence with *low = 0 and high = mid-1*
  - If the target is greater than the median candidate, then we recur on the second half of the sequence with *low = mid + 1 and high=n-1*
- *An unsuccessful search occurs if low > high, as the interval [low,high] is empty*

Binary Search Algorithm

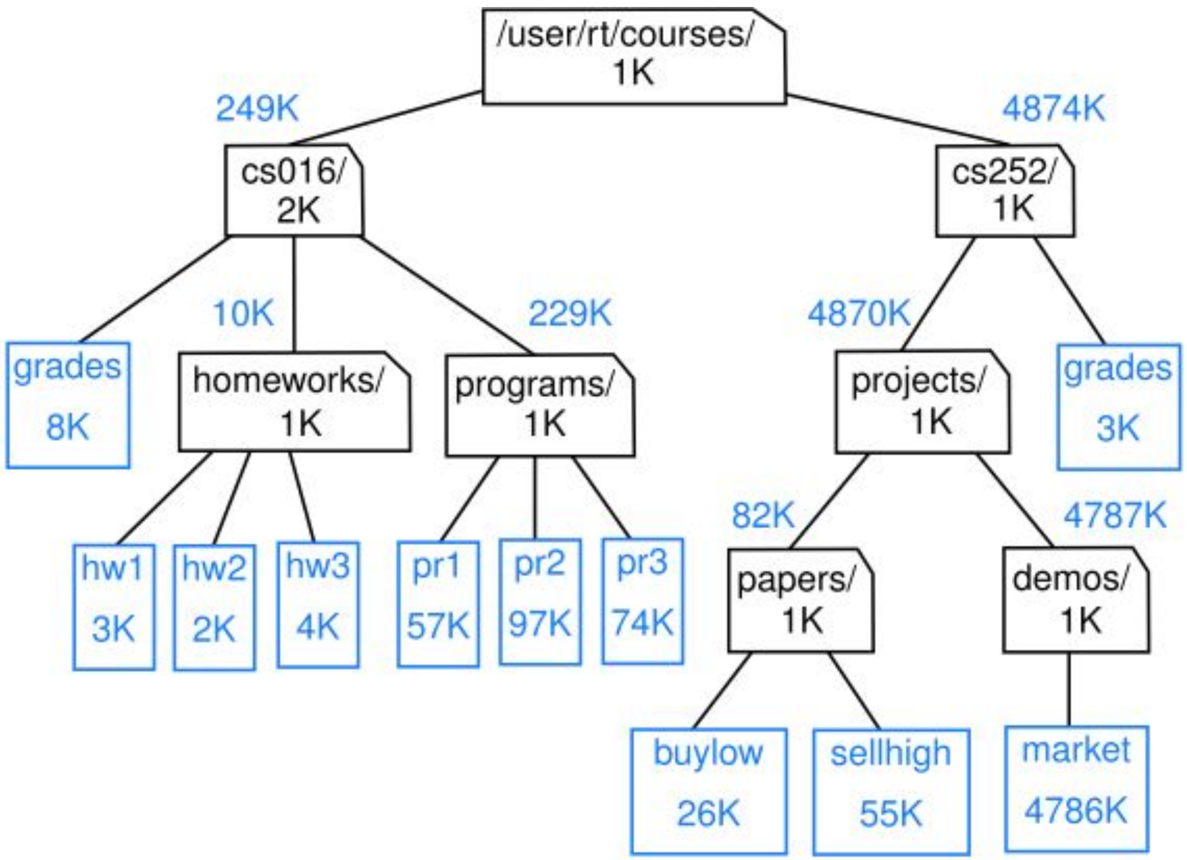
```
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                      // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                     // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);   // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);  // recur right of the middle
  }
}
```

Example of a binary search for target value 22



File Systems

- Modern operating systems define file-system directories (also called "folders") in a recursive way
- The operating system allows directories to be nested arbitrarily deeply (as long as there is enough memory)
- There must be some base directories that contain only files
- In OS, copying a directory or deleting a directory, are implemented with recursive algorithms
- We consider one such algorithm: computing the total disk usage for all files and directories nested within a particular directory



Disk usage

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*

The java.io.File Class

- We use the following methods of the class:
- new File(pathString) or new File(parentFile, childString)
  - A new File instance can be constructed either by providing the full path as a string, or by providing an existing File instance that represents a directory and a string that designates the name of a child entry within that directory.
- file.length()
  - Returns the immediate disk usage (measured in bytes) for the operating system entry represented by the File instance
- file.isDirectory()
  - Returns true if the File instance represents a directory; false otherwise
- file.list()
  - Returns an array of strings designating the names of all entries within the given directory.

Disk usage - Java Implementation

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

```
8       /user/rt/courses/cs016/grades
3       /user/rt/courses/cs016/homeworks/hw1
2       /user/rt/courses/cs016/homeworks/hw2
4       /user/rt/courses/cs016/homeworks/hw3
10      /user/rt/courses/cs016/homeworks
57      /user/rt/courses/cs016/programs/pr1
97      /user/rt/courses/cs016/programs/pr2
74      /user/rt/courses/cs016/programs/pr3
229     /user/rt/courses/cs016/programs
249     /user/rt/courses/cs016
26      /user/rt/courses/cs252/projects/papers/buylow
55      /user/rt/courses/cs252/projects/papers/sellhigh
82      /user/rt/courses/cs252/projects/papers
4786    /user/rt/courses/cs252/projects/demos/market
4787    /user/rt/courses/cs252/projects/demos
4870    /user/rt/courses/cs252/projects
3       /user/rt/courses/cs252/grades
4874    /user/rt/courses/cs252
5124    /user/rt/courses/
```