

Chapter 12

Recursion, Complexity, and Searching and Sorting

Fundamentals of Java



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.

Objectives (cont.)

- Understand how a computer executes a recursive method.
- Perform a simple complexity analysis of an algorithm using big-O notation.
- Recognize some typical orders of complexity.
- Understand the behavior of a complex sort algorithm, such as the quicksort.

Vocabulary

- Activation record
- Big-O notation
- Binary search algorithm
- Call stack
- Complexity analysis

Vocabulary (cont.)

- Infinite recursion
- Iterative process
- Merge sort
- Quicksort
- Recursive method

Vocabulary (cont.)

- Recursive step
- Stack
- Stack overflow error
- Stopping state
- Tail-recursive

Recursion

- **Recursive** functions are defined in terms of themselves.

- $sum(n) = n + sum(n-1)$ if $n > 1$

- Example:

```
sum(4) = 4 + sum(3)
        = 4 + 3 + sum(2)
        = 4 + 3 + 2 + sum(1)
        = 4 + 3 + 2 + 1
```

- Other functions that could be defined recursively include factorial and Fibonacci sequence.

Recursion (cont.)

- A method is recursive if it calls itself.
- Factorial example:

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

- Fibonacci sequence example:

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

Recursion (cont.)

- 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
```

Recursion (cont.)

- Guidelines for implementing recursive methods:
 - Must have a well-defined **stopping state**
 - The **recursive step** must lead to the stopping state.
 - The step in which the method calls itself
 - If either condition is not met, it will result in **infinite recursion**.
 - Creates a **stack overflow error** and program crashes

Recursion (cont.)

- Run-time support for recursive methods:
 - A ***call stack*** (large storage area) is created at program startup.
 - When a method is called, an ***activation record*** is added to the top of the call stack.
 - Contains space for the method parameters, the method's local variables, and the return value
 - When a method returns, its activation record is removed from the top of the stack.

Recursion (cont.)

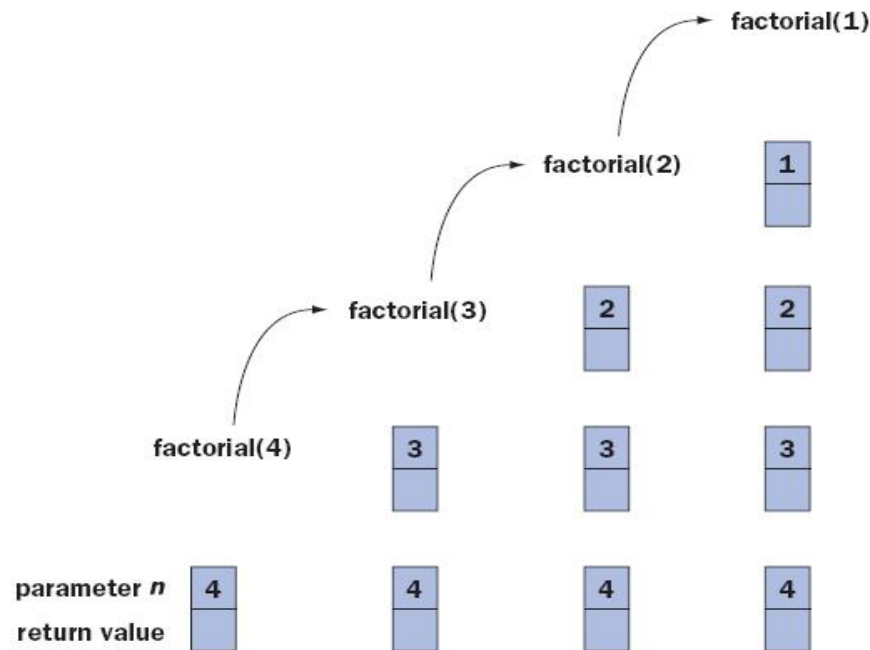


Figure 12-1: Activation records on the call stack during recursive calls to factorial

Recursion (cont.)

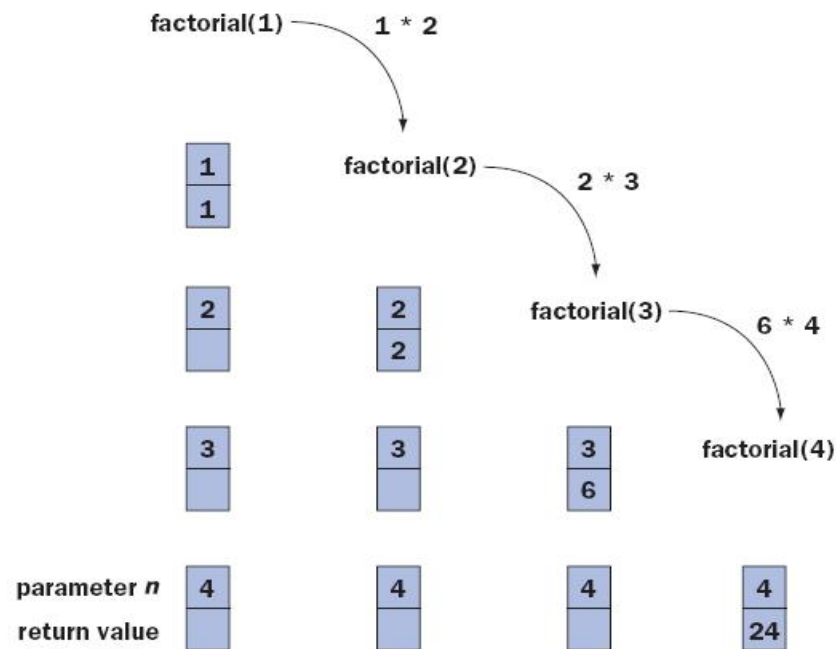


Figure 12-2: Activation records on the call stack during returns from recursive calls to factorial

Recursion (cont.)

- Recursion can always be used in place of iteration, and vice-versa.
 - Executing a method call and the corresponding return statement usually takes longer than incrementing and testing a loop control variable.
 - Method calls tie up memory that is not freed until the methods complete their tasks.
 - However, recursion can be much more elegant than iteration.

Recursion (cont.)

- **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);
}
```

Complexity Analysis

- What is the effect on the method of increasing the quantity of data processed?
 - Does execution time increase exponentially or linearly with amount of data being processed?
- Example:

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

Complexity Analysis (cont.)

- If array's size is n , the execution time is determined as follows:

```
executionTime
  = t1 + n * (t2 + t3) + t4
  = k1 + n * k2           where k1 and k2 are method-dependent constants
  = n * k2                for large values of n
```

- Execution time can be expressed using **Big-O notation**.
 - Previous example is **$O(n)$**
 - Execution time is on the *order of* n .
 - Linear time

Complexity Analysis (cont.)

- Another $O(n)$ method:

```
int search (int[] a, int searchValue){  
    for (int 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  
}
```

```
executionTime  
= (n / 2) * (t1 + t2) + t3  
= n * k1 + k2           where k1 and k2 are method-dependent constants.  
=  $O(n)$ 
```

Complexity Analysis (cont.)

- An $O(n^2)$ method:

```
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
    }
}
```

```
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)
```

Complexity Analysis (cont.)

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

Table 12-1: Names of some common big-O values

Complexity Analysis (cont.)

- $O(1)$ is best complexity.
- Exponential complexity is generally bad.

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	$\approx 1.3 \text{ e}30$
1,000	1	3	1,000	3,000	1,000,000	1,000,000,000	$\approx 1.1 \text{ e}301$

Table 12-2: How big-O values vary depending on n

Complexity Analysis (cont.)

- Recursive Fibonacci sequence algorithm is exponential.
 - $O(r^n)$, where $r \approx 1.62$

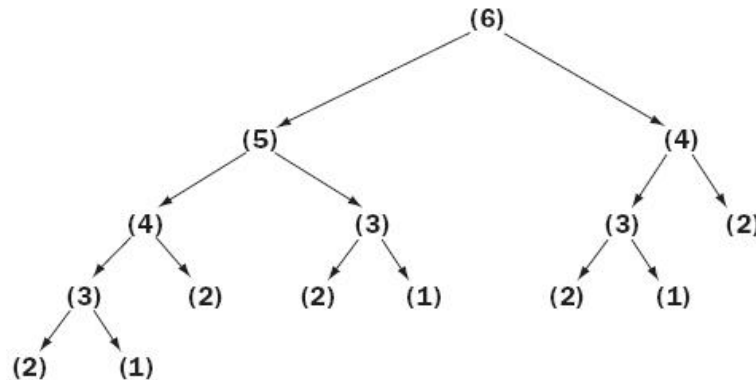


Figure 12-7: Calls needed to compute the sixth Fibonacci number recursively

Complexity Analysis (cont.)

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

Table 12-3: Calls needed to compute the *n*th Fibonacci number recursively

Complexity Analysis (cont.)

- 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?

Complexity Analysis (cont.)

- 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 $O(n/2)$

Complexity Analysis (cont.)

- 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 $O(n^2)$.

Binary Search

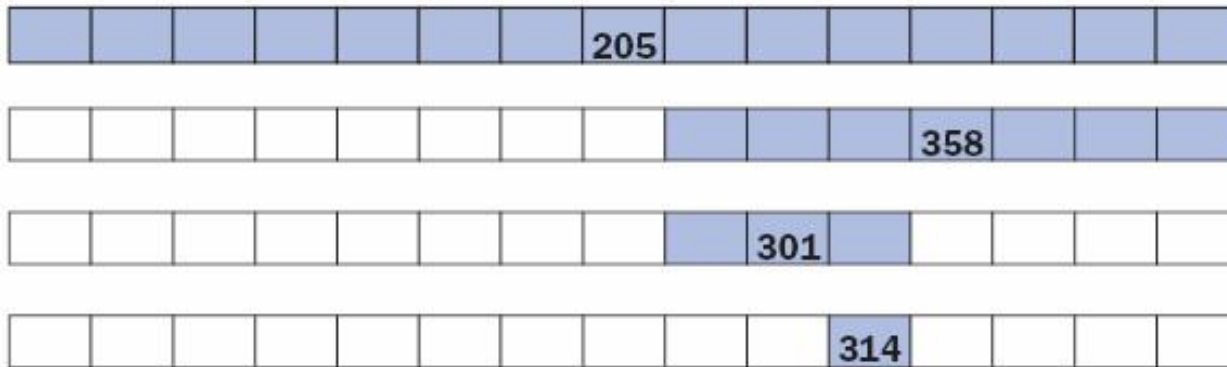


Figure 12-8: Binary search algorithm (searching for 320)

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

Figure 12-9: List for the binary search algorithm with all numbers visible

Binary Search (cont.)

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 $2^{n+1} - 1$	$n + 1$

Table 12-4: Maximum number of steps needed to binary search lists of various sizes

Binary Search (cont.)

- Iterative and recursive binary searches are $O(\log n)$.

```
// Iterative binary search of an ascending array
int search (int[] a, int target){
    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] == target) // Target found; return its index
            return midpoint;
        else if (a[midpoint] < target) // Target to right of midpoint
            left = midpoint + 1;
        else // Target to left of midpoint
            right = midpoint - 1;
    }
    return -1; // Target not found
}
```


Binary Search (cont.)

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

Quicksort

- Sorting algorithms, such as insertion sort and bubble sort, are $O(n^2)$.
- Quick Sort is $O(n \log n)$.
 - Break array into two parts and then move larger values to one end and smaller values to other end.
 - Recursively repeat procedure on each array half.

Quicksort (cont.)

5	12	3	11	2	7	20	10	8	4	9
---	----	---	----	---	---	----	----	---	---	---

Figure 12-11: An unsorted array

- Phase 1:

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

FIGURE 12-12

Step 2 of quicksort

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

Quicksort (cont.)

- Phase 1 (cont.):

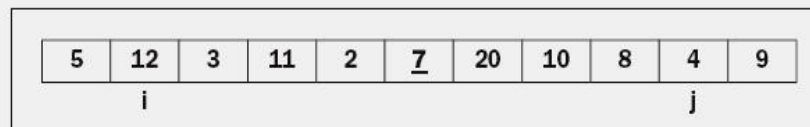
3. Tag the elements at the left and right ends of the array as i and j , respectively (Figure 12-13).

FIGURE 12-13
Step 3 of quicksort



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

FIGURE 12-14
Step 4 of quicksort

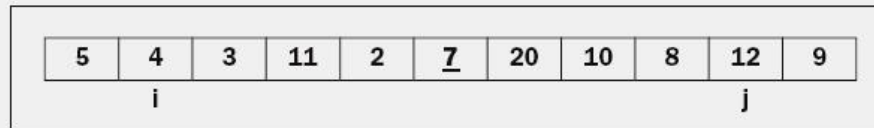


Quicksort (cont.)

- Phase 1 (cont.):

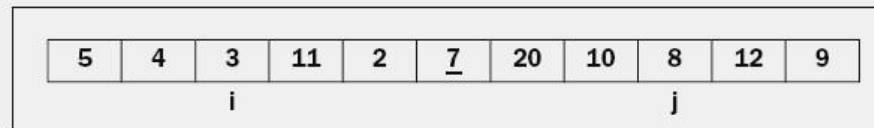
5. If $i > j$ then
 end the phase
else
 interchange $a[i]$ and $a[j]$ (Figure 12-15).

FIGURE 12-15
Step 5 of quicksort



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

FIGURE 12-16
Step 6 of quicksort



Quicksort (cont.)

- Phase 1 (cont.):

7. Repeat Step 4, that is,
While $a[i] < \text{pivot value}$, increment i
While $a[j] > \text{pivot value}$, decrement j (Figure 12-17).

FIGURE 12-17
Step 7 of quicksort

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

8. Repeat Step 5, that is,
If $i > j$ then
end the phase
else
interchange $a[i]$ and $a[j]$ (Figure 12-18).

FIGURE 12-18
Step 8 of quicksort

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

Quicksort (cont.)

- Phase 1 (cont.):

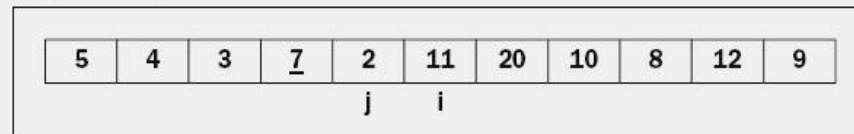
9. Repeat Step 6, that is,
Increment i and decrement j .
If $i > j$ then end the phase (Figure 12-19).

FIGURE 12-19
Step 9 of quicksort



10. Repeat Step 4, that is,
While $a[i] < \text{pivot value}$, increment i
While $a[j] > \text{pivot value}$, decrement j (Figure 12-20).

FIGURE 12-20
Step 10 of quicksort



Quicksort (cont.)

- Phase 1 (cont.):

12. 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 (Figure 12-21). 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 to the pivot.

FIGURE 12-21
Step 12 of quicksort

5	4	3	7	2	11	20	10	8	12	9
---	---	---	---	---	----	----	----	---	----	---

- Phase 2 and beyond: Recursively perform phase 1 on each half of the array.

Quicksort (cont.)

- Complexity analysis:
 - Amount of work in phase 1 is $O(n)$.
 - Amount of work in phase 2 and beyond is $O(n)$.
 - In the typical case, there will be $\log_2 n$ phases.
 - Overall complexity will be $O(n \log_2 n)$.

Quicksort (cont.)

- Implementation:

```
void quickSort (int[] a, int left, int right){  
  
    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);  
}
```

Merge Sort

- Recursive, divide-and-conquer approach
 - Compute middle position of an array and recursively sort left and right subarrays.
 - Merge sorted subarrays into single sorted array.
- Three methods required:
 - `mergeSort`: Public method called by clients
 - `mergeSortHelper`: Hides extra parameter required by recursive calls
 - `merge`: Implements merging process

Merge Sort (cont.)

```
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);
}
```

```
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);
    }
}
```

Merge Sort (cont.)

```
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
    }
```

Merge Sort (cont.)

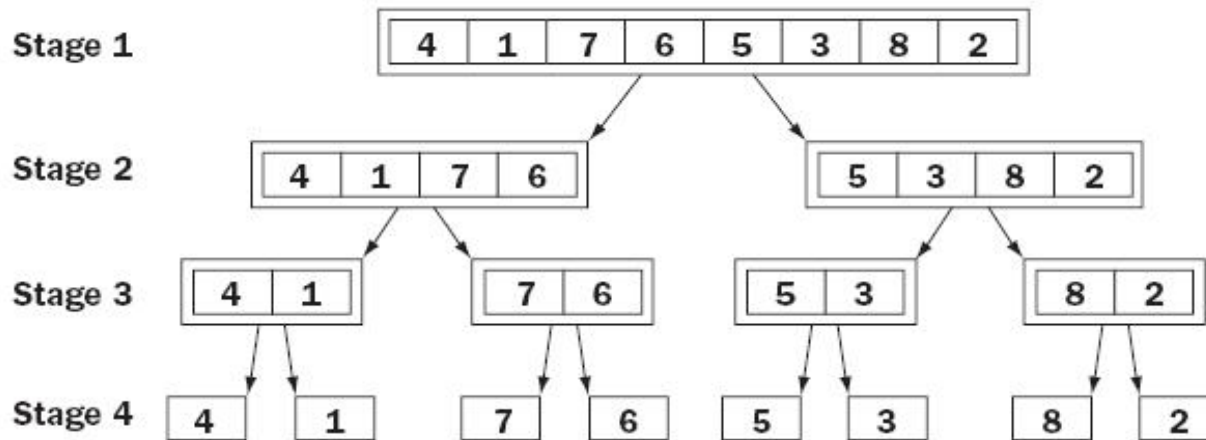


Figure 12-22: Subarrays generated during calls of `mergeSort`

Merge Sort (cont.)

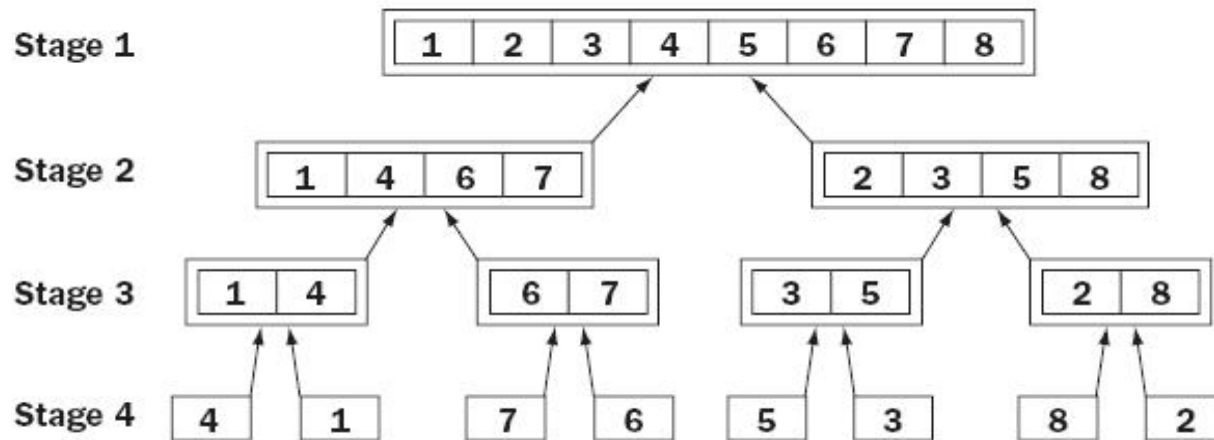


Figure 12-23: Merging the subarrays generated during a merge sort

Merge Sort (cont.)

- 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 $O(n \log n)$.

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.