Lesson 12:

Recursion, Complexity, Searching and Sorting

Modifications By Mr. Dave Clausen
Updated for Java 1_5

Lesson 12: Recursion, Complexity, and Searching and Sorting

Objectives:

- Design and implement a recursive method to solve a problem.
- Understand the similarities and differences between recursive and iterative solutions of a problem.
- Check and test a recursive method for correctness.
- Understand how a computer executes a recursive method.

Lesson 12: Recursion, Complexity, and Searching and Sorting

Objectives:

- Perform a simple complexity analysis of an algorithm using big-O notation.
- Recognize some typical orders of complexity.
- Understand the behavior of complex sorting algorithms such as the quick sort and merge sort.

Lesson 12: Recursion, Complexity, and Searching and Sorting

Vocabulary:

- activation record
- big-O notation
- binary search algorithm
- call stack
- complexity analysis
- infinite recursion
- iterative process
- Quick Sort
- Merge Sort
- recursive method
- recursive step
- stack
- stack overflow error
- stopping state
- tail-recursive

12.1 Recursion

- A recursive algorithm is one that refers to itself by name in a manner that appears to be circular.
- Everyday algorithms, such as a recipe to bake cake or instructions to change car oil, are not expressed recursively, but recursive algorithms are common in computer science.
- Complexity analysis is concerned with determining an algorithm's efficiency.

12.1 Recursion

- Java's looping constructs make implementing this process easy.

    sum(1) = 1

    sum(N) = N + sum(N - 1) if N > 1

- Consider what happens when the definition is applied to the problem of calculating sum(4):

    sum(4) = 4 + sum(3)

    = 4 + 3 + sum(2)

    = 4 + 3 + 2 + sum(1)

    = 4 + 3 + 2 + 1

- The fact that sum(1) is defined to be 1 without making reference to further invocations of sum saves the process from going on forever and the definition from being circular.

12.1 Recursion

- Methods (Functions) that are defined in terms of themselves in this way are called ***recursive***.
- Here, for example, are two ways to express the definition of factorial,
  - the **first iterative** and
  - the **second recursive**:
    1. factorial(N) = 1 * 2 * 3 * ... * N, where N >= 1
    2. factorial(1) = 1

       factorial(N) = N * factorial(N - 1) if N > 1
- In this case the iterative definition is more familiar and thus easier to understand than the recursive one; however, such is not always the case.

12.1 Recursion

- Consider the definition of Fibonacci numbers below.
- The first and second numbers in the Fibonacci sequence are 1.
- Thereafter, each number is the sum of its two immediate predecessors, as follows:

1 1 2 3 5 8 13 21 34 55 89 144 233 ...

Or in other words:

fibonacci(1) = 1

fibonacci(2) = 1

fibonacci(N) = fibonacci(N - 1) + fibonacci(N - 2) if N > 2

- This is a recursive definition, and it is hard to imagine how one could express it nonrecursively.

12.1 Recursion

- Recursion involves two factors:
  - Some function f(N) is expressed in terms of f(N - 1)and perhaps f(N - 2), and so on.
  - Second, to prevent the definition from being circular, f(1) and perhaps f(2), and so on, are defined explicitly.

12.1 Recursion

Implementing Recursion

- Given a recursive definition of some process, it is usually easy to write a ***recursive method*** that implements it.
- A method is said to be recursive if it calls itself.
- Start with a method that computes factorials.

```
int factorial (int n){
//Precondition n >= 1
  if (n = = 1)
    return 1;
  else
    return n * factorial (n - 1);
}
```

12.1 Recursion

- Tracing a recursive method can help to understand it:

```
factorial(4)
            calls factorial(3)
                              calls factorial(2)
                                                calls factorial(1)
                                                which returns 1
                              which returns 2 * 1       which is 2
            which returns 3 * 2     which is 6
which returns 4 * 6       which is 24
```

12.1 Recursion

- For comparison, here is an **iterative** version of the method.
- As you can see, it is slightly longer and no easier to understand.

```
int factorial (int n){
  int product = 1;
  for (int i = 2; i <= n; i++)
    product = product * i;
  }
  return product;
}
```

12.1 Recursion

- As a second example of recursion, below is a method that calculates Fibonacci numbers:

```
int fibonacci (int n){
   if (n <= 2)
      return 1;
   else
      return fibonacci (n - 1) +
fibonacci (n - 2);
}
```

12.1 Recursion

Guidelines for Writing Recursive Methods

- A recursive method must have a well-defined termination or ***stopping state***.
- For the factorial method, this was expressed in the lines:

```
if (n = = 1)
   return 1;
```

- The ***recursive step***, in which the method calls itself, must eventually lead to the stopping state.
- For the factorial method, the recursive step was expressed in the lines:

```
else
   return n * factorial(n - 1);
```

12.1 Recursion

- Because each invocation of the factorial method is passed a smaller value, eventually the stopping state must be reached.
- Had we accidentally written:

```
else
    return n * factorial(n + 1);
```

- the method would describe an ***infinite recursion***.
- Eventually, the user would notice and terminate the program, or else the Java interpreter would run out memory, and the program would terminate with a ***stack overflow error***.

12.1 Recursion

- Here is a subtler example of a malformed recursive method:

```
int badMethod (int n){
   if (n = = 1)
      return 1;
   else
      return n * badMethod(n - 2);
}
```

- This method works fine if n is odd, but when n is even, the method passes through the stopping state and keeps on going.

12.1 Recursion

Runtime Support for Recursive Methods

- Computers provide the following support at run time for method calls:
  - A large storage area known as a ***call stack*** is created at program startup.
  - When a method is called, an ***activation record*** is added to the top of the call stack.
  - The activation record contains, among other things, space for the parameters passed to the method, the method's local variables, and the value returned by the method.
  - When a method returns, its activation record is removed from the top of the stack.

12.1 Recursion

- An activation record for this method requires cells for the following items:
  - The value of the parameter n
  - The return value of factorial.

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

- Suppose we call factorial(4). A trace of the state of the call stack during calls to factorial down to factorial(1) is shown in Figure 12-1.

12.1 Recursion

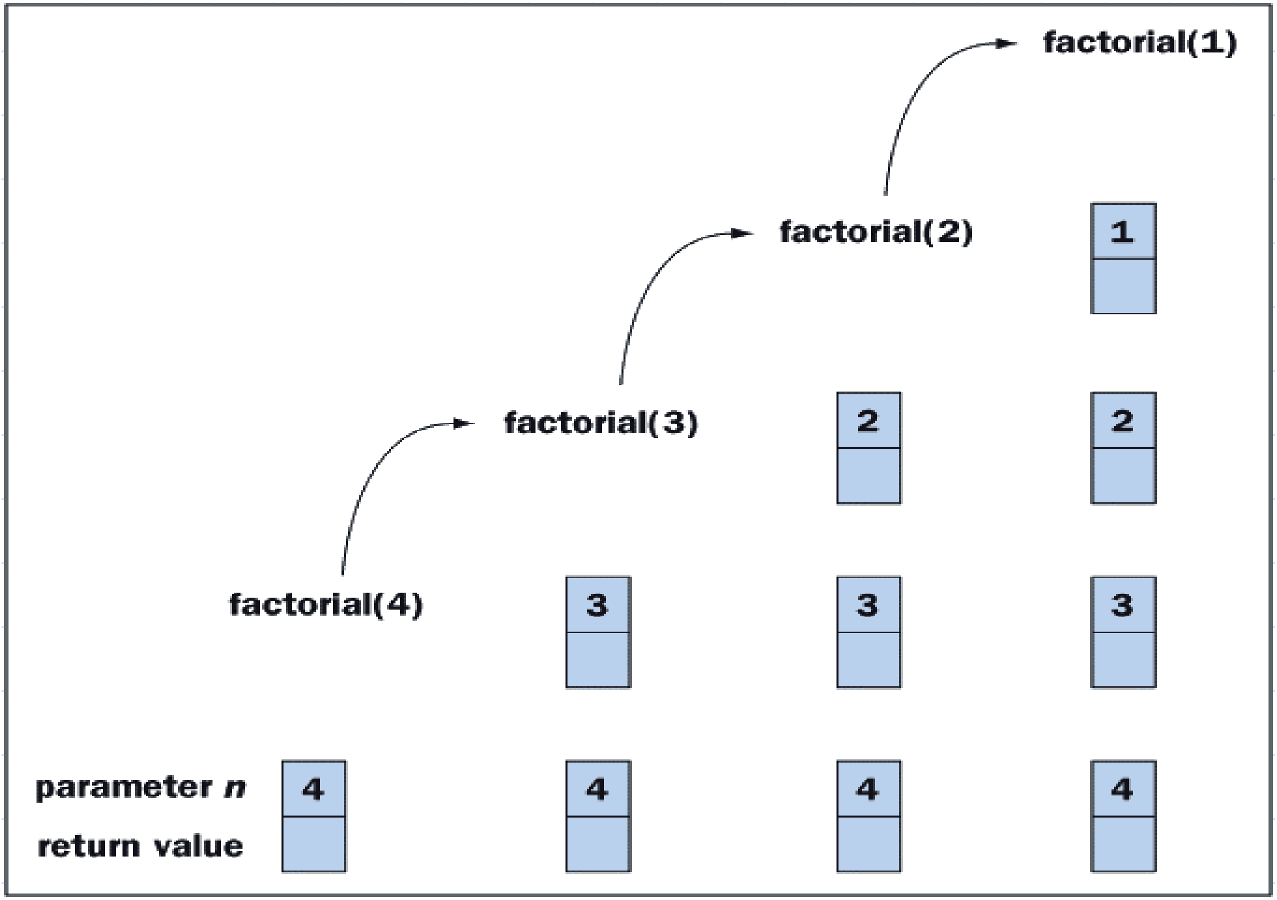


Figure 12-1: Activation records on the call stack during recursive calls to factorial

12.1 Recursion

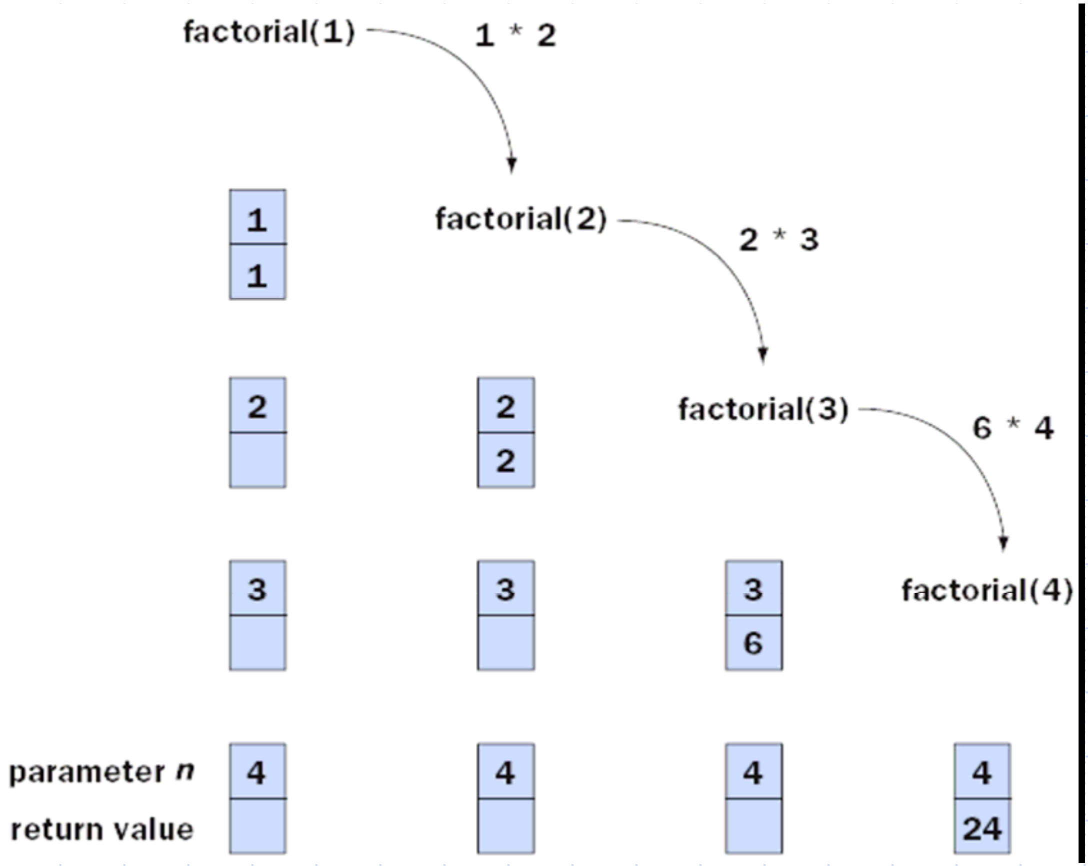


Figure 12-2: Activation records on the call stack during returns from recursive calls to factorial

12.1 Recursion

When to Use Recursion

- Recursion can always be used in place of iteration, and vice versa.
- Recursion involves a method repeatedly calling itself.
- Executing a method call and the corresponding return statement usually takes longer than incrementing and testing a loop control variable.

12.1 Recursion

- A method call ties up some memory that is not freed until the method completes its task.
- Naïve programmers often state these facts as an argument against ever using recursion.
- However, there are many situations in which recursion provides the clearest, shortest, and most elegant solution to a programming task.

Tail-recursive Algorithms

- **Tail-recursive** algorithms perform no work after the recursive call.
  - Some compilers can optimize the compiled code so that no extra stack memory is required.

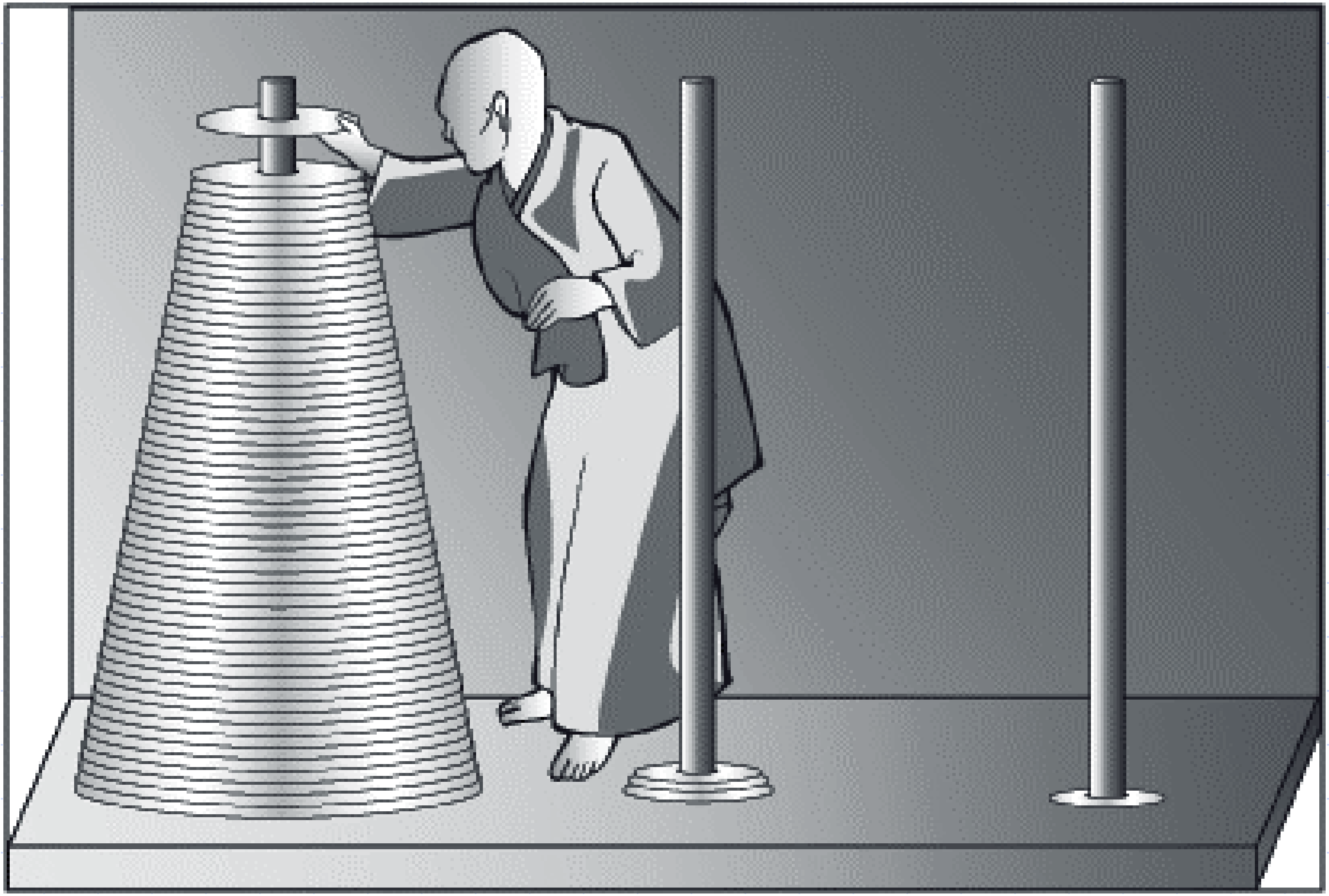
```
int tailRecursiveFactorial (int n, int result){
    if (n == 1)
        return result;
    else
        return tailRecursiveFactorial (n - 1, n * result);
}
```

12.1 Recursion

TOWERS OF HANOI

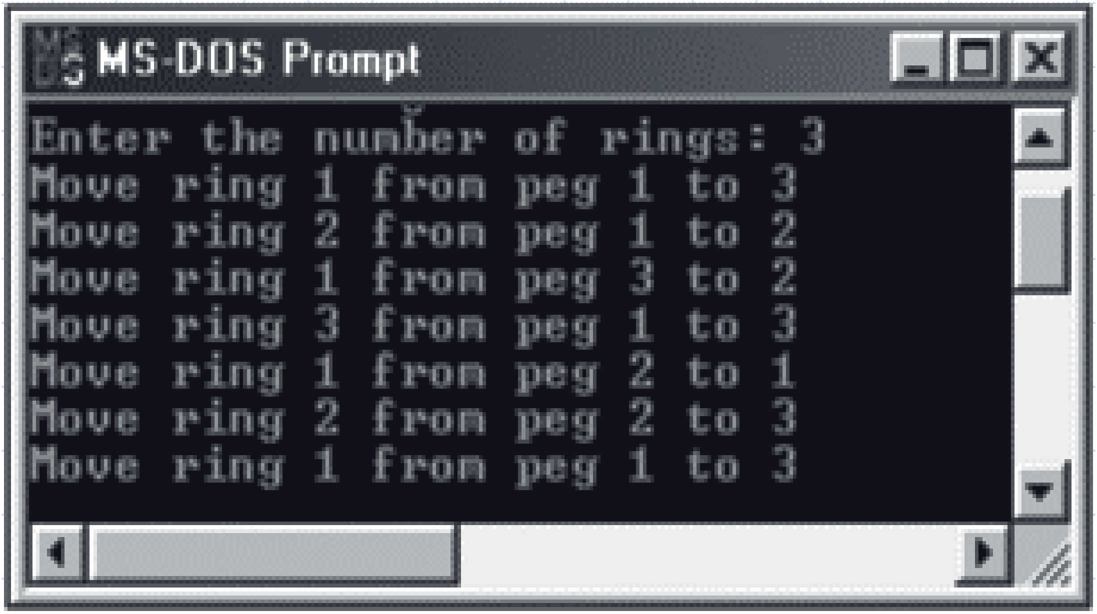
- Many centuries ago in the city of Hanoi, the monks in a certain monastery attempted to solve a puzzle.
- Sixty-four rings of increasing size had been placed on a vertical wooden peg (Figure 12-3).
- Beside it were two other pegs, and the monks were attempting to move all the rings from the first to the third peg - subject to two constraints:
    - only one ring could be moved at a time
    - a ring could be moved to any peg, provided it was not placed on top of a smaller ring.

12.1 Recursion



12.1 Recursion

- Figure 12-4 shows the result of running the program with three rings.
- In the output, the rings are numbered from smallest (1) to largest (3).



12.1 Recursion

- The program uses a recursive method called move.
- The first time this method is called, it is asked to move all N rings from peg 1 to peg 3.
- The method then proceeds by calling itself to move the top N - 1 rings to peg 2, prints a message to move the largest ring from peg 1 to peg 3, and finally calls itself again to move the N - 1 rings from peg 2 to peg 3.

Famous Recursive Problems

[TowersOfHanoi.java]

[ManyQueens.java]

12.2 Complexity Analysis

- Examining the effect on the method of increasing the quantity of data processed is called ***complexity analysis***.

Sum Methods

- The `Sum` method processes an array whose size can be varied.
- To determine the method's execution time, beside each statement we place a symbol (t1, t2, etc.) that indicates the time needed to execute the statement.
- Because we have no way of knowing what these times really are, we can do no better.

12.2 Complexity Analysis

```
int sum (int[] a){
   int i, result;
   result = 0;                           // Assignment: time = t1
   for (i = 0; i < a.length; i++)
   {  // Overhead for going once around the loop: time = t2
      result += a[i];                    // Assignment: time = t3
   }
   return result;                        // Return: time = t4
}
```

12.2 Complexity Analysis

- Adding these times together and remembering that the method goes around the loop n times, where *n* represents the array's size, yields:

```
executionTime
          = t1 + n * (t2 + t3) + t4
          = k1 + n * k2
```

where k1 and k2 are method-dependent constants

$\approx$ `n * k2`

for large values of *n*

12.2 Complexity Analysis

- Thus, the execution time is linearly dependent on the array's length, and as the array's length increases, the contribution of k1 becomes negligible.
- Consequently, we can say with reasonable accuracy that doubling the length of the array doubles the execution time of the method.
- Computer scientists express this linear relationship between the array's length and execution time using ***big-O notation***:

```
executionTime = O(n).
```

12.2 Complexity Analysis

- Phrased differently, the execution time is of order *n*.
- Observe that from the perspective of big-O notation, we make no distinction between a method whose execution time is:

  `1000000 + 1000000*n`

  and one whose execution time is

  `n / 1000000`

- although from a practical perspective the difference is enormous.

12.2 Complexity Analysis

- Complexity analysis can also be applied to recursive methods.
- Here is a recursive version of the sum method. It too is O(*n*).

```
int sum (int[] a, int i){
   if (i >= a.length)                 // Comparison: t1
      return 0;                       // Return: t2
   else
      return a[i] + sum (a, i + 1);  // Call and return: t3
}
```

12.2 Complexity Analysis

- The method is called initially with i = 0.
- A single activation of the method takes time:

```
t1 + t2      if  i >= a.length
```

and

```
t1 + t3      if  i < a.length.
```

- The first case occurs once and the second case occurs the a.`length` times that the method calls itself recursively.
- Thus, if n equals a.`length`, then:

```
executionTime
     = t1 + t2 + n * (t1 + t3)
     = k1 + n * k2
```

where k1 and k2 are method-dependent constants

```
     = O(n)
```

12.2 Complexity Analysis

- Following is a linear search method form Lesson 11:

```
int search (int[] a, int searchValue){
   for (i = 0; i < a.length; i++)    // Loop overhead: t1
      if (a[i] == searchValue)       // Comparison: t2
         return i;                   // Return point 1: t3
   return location;                  // Return point 2: t4
}
```

12.2 Complexity Analysis

- The analysis of the linear search method is slightly more complex than that of the sum method.
- Each time through the loop, a comparison is made.
- If and when a match is found, the method returns from the loop with the search value's index.

```
executionTime
     = (n / 2) * (t1 + t2) + t3
     = n * k1 + k2
```

where `k1` and `k2` are method-dependent constants.

```
     = O(n)
```

12.2 Complexity Analysis

- Now let us look at a method that processes a two-dimensional array:

```
int[] sumRows (int[][] a){
   int i, j;
   int[] rowSum = new int[a.length];     // Instantiation: t1
   for (i = 0; i < a.length; i++){       // Loop overhead: t2
      for (j = 0; j < a[i].length; j++){ // Loop overhead: t3
         rowSum[i] += a[i][j];           // Assignment: t4
      }
   }
   return rowSum;                        // Return: t5
}
```

12.2 Complexity Analysis

- Let n represent the total number of elements in the array and r the number of rows.
- For the sake of simplicity, we assume that each row has the same number of elements, say, c.
- The execution time can be written as:

```
executionTime
     = t1 + r * (t2 + c * (t3 + t4)) + t5
     = (k1 + n * k2) + (n/c) * t2 + n * (t3 + t4) + t5
```

where r = n/c

```
     = (k1 + n * k2) + n * (t2 / c + t3 + t4) + t5
     = k2 + n * k3
```

where k1, k2, k3, and k4 are constants

```
     = O(n)
```

12.2 Complexity Analysis

An O(n^2) Method

- Not all array processing methods are O(n), as an examination of the bubbleSort method reveals.
- This one does not track whether an exchange was made in the nested loop, so there is no early exit.

12.2 Complexity Analysis

```
void bubbleSort(int[] a){
   int k = 0;

   // Make n - 1 passes through array

   while (k < a.length() - 1){                // Loop overhead: t1
      k++;
      for (int j = 0; j < a.length() - k; j++)   // Loop overhead: t2
         if (a[j] > a[j + 1])                 // Comparison: t3
            swap(a, j, j + 1);                // Assignments: t4
   }
}
```

12.2 Complexity Analysis

- The outer loop of the sort method executes n - 1 times, where n is the length of the array.
- Each time the inner loop is activated, it iterates a different number of times.
- On the first activation it iterates n - 1 times, on the second n - 2, and so on.
- On the last activation it iterates once.
- The average number of iterations is n / 2.
- On some iterations, elements a[i] and a[j] are interchanged in time t4, and on other iterations, they are not.
- So on the average iteration, let's say time t5 is spent doing an interchange.

12.2 Complexity Analysis

- The execution time of the method can now be expressed as:

```
executionTime
     = t1 + (n - 1) * (t1 + (n / 2) * (t2 + t3 + t5))
     = t1 + n * t1 – t1 + (n * n / 2) * (t2 + t3 + t4) –
                          (n / 2) * (t2 + t3 + t4)
     = k1 + n * k2 + n * n * k3
     ≈ n * n * k3                    for large values of n
     = O(n²)
```

12.2 Complexity Analysis

Common Big-O Values

- Table 12-1 lists some other common big-O values together with their names.

BIG-O VALUE	NAME
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	$n \log n$
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential

12.2 Complexity Analysis

- The values in Table 12-1 are listed from "best" to "worst."
- Given two methods that perform the same task, but in different ways, we tend to prefer the one that is O(n) over the one that is O(n2).
- Suppose that the exact run time of two methods is:

```
10,000 + 400n          // method 1
```

and

$$10,000 + n^2 \qquad \text{// method 2}$$

12.2 Complexity Analysis

- For small values of *n*, method 2 is faster than method 1
- For all values of *n* larger than a certain threshold, method 1 is faster.
- The threshold in this example is 400.
- So if you know ahead of time that *n* will always be less than 400, you are advised to use method 2, but if *n* will have a large range of values, method 1 is superior.

12.2 Complexity Analysis

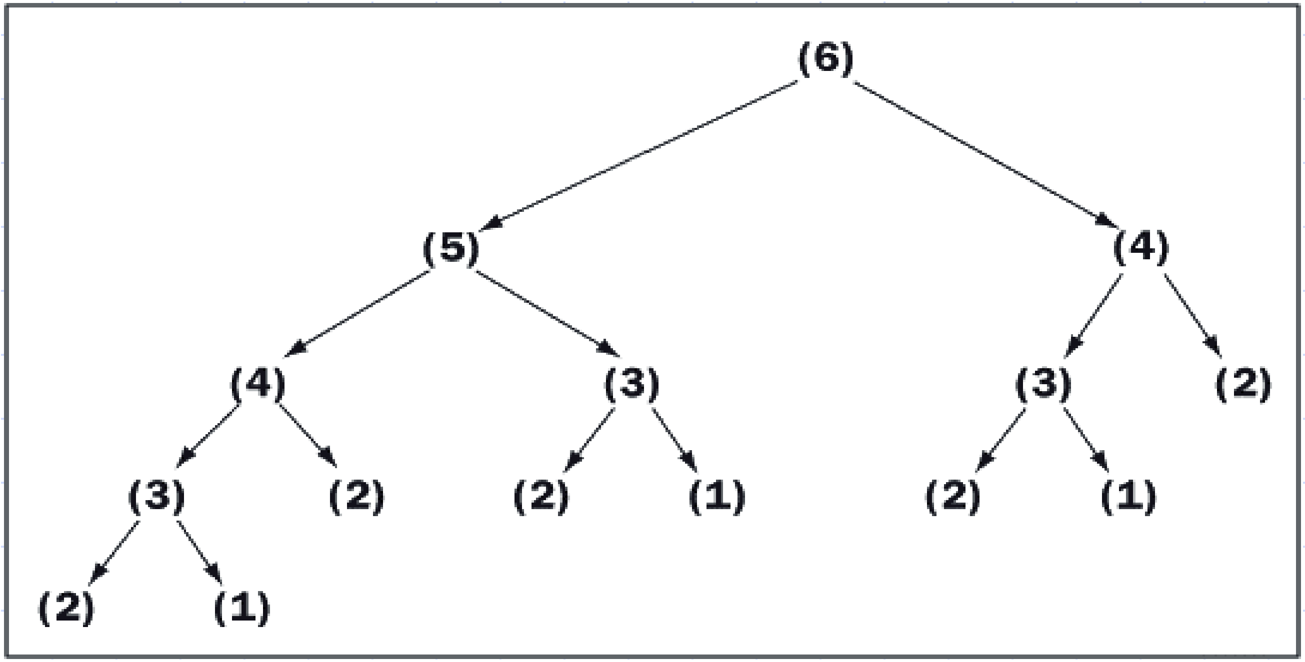
- To get a feeling for how the common big-O values vary with *n*, consider Table 12-2.
- We use base 10 logarithms.
- This table vividly demonstrates that a method might be useful for small values of *n*, but totally worthless for large values.

n	1	LOG n	n	n LOG n	n^2	n^3	2^n
10	1	1	10	10	100	1,000	1,024
100	1	2	100	200	10,000	1,000,000	≈ 1.3 e30
1,000	1	3	1,000	3,000	1,000,000	1,000,000,000	≈ 1.1 e301

12.2 Complexity Analysis

An O(r^n) Method

- Figure 12-7 shows the calls involved when we use the recursive method to compute the sixth Fibonacci number.



12.2 Complexity Analysis

- Table 12-3 shows the number of calls as a function of *n*.

n	CALLS NEEDED TO COMPUTE *n*TH FIBONACCI NUMBER
2	1
4	5
8	41
16	1,973
32	**4,356,617**

12.2 Complexity Analysis

- Three cases of complexity are typically analyzed for an algorithm:
  - **Best case:** When does an algorithm do the least work, and with what complexity?
  - **Worst case:** When does an algorithm do the most work, and with what complexity?
  - **Average case:** When does an algorithm do a typical amount of work, and with what complexity?

12.2 Complexity Analysis

- Examples:
  - A summation of array values has a best, worst, and typical complexity of O(n).
    - Always visits every array element
  - A linear search:
    - Best case of O(1) – element found on first iteration
    - Worst case of O(n) – element found on last iteration
    - Average case of O(n/2)

12.2 Complexity Analysis

- Bubble sort complexity:
  - Best case is O(n) – when array already sorted
  - Worst case is O(n^2) – array sorted in reverse order
  - Average case is closer to O(n^2).

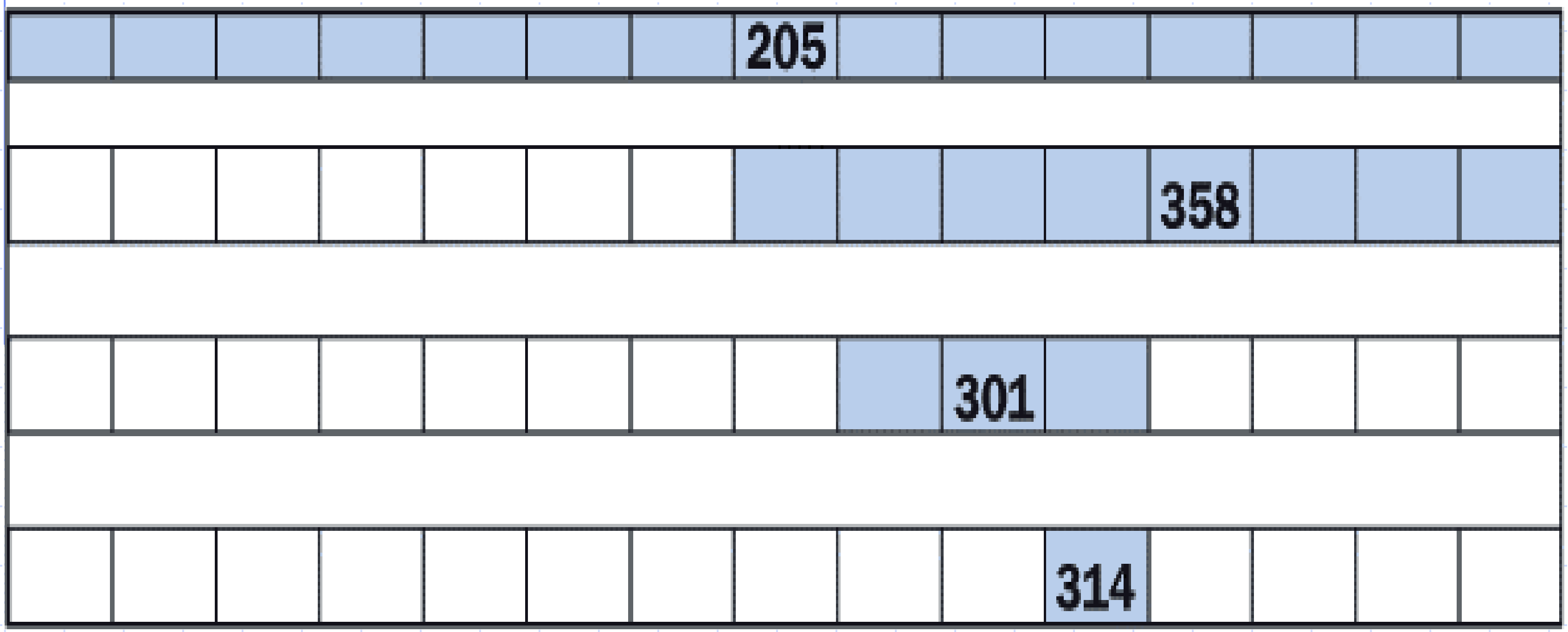
12.3 Recursive Binary Search

- If we know in advance that a list of numbers is in ascending order, we can quickly zero in on the search value or determine that it is absent using the ***binary search algorithm***.
- We shall show that this algorithm is O(log *n*).

12.3 Binary Search

- Figure 12-8 is an illustration of the binary search algorithm.
- We are looking for the number 320.
- At each step we highlight the sublist that might still contain 320.
- At each step, all the numbers are invisible except the one in the middle of the sublist, which is the one that we are comparing to 320.

12.3 Binary Search



12.3 Binary Search

- After only four steps, we have determined that 320 is not in the list.
- Had the search value been 205, 358, 301, or 314 we would have located it in four or fewer steps.
- The binary search algorithm is guaranteed to search a list of 15 sorted elements in a maximum of four steps.
- Incidentally, the list with all the numbers visible looks like Figure 12-9.

15	36	87	95	100	110	194	205	297	301	314	358	451	467	486

12.3 Binary Search

- Table 12-4 shows the relationship between a list's length and the maximum number of steps needed to search the list.
- To obtain the numbers in the second column, add 1 to the larger numbers in the first column and take the logarithm base 2.
- A method that implements a binary search is O(log *n*).

12.3 Binary Search

LENGTH OF LIST	MAXIMUM NUMBER OF STEPS NEEDED
1	1
2 to 3	2
4 to 7	3
8 to 15	4
16 to 31	5
32 to 63	6
64 to 127	7
128 to 255	8
256 to 511	9
512 to 1023	10
1024 to 2047	11
2^n to $2^{n+1} - 1$	$n + 1$

12.3 Binary Search

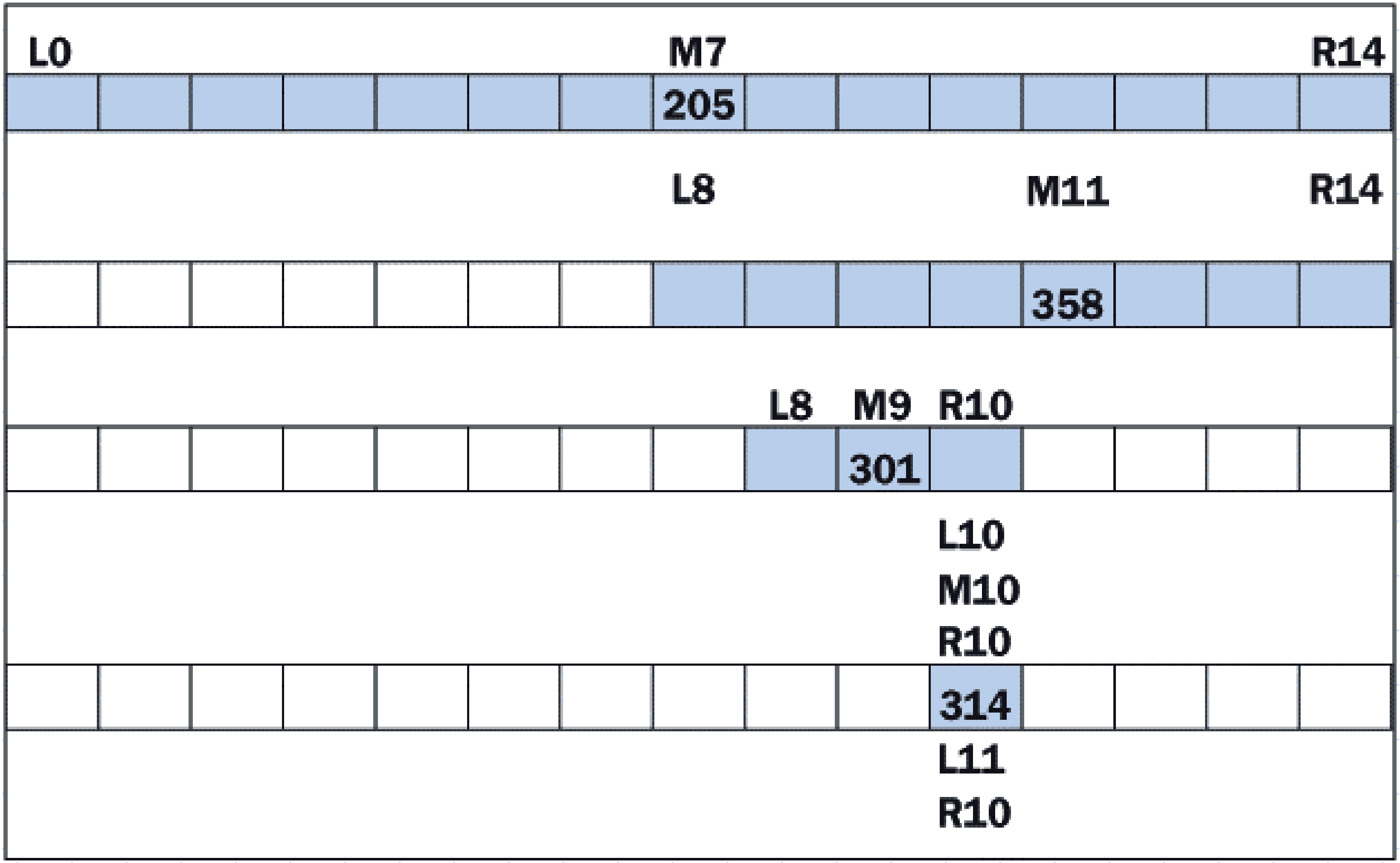
- We now present two versions of the binary search algorithm, one iterative and one recursive, and both O(log *n*).

```
// Iterative binary search of an ascending array
int binarySearch (int[] a, int searchValue){
   int left = 0;                          // Establish the initial
   int right = a.length – 1;              // endpoints of the array
   while (left <= right){                 // Loop until the endpoints cross
      int midpoint = (left + right) / 2;  // Compute the current midpoint
      if (a[midpoint] == searchValue)     // Target found; return its index
         return midpoint;
      else if (a[midpoint] < searchValue) // Target to right of midpoint
         left = midpoint + 1;
      else                                // Target to left of midpoint
         right = midpoint – 1;
   }
   return -1;                             // Target not found
}
```

12.3 Binary Search

- Figure 12-10 illustrates an iterative search for 320 in the list of 15 elements.
- L, M, and R are abbreviations for left, midpoint, and right.
- At each step, the figure shows how these variables change.
- Because 320 is absent from the list, eventually (left > right) and the method returns –1.

12.3 Binary Search



12.3 Binary Search

- Now for the recursive version of the algorithm:

```
// Recursive binary search of an ascending array
int binarySearch (int[] a, int searchValue, int left, int right){
   if (left > right)
      return –1;
   else{
      int midpoint = (left + right) / 2;
      if (a[midpoint] == searchValue)
         return midpoint;
      else if (a[midpoint] < searchValue)
         return binarySearch (a, searchValue, midpoint + 1, right);
      else
         return binarySearch (a, searchValue, left, midpoint - 1);
   }
}
```

12.3 Binary Search

- The two versions are similar, and they use the variables left, midpoint, and right in the same way.
- They, of course, differ in that one uses a loop and the other uses recursion.
- We conclude the discussion by showing how the two methods are called:

```
int[] a =
{15,36,87,95,100,110,194,205,297,301,314,358,451,467,486};
int x = 320;
int location;

location = binarySearch (a, x);                  // Iterative version
location = binarySearch (a, x, 0, a.length - 1); // Recursive version
```

12.4 Quicksort

- There are also several much better algorithms that are O(n log n).
- ***Quicksort*** is one of the simplest.
- The general idea behind quicksort is this:
  - Break an array into two parts
  - Move elements around so that all the larger values are in one end and all the smaller values are in the other.
  - Each of the two parts is then subdivided in the same manner, and so on until the subparts contain only a single value, at which point the array is sorted.

12.4 Quicksort

- To illustrate the process, suppose an unsorted array, called a, looks like Figure 12-11.

5	12	3	11	2	7	20	10	8	4	9

12.4 Quicksort

Phase 1

1. If the length of the array is less than 2, then done.
2. Locate the value in the middle of the array and call it the pivot. The pivot is 7 in this example (Fig 12-12).

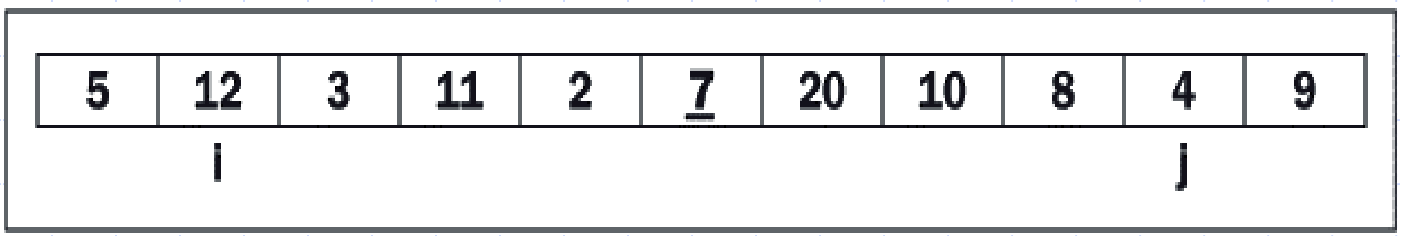
5	12	3	11	2	7	20	10	8	4	9

3. Tag the elements at the left and right ends of the array as i and j, respectively (Fig 12-13).

5	12	3	11	2	7	20	10	8	4	9
i										j

12.4 Quicksort

4. While a[i] < pivot value, increment i.
   While a[j] >= pivot value, decrement j:
   (Fig 12-14)



12.4 Quicksort

5. If i > j then

   end the phase

   else

   interchange a[i] and a[j]:

   (Fig 12-15)

5	4	3	11	2	7	20	10	8	12	9
	i								j	

12.4 Quicksort

6. Increment i and decrement j.
   If i > j then end the phase:
   (Fig 12-16)

5	4	3	11	2	<u>7</u>	20	10	8	12	9
		i						j		

12.4 Quicksort

7. Repeat step 4, i.e.,

   While a[i] < pivot value, increment i

   While a[j] >= pivot value, decrement j:

   (Fig 12-17)

5	4	3	11	2	7	20	10	8	12	9
			i		j					

12.4 Quicksort

8. Repeat step 5, i.e.,

   If i > j then

   end the phase

   else

   interchange a[i] and a[j]:

   (Fig 12-18)

5	4	3	7	2	11	20	10	8	12	9
			i		j					

12.4 Quicksort

9. Repeat step 6, i.e.,
   Increment i and decrement j.
   If i < j then end the phase:
   (Fig 12-19)

5	4	3	7	2	11	20	10	8	12	9

ij

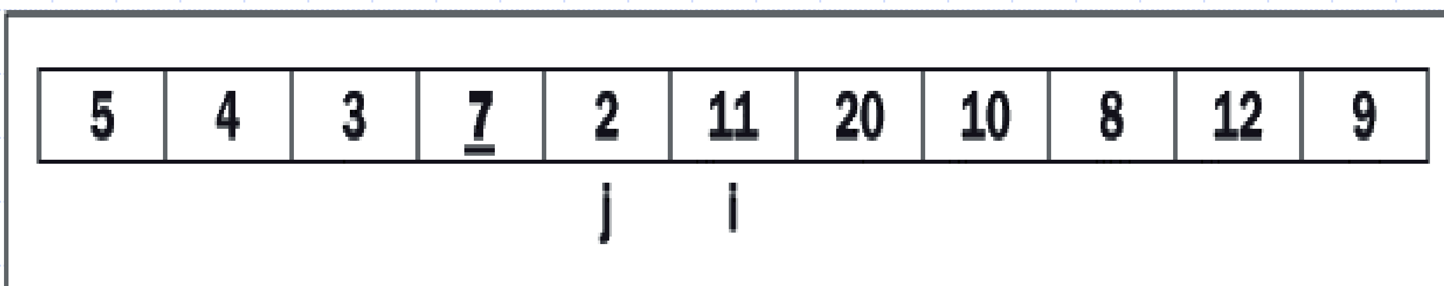
12.4 Quicksort

10. Repeat step 4, i.e.,

    While a[i] < pivot value, increment i

    While a[j] >= pivot value, decrement j:

    (Fig 12-20)



12.4 Quicksort

11. Repeat step 5, i.e.,

    If i > j then

    end the phase

    else

    interchange a[i] and a[j].

12.4 Quicksort

- This ends the phase.
- Split the array into the two subarrays a[0..j] and a[i..10].
- For clarity, the left subarray is shaded.
- Notice that all the elements in the left subarray are less than or equal to the pivot, and those in the right are greater than or equal.

(Fig 12-21)

5	4	3	7	2	11	20	10	8	12	9

12.4 Quicksort

Phase 2 and Onward

- Reapply the process to the left and right subarrays and then divide each subarray in two and so on until the subarrays have lengths of at most one.

12.4 Quicksort

Complexity Analysis

- During phase 1, i and j moved toward each other.
- At each move, either an array element is compared to the pivot or an interchange takes place.
- As soon as i and j pass each other, the process stops.
- Thus, the amount of work during phase 1 is proportional to n, the array's length.

12.4 Quicksort

- The amount of work in phase 2 is proportional to the left subarray's length plus the right subarray's length, which together yield n.
- When these subarrays are divided, there are four pieces whose combined length is n, so the combined work is proportional to n yet again.
- At successive phases, the array is divided into more pieces, but the total work remains proportional to n.

12.4 Quicksort

- To complete the analysis, we need to determine how many times the arrays are subdivided.
- When we divide an array in half repeatedly, we arrive at a single element in about $\log_2 n$ steps.
- Thus the algorithm is $O(n \log n)$ in the best case.
- In the worst case, the algorithm is $O(n^2)$.

12.4 Quicksort

Implementation

- The quicksort algorithm can be coded using either an iterative or a recursive approach.
- The iterative approach also requires a data structure called a ***stack***.
- The following example implements the quicksort algorithm recursively:

12.4 Quicksort

```
void quickSort (int[] a, int left, int right){
 //Recursive Version
   if (left >= right) return;

   int i = left;
   int j = right;
   int pivotValue = a[(left + right) / 2];
   while (i < j){
      while (a[i] < pivotValue) i++;
      while (pivotValue < a[j]) j--;
      if (i <= j){
         int temp = a[i];
         a[i] = a[j];
         a[j] = temp;
         i++;
         j--;
      }
   }
   quickSort (a, left, j);
   quickSort (a, i, right);
}
```

12.5 Merge Sort

- Merge Sort uses a recursive, divide and conquer strategy to break the $O(n^2)$ barrier.
- Here is the outline of the algorithm:
  - Compute the middle position of an array and recursively sort its left and right sub-arrays (divide and conquer).
  - Merge the two sorted sub-arrays back into a single sorted array.

12.5 Merge Sort

- Stop the process when the sub-arrays can no longer be subdivided.
- This strategy uses three Java Methods:
  - **mergeSort**: the public method called by clients
  - **mergeSortHelper**: a private helper method that hides the extra parameter required by the recursive calls.
  - **merge**: a private method that implements the merging process.

12.5 Merge Sort

- The merging process uses an extra array, called **copyBuffer**.
- **copyBuffer** is declared and initialized in **mergeSort** and passed to **mergeSortHelper** and merge.

mergeSort Code

mergeSort is called by the client, creates the temporary array, **copyBuffer**, and sends the necessary parameters to **mergeSortHelper**.
The parameters are the original array "**a**", the temporary array, **copyBuffer** and the starting and ending positions of the original array, here 0 and a.length-1 (or 0 and logicalSize-1)

```
void mergeSort(int[] a){
   // a            array being sorted
   // copyBuffer   temp space needed during merge

   int[] copyBuffer = new int[a.length];
   mergeSortHelper(a, copyBuffer, 0, a.length - 1);
}
```

mergeSortHelper Code

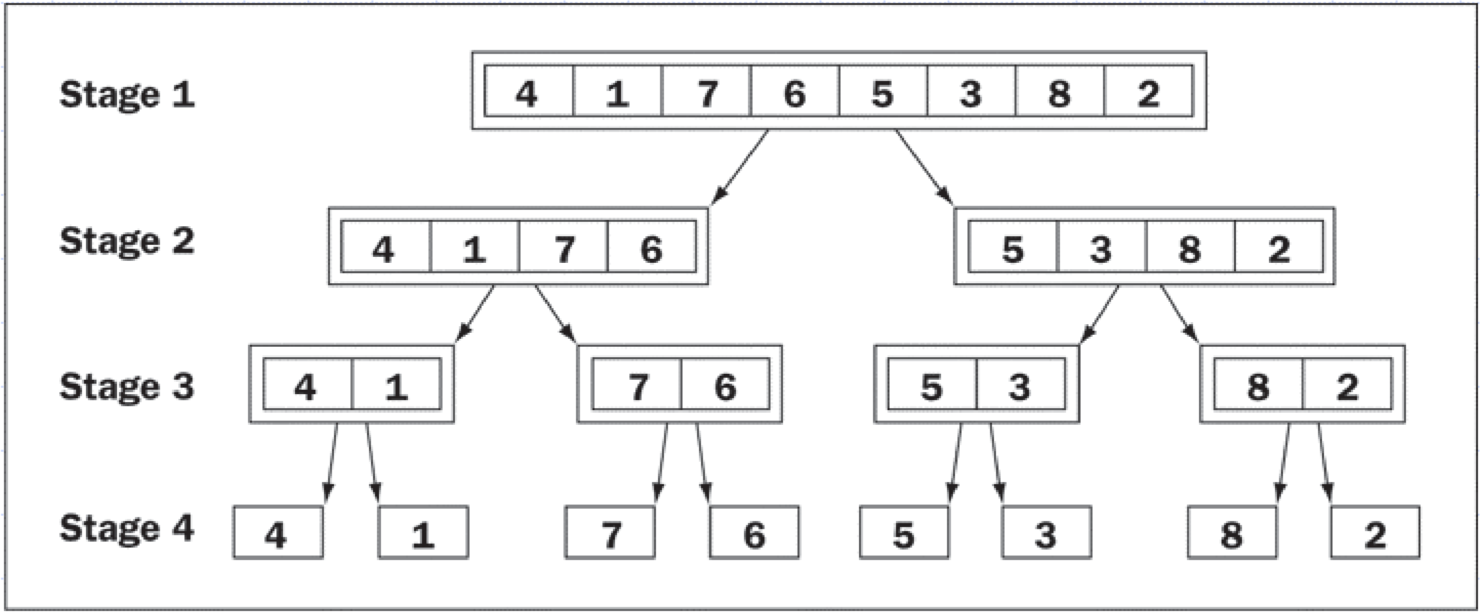
mergeSortHelper computes the midpoint of the sub-array, recursively sorts the portions below and above the mid-point, and calls **merge** to merge the results.

```
void mergeSortHelper(int[] a, int[] copyBuffer,
                     int low, int high){
   // a              array being sorted
   // copyBuffer     temp space needed during merge
   // low, high      bounds of subarray
   // middle         midpoint of subarray

   if (low < high){
      int middle = (low + high) / 2;
      mergeSortHelper(a, copyBuffer, low, middle);
      mergeSortHelper(a, copyBuffer, middle + 1, high);
      merge(a, copyBuffer, low, middle, high);
   }
}
```

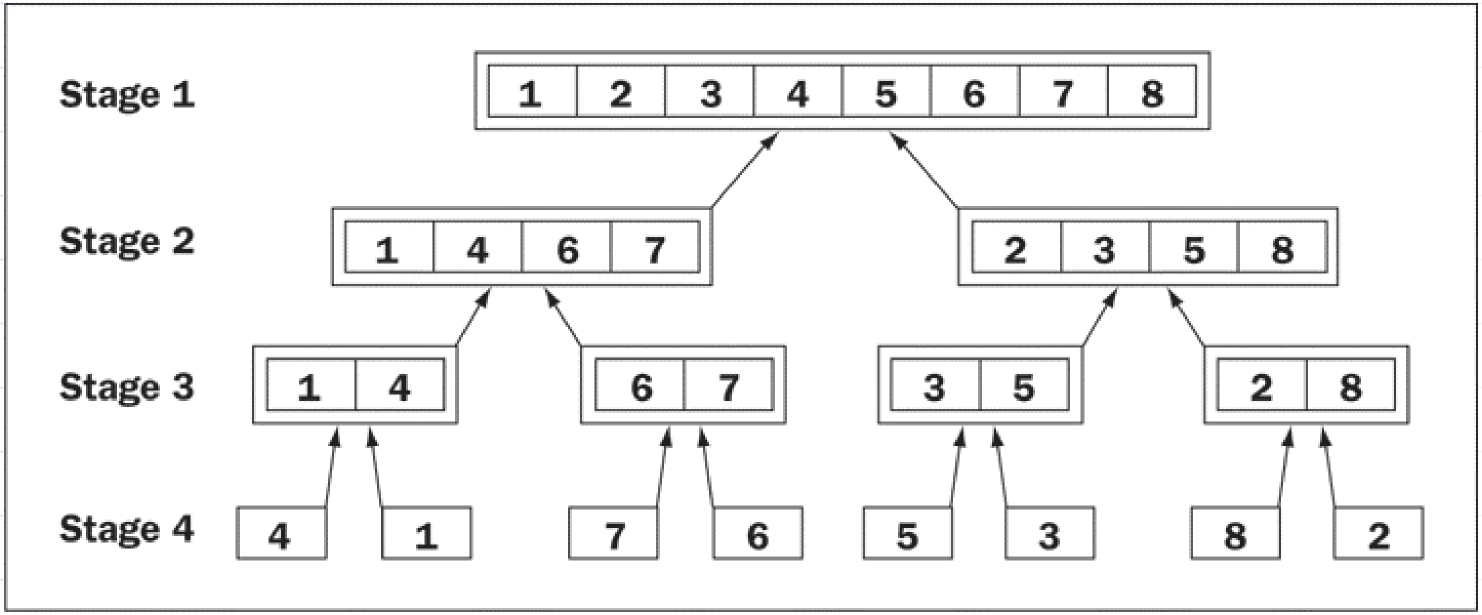
12.5 Merge Sort

- Fig. 12-22 shows the sub-arrays generated during recursive calls to **mergeSortHelper**, starting from an array of 8 items.



12.5 Merge Sort

- Fig. 12-23 traces the process of merging the sub-arrays generated in the previous figure.



merge Code

- Here is the code for the **merge** method:

```
void merge(int[] a, int[] copyBuffer,
           int low, int middle, int high){
   // a             array that is being sorted
   // copyBuffer    temp space needed during the merge process
   // low           beginning of first sorted subarray
   // middle        end of first sorted subarray
   // middle + 1    beginning of second sorted subarray
   // high          end of second sorted subarray

   // Initialize i1 and i2 to the first items in each subarray
   int i1 = low, i2 = middle + 1;
  // Interleave items from the subarrays into the copyBuffer in such a
  // way that order is maintained.
  for (int i = low; i <= high; i++){
     if (i1 > middle)
        copyBuffer[i] = a[i2++];     // First subarray exhausted
     else if (i2 > high)
        copyBuffer[i] = a[i1++];     // Second subarray exhausted
     else if (a[i1] < a[i2])
        copyBuffer[i] = a[i1++];     // Item in first subarray is less
     else
        copyBuffer[i] = a[i2++];     // Item in second subarray is less
  }

  for (int i = low; i <= high; i++) // Copy sorted items back into
     a[i] = copyBuffer[i];          // proper position in a
}
```

12.5 Merge Sort

- The merge method combines two sorted sub-arrays into a larger sorted sub-array.
  - The first sub-array lies between **low** and **middle**
  - The second sub-array is between **middle +1** and **high**.

12.5 Merge Sort

- The process consists of three steps:
  - Set up index pointers to the first items in each sub-array. These are positions **low** and **middle +1.**
  - Starting with the first item in each sub-array, repeatedly compare items. Copy the smaller item from its sub-array to the copy buffer and advance to the next item in the sub-array. Repeat until all items have been copied from both sub-arrays. If the end of one sub-array is reached before the other's, finish by copying the remaining items from the other sub-array.
  - Copy the portion of **copyBuffer** between **low** and **high** back to the corresponding positions in the array "**a**".

12.5 Merge Sort

- Complexity analysis:
  - Execution time dominated by the two `for` loops in the `merge` method
  - Each loops (`high + low + 1`) times
    - For each recursive stage, O(`high + low`) or $O(n)$
  - Number of stages is $O(\log n)$, so overall complexity is $O(n \log n)$.

Case Study

ComparingSortAlgorithms.java

ComparingSortAlgorithms.txt

Design, Testing, and Debugging Hints

- When designing a recursive method, ensure:
  - A well-defined stopping state
  - A recursive step that changes the size of the data so the stopping state will eventually be reached
- Recursive methods can be easier to write correctly than equivalent iterative methods.
- More efficient code is often more complex.

Summary

- A recursive method is a method that calls itself to solve a problem.
- Recursive solutions have one or more base cases or termination conditions that return a simple value or `void`.
- Recursive solutions have one or more recursive steps that receive a smaller instance of the problem as a parameter.

Summary (cont.)

- Some recursive methods combine the results of earlier calls to produce a complete solution.
- Run-time behavior of an algorithm can be expressed in terms of big-O notation.
- Big-O notation shows approximately how the work of the algorithm grows as a function of its problem size.

Summary (cont.)

- There are different orders of complexity such as constant, linear, quadratic, and exponential.
- Through complexity analysis and clever design, the complexity of an algorithm can be reduced to increase efficiency.
- Quicksort uses recursion and can perform much more efficiently than selection sort, bubble sort, or insertion sort.