Module 2

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 2 Study Guide and Deliverables

Topics:	• Algorithm Analysis • Recursion • Stacks and Queues
Readings:	• Module 2 online content • Textbook: Chapter 4, Chapter 5, Chapter 6
Assignments:	• Assignment 2 due **Tuesday, March 29 at 6:00 AM ET**
Assessments:	• Quiz 2 due **Tuesday, March 29, at 6:00 AM ET**
Live Classrooms:	• **Tuesday, March 22 from 8:00-9:30 PM** • **Thursday, March 24 from 8:00-10:00 PM** • Another one-hour live office hour session led by your facilitator: TBD

Module 2 Learning Objectives

In this module we discuss the following three topics:

- Algorithm analysis
- Recursion
- Stacks and queues

After successfully completing this module, you will be able to:

1. Analyze the running time of an algorithm.
2. Express the running time of an algorithm using an asymptotic notation such as *big-oh*.
3. Compare the performance of different algorithms in terms of running times.
4. Prove properties of computational problems and algorithms.
5. Illustrate the recursive call trace of a recursive algorithm.
6. Analyze the performance of a recursive algorithm.
7. Write a program that implements a recursive algorithm.
8. Illustrate stack operations.
9. Implement a stack ADT using an array.
10. Implement a stack ADT using a linked list.
11. Illustrate queue operations.
12. Implement a queue ADT using an array.
13. Implement a queue ADT using a linked list.

Section 2.1 Algorithm analysis

Section 2.1. Algorithm Analysis

Overview

An algorithm is a finite sequence of steps that solves a problem. The efficiency of an algorithm can be measured in terms of the memory/storage space it uses and the amount of time it takes to solve a problem. In this section, we discuss how to express the efficiency of an algorithm in terms of its running time. More specifically, we discuss how to express the running time of an algorithm as a function of the input size.

Section 2.1.1. Experimental Studies

An algorithm that solves a computing problem usually takes different amounts of time for different input sizes. Even when the input sizes are the same, the running times can be different for different inputs.

One way of evaluating the efficiency of an algorithm is to write a code that implements the algorithm, execute the code, and measure its actual elapsed time. A typical high-level programming language has a built-in feature that gives the current time. We get the current time first before starting the execution of the code and then immediately after the execution of the code. Then, we use the difference between the two current times as the elapsed time.

In Java, we can use the *currentTimeMillis*( ) method defined in the *System* class, which returns the current time in milliseconds, to calculate the elapsed time of the code that implements an algorithm.

The following code segment illustrates this method:

Code Segment 2.1

```
1   long startTime = System.currentTimeMillis();
2   /* execute the code of an algorithm */
3   long endTime = System.currentTimeMillis();
4   long elapsedTime = endTime – startTime;
```

If the execution time of an algorithm is extremely short, we can use the *nanotime* method in the *System* class.

To illustrate this experimental approach, the textbook compares two string-concatenation methods. The first method uses the string-concatenation operator "+", and the second uses the *append* method of the *StringBuilder* class. The experiment measures the elapsed times of both methods for various input sizes. The result shows that the second method is much faster, and its running time grows much more slowly than that of the first method.

As another example, we compared the elapsed times of two sorting algorithms: insertion-sort and quicksort (these sorting algorithms will be discussed later). The two sorting algorithms were applied on an array of integers. As described earlier, *current time* was obtained using System.*currentTimeMillis*( ) method before and after the invocation of each sorting algorithm, and the elapsed time was calculated from the two current times. This was performed ten times while the size of the array was increased from 10,000 to 100,000 with the increment of 10,000.

The result of this experiment is shown below as a table and a graph.

Table 2.1. Comparison of running times of insertion sort and quicksort

Array Size	Elapsed Time (ms)	
	Insertion-Sort	Quicksort
10000	180	20
20000	602	37
30000	1179	48
40000	2020	69

50000	3150	101
60000	4525	141
70000	6237	182
80000	7975	227
90000	10133	282
100000	12484	341

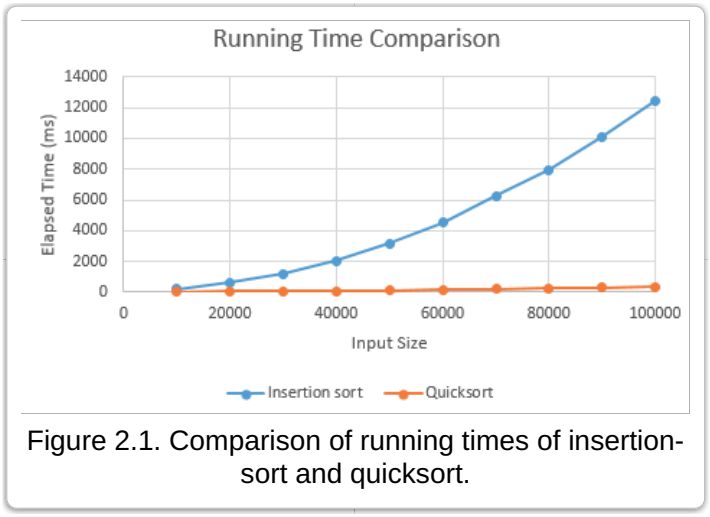


Figure 2.1. Comparison of running times of insertion-sort and quicksort.

From the above result, we can conclude that (1) the quicksort algorithm is much faster than the insertion-sort algorithm and (2) the running time of the insertion-sort algorithm grows more quickly than the running time of the quicksort algorithm as the input size is increased.

So, we can use this experimental approach when we estimate the running time of an algorithm or compare the running times of different algorithms.

However, this approach has some drawbacks:

- Experimental running times of different algorithms are difficult to compare accurately unless they are measured in the identical hardware and software environment, which is not always possible.
- We can perform such experiments only on a limited number of inputs. If the same experiments are performed on different inputs, we may obtain different results. So, the conclusion we draw from an experimental result cannot be generalized.
- To obtain and compare elapsed times, we need to fully implement the algorithms of which we want to compare the performance. However, most of the time, we want to compare the efficiency of different algorithms before we fully implement them. If we can determine which algorithm is more efficient with a high-level analysis without implementing either, we can save time.

We now discuss how to analyze the running time of algorithms at a high-level. A high level analysis possesses the following properties:

- It shows us, in relative terms, which algorithm is more efficient. When comparing the efficiency of two algorithms, it is not necessary to know and compare their actual running times. It would be sufficient to learn which one was faster.
- The decision is independent of the hardware and software environment.
- The decision is made by analyzing high-level descriptions of the algorithms without having to implement the algorithms.
- The decision is independent of input variations. In other words, a high-level analysis takes into consideration all possible inputs when making a determination.

When comparing the efficiency of algorithms, instead of measuring actual running times, we count the number of primitive operations each algorithm performs and use the counts for comparison. Examples of primitive operations follow:

- Assignment operation
- Accessing an object via its reference
- Performing an arithmetic operation (such as addition or subtraction)

- Comparing two numbers
- Accessing an array element using its index
- Invoking or returning from a method (excluding the execution of the method)

We assume that each primitive operation takes a constant amount of time and the execution times of different primitive operations are similar. We further assume that the number of primitive operations performed by an algorithm, or the *operation count*, is proportional to the running time of the algorithm. Then we can use the operation count as an estimate of the algorithm's running time for the purpose of comparing the running-time efficiencies of different algorithms.

When we compare the running-time efficiency of algorithms, we are more interested in how quickly or slowly an algorithm's running time grows as the size of input is increased. So, we express the running time of an algorithm (in terms of the operation count) as a function of input size, and we analyze the function's rate of growth.

The running time of an algorithm can vary with the input. An algorithm may run quickly for some inputs, but slowly for others. We ran the insertion-sort algorithm (which was used in the earlier experiment) on three arrays of integers and measured the elapsed times. All three arrays had 100,000 integers, but the integers' distributions were different. In the first array, integers were already sorted in the nondecreasing order. In the second array, integers were randomly distributed over the whole array. In the third array, integers were sorted in the reverse order. The result is shown below:

```
The first array (representing the best case)—1 ms
The second array (representing the average case)—12,145 ms
The third array (representing the  worst case)—24,810 ms
```

In general, we can analyze the running time of an algorithm for all three cases: best, average, and worst.

If, however, we want to perform only one analysis, we usually perform only a worst-case analysis. The reasons follow:

- Most often, the result of the average- and worst-case analyses are very similar.
- The worst-case analysis is usually much easier to perform than the average-case analysis.
- The worst-case analysis gives us an upper bound on the running time of an algorithm. So, it allows us to say, "*The algorithm won't take longer than this amount of time for any input*."

Section 2.1.2. Mathematical Functions

As mentioned in the previous section, when we analyze the running time of an algorithm, we express it as a function of input size. In this section we describe some mathematical functions that are frequently used to express the running times of algorithms. In the following discussions, *n* denotes the size of the input.

Constant Functions

$$f(n) = c$$

Here, *c* is some constant. The constant function represents the running time of an algorithm that is independent of the input size.

Logarithm Functions

$$f(n) = \log_b n$$

As we will see in later modules, many algorithms have a running-time expression that includes a logarithm function of the input size. Some logarithm identities are shown below:

$$\log_b (ac) = \log_b a + \log_b c$$

$$\log_b \left(\frac{a}{c}\right) = \log_b a - \log_b c$$

$$\log_b (a^c) = c \log_b a$$

$$\log_b a = \frac{\log_d a}{\log_d b}$$

$$b^{\log_d a} = a^{\log_d b}$$

Linear Function

$$f(n) = n$$

When the running time of an algorithm grows in proportion to the input size, there is a linear relationship between the running time and the input size. Such an algorithm's running time can be expressed as some linear function of the input size. A linear function grows more rapidly than a logarithm function, but more slowly than an *n*-log-*n* function.

N-log-N Function

$$f(n) = n \log n$$

The *n*-log-*n* function grows more rapidly than the linear function, but more slowly than the quadratic function. The average case running time of quicksort is $f(n) = n \log n$.

Quadratic Function

$$f(n) = n^2$$

The quadratic function grows more rapidly than the *n*-log-*n* function. If an algorithm has a nested loop and each loop has *n* iterations, then the running time of the algorithm is a quadratic function. The worst case running time of insertion-sort is a quadratic function.

Cubic Function and Other Polynomials

$$f(n) = n^3$$

The cubic function occurs sometimes, but not as frequently as other functions discussed earlier.

The linear, quadratic, and cubic functions are special cases of more general *polynomial functions*. A polynomial function has the following form:

$$f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$$

Here, $a_0, a_1, \ldots, a_d$ are constants called *coefficients*, and *d* is called the *degree* of the polynomial.

Some examples of polynomials follow:

$$f(n) = 5n^3 + 2n^2 - 8n + 5$$

$$f(n) = n^3 - 5$$

$$f(n) = 3n + 21$$

$$f(n) = 5n^4$$

Summations

A summation is defined as follows:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \cdots + f(b)$$

An example follows:

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

A polynomial $f(n)$ of degree *d* with coefficients $a_0, a_1, \ldots, a_d$ can be written as follows:

$$f(n) = \sum_{i=0}^{d} a_i n^i$$

Exponential Function

$$f(n) = b^n$$

The exponential function grows very quickly as the input size grows. So, if the running time of an algorithm is an exponential function, we consider the problem not solvable in practice (even though it is solvable theoretically). Such an algorithm is said to be intractable.

The following are some exponential identities:

$$(b^a)^c = b^{ac}$$

$$b^a b^c = b^{a+c}$$

$$\frac{b^a}{b^c} = b^{a-c}$$

Geometric Sums

The following summation is called *geometric* summation:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1}-1}{a-1}$$

Comparing Growth Rates

As mentioned earlier, when we compare the efficiency of algorithms, we do not actually implement the algorithms and compare the actual running times. Instead we perform high-level analysis with a high-level description of algorithms, usually given in the form of pseudocode, and express the running time of each algorithm as a function of the input size. Then, we analyze and compare the functions' rates of growth.

In this section, seven mathematical functions, which occur frequently in algorithm analysis, were briefly discussed. The following table shows the seven functions in increasing order of their growth rates:

Figure 2.2. Seven functions commonly used in algorithm analysis

constant	**logarithm**	**linear**	***n*-log-*n***	**quadratic**	**cubic**	**exponential**
c	$log\ n$	n	$n\ log\ n$	n^2	n^3	a^n

Section 2.1.3. Asymptotic Analysis

Asymptotic analysis of algorithms' running times tries to capture a *big-picture* aspect of these running times. We want to focus on the growth rate when the input size becomes very large, approaching infinity. This asymptotic analysis needs special notations to express the asymptotic behavior of algorithms' running times. In this section, we discuss three asymptotic notations: *big-oh*, *big-omega*, and *big-theta*.

Section 2.1.3.1. The *Big-Oh* Notation

Given two functions f: **N**⟶**R** and *g*: **N**⟶**R**, in which **N** is a set of natural numbers and **R** is a set of positive real numbers, *Big-oh* is defined as follows:

$$Og(n)) = \{f(n)\text{: there exist positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq c\,g(n) \text{ for } n \geq n_0\}$$

The above definition states, informally, that $g(n)$ is an upper bound on $f(n)$ with a constant factor c. We say $f(n)$ is *big-oh* of $g(n)$ and we write $f(n) = O(g(n))$ (even though, mathematically, $f(n)$ is a member of the set $O(g(n))$, and the correct expression is $f(n) \in O(g(n))$). We also say that $g(n)$ is an asymptotic upper bound of $f(n)$.

The following graph illustrates the concept of *big-oh*:

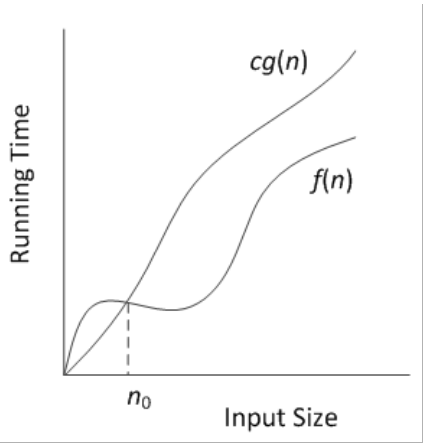


Figure 2.3. Illustration of *big-oh* notation.

As shown in the figure, $cg(n)$ is always greater than or equal to $f(n)$ for all n that is greater than or equal to some constant n_0 up to a constant factor c.

These are some examples:

- $f(n) = 3n + 2 => f(n) = O(n)$

  Proof:

  Let $g(n) = n$, $c = 4$, and $n_0 = 2$. Then,
  $f(n) = 3n + 2 \leq 4n$ for all $n \geq 2$, or
  $f(n) \leq cg(n)$ for all $n \geq n_0$, or
  So, $f(n) = O(n)$

- $\int(n) = 5n^3 + 2n^2 + 8n + 4 => \int(n) = O(n^3)$

  Proof:

  $$\begin{aligned} f(n) &= 5n^3 + 2n^2 + 8n + 4 \\ &\leq 5n^3 + 2n^3 + 8n^3 + 4n^3 \\ &= 19n^3 \end{aligned}$$

  If we let $g(n) = n^3$, $c = 19$ and $n_0 = 1$,
  $f(n) \leq cg(n)$ for all $n \geq n_0$
  So, $f(n) = O(n^3)$

When we prove $f(n) \leq cg(n)$, in general, there are many values of c and n_0 that satisfy the inequality. We don't need to consider all possible values. It is sufficient to choose some c and some n_0 and show that the pair satisfies the inequality.

In the above examples, we proved the *big-oh* of two functions. However, most of the time, we can decide the *big-oh* of a function using a high-level concept without a proof. The high-level concept here is the *rate of growth*.

Consider the second example, $f(n) = 5n^3 + 2n^2 + 8n + 4$. When the input size, n, is very large, the first term, $5n^3$, will be very large, and the remaining terms will be very small compared with the first term. In other words, when we consider the rate of growth, the contribution of the first term is dominant, and that of the remaining terms is negligible. So, we discard all lower-order terms. In addition, we also drop the coefficient of the first term because its contribution to the rate of growth is again negligible compared with that of the *n*3 term. So, we decide that $f(n) = O(n^3)$.

In general, once we express the running time of an algorithm as a function of the input size, we discard all lower terms and ignore the coefficient of the leading term to derive the *big-oh* of the running time.

More examples are shown below (without proof):

- $f(n) = 2n^2 + 2n \log n + 2n + 4 => O(n^2)$

$n \log n$, n, and 4 terms are lower than the n^2 term so they are discarded

- $f(n) = 2n \log n + 10n - 6 => O(n \log n)$
  n and 6 terms are lower than the $n \log n$ term, so they are discarded
- $f(n) = 5n + 23 \log n => O(n)$
  $\log n$ term is lower than the n term, so it is discarded
- $f(n) = 3 \log n + 10 => O(\log n)$
  6 is dropped.

Section 2.1.3.2. The *Big-Omega* Notation

Big-omega is defined as follows:

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $f(n) \geq c\,g(n)$ for $n \geq n_0$

We say $f(n)$ is *big-omega* of $g(n)$, and we write $f(n)_{=}\Omega(g(n))$. We also say that $g(n)$ is an asymptotic lower bound of $f(n)$. The following figure graphically illustrates this definition:

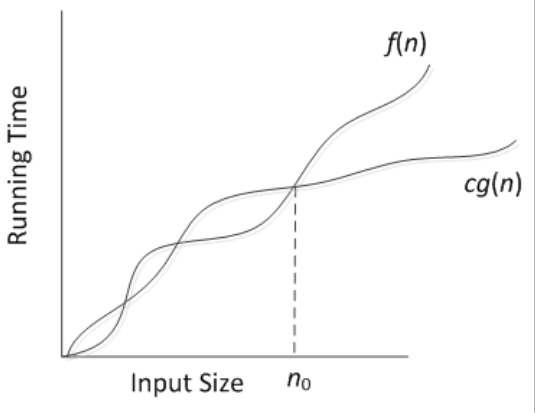


Figure 2.4. Illustration of *big-omega* notation.

Examples:

- $f(n) = 3n \log n - 2n = \Omega(n \log n)$
  Proof:
  $3n \log n - 2n = n \log n + 2n(\log n - 1) \geq n \log n$ for $n \geq 2$
  If we choose $c = 1$, and $n_0 = 2$, $f(n) \geq n \log n$ for $n \geq 2$, or
  $f(n) \geq c\,g(n)$ for all $n \geq n_0$
  So, $f(n) = \Omega(n \log n)$
- $f(n) = 5n^3 + 2n^2 + 8n + 4 => f(n) = \Omega(n^3)$
  Proof:
  $f(n) = 5n^3 + 2n^2 + 8n + 4$
  $> 5n^3$ for $n \geq 1$
  If we choose $c = 5$, and $n_0 = 1$, $f(n) > 5n^3$ for $n \geq 1$, or
  $f(n) \geq c\,g(n)$ for all $n \geq n_0$
  So, $f(n) = \Omega(n^3)$

Section 2.1.3.3. The *Big-Theta* Notation

A formal definition of *big-theta* is as follows:

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

We say $f(n)$ is *big-theta* of $g(n)$, and we write $f(n) = \Theta(g(n))$. We also say that $g(n)$ is an asymptotic tight bound of $f(n)$. The *big-omega* is graphically illustrated below:

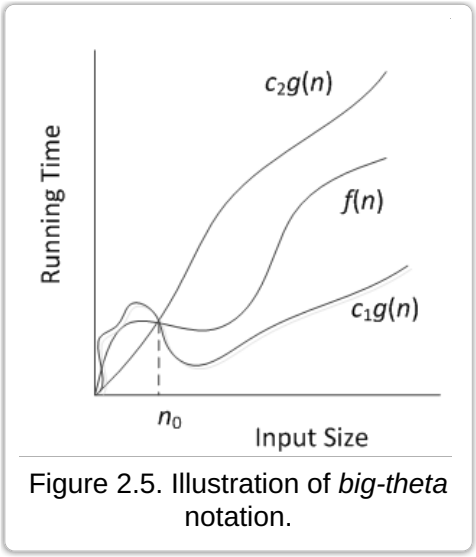


Figure 2.5. Illustration of *big-theta* notation.

Examples:

- $f(n) = 3n\log n + 4n + 5\log n => f(n) = \Theta(n\log n)$

  Proof:

  $3n\log n \le 3n\log n + 4n + 5\log n \le (3+4+5)\, n\log n$ for $n \ge 2$

  If we choose $c_1 = 3, c_2 = 12$, and $n_0 = 2$, then

  $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

  So, $f(n) = \Theta(n\log n)$

- $f(n) = 5n^3 + 2n^2 + 8n + 4 => f(n) = \Theta(n^3)$

  Proof:

  $5n^3 < 5n^3 + 2n^2 + 8n + 4 \le 19n^3$ for $n \ge 1$

  If we choose $c_1 = 5, c_2 = 19$, and $n_0 = 1$, then

  $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

  So, $f(n) = \Theta(n^3)$

Among the three asymptotic notations, the *big-oh* is frequently used to compare the running-time efficiency of algorithms. The seven functions we described above can be ordered by increasing rate of growth as follows:

$1, \log n, n, n\log n, n^2, n^3, 2^n$

The functions' rates of growth are illustrated in the following table for some selected n values:

Figure 2.6. The rate of growth of seven functions.

n	$\log n$	n	$\log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4294967296
64	6	64	384	4096	262144	1.84467E+19
128	7	128	896	16384	2097152	3.40282E+38

256	8	256	2048	65536	16777216	1.15792E+77
512	9	512	4608	262144	134217728	1.3408E+154

Section 2.1.3.4. Examples

In this section, we illustrate how to represent the running times of some simple algorithms using the *big-oh* notation.

Finding the Maximum in an Array

This algorithm finds the largest element in the given array. The following code segment is an implementation of the algorithm.

Code Segment 2.2

```
1   public static double arrayMax(double[] data) {
2     int n = data.length;
3     double currentMax = data[0];
4     for (int j=1; j < n; j++)
5       if (data[j] > currentMax)
6         currentMax = data[j];
7     return currentMax;
8    }
```

Each of lines 2 and 3 is executed once and takes a constant amount of time (accessing an array element using an index takes a constant amount of time): c_1 and c_2, respectively.

The body of the *for loop* of lines 4 through 6 is executed $n-1$ times, and a single execution takes a constant amount of time c_3.

Line 7 is executed once and takes a constant amount of time c_5.

So, the total running time $f(n) = c_1 + c_2 + c_3(n-1) + c_4$.
The function $f(n)$ can be rewritten as $f(n) = c_3 n + c_5$.

If we discard the lower order term and ignore the coefficient, we have $f(n) = O(n)$.

Composing Long Strings

We now consider an algorithm that constructs a string by concatenating individual characters. The code of an implementation is shown below:

Code Segment 2.3

```
1   public static String repeat1(char c, int n) {
2     String answer =  "";
3     for (int i=0; i < n; i++)
4       answer += c;
5     return answer;
6    }
```

When analyzing the running time of this implementation, we need to be aware that a string in Java is an immutable object. What this means with regard to this implementation is that, in line 5 a character C is not simply concatenated to the string *answer*. A new string is created to include the existing *answer* with C concatenated to it. The time to perform this operation is proportional to the length of the string.

So, in the first iteration of the *for* loop, the time for the operation is 1; in the second iteration of the *for* loop, the time taken is 2; and so on. So, the total time that is needed to create a new string of length n is as follows:

$$1 + 2 + \cdots + n$$

This is $O(n^2)$.

Three-Way Set Disjointness Problem

Given three sets of integers, which are implemented using arrays, we want to find out whether the intersection of the three sets is empty. Suppose that *A*, *B*, and *C* are the given three sets. Then this problem is designed to determine whether there is no element x such that $x \in A$, $x \in B$, and $x \in C$.

One straightforward implementation is shown below:

Code Segment 2.4

```
1   public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC){
2     for (int a : groupA)
3       for (int b : groupB)
4         for (int c : groupC)
5           if ((a == b) && (b == c))
6             return false
7     return true;
8   }
```

Assume that the size of each set is *n*. Then, the running time of the implementation is $O(n^3)$. This is because there are three levels of nested loops, and each level is executed n times.

An improved version is shown below:

Code Segment 2.5

```
1   public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC){
2     for (int a : groupA)
3       for (int b : groupB)
4         if (a == b)
5           for (int c : groupC)
6             if (b == c)
7               return false
8     return true;
9   }
```

In this code, an improvement is made based on the observation that if an element a (from *groupA*) and an element b (from *groupB*) are not identical, we don't need to check whether either is the same as an element c (from *groupC*). So, inside the loop for *groupB*, in line 4, we have an *if* statement that tests whether $a == b$. The test in line 5 is performed only if it is true.

In line 4, all possible pairs (a, b) from *groupA* and *groupB* are compared. So, line 4 is executed n^2 times. The inner loop of line 5 is executed only if a and b are identical. There are at most n such pairs. So, in lines 6 and 7, the body of the innermost loop over *groupC* is executed at most n^2 times.

For example, suppose that $n = 3$. There are nine total (a, b) pairs. So, line 4 is executed nine times (or n^2 times). Among these nine pairs, only three pairs can have an identical a and b (or at most n pairs). For each of these three pairs, the innermost loop is executed three times (or n times). So, lines 6 and 7 are executed at most nine times (or n^2 times).

So, the total running time is $O(n^2)$.

Element Uniqueness

We are given a bag of elements stored in an array. The array has potentially duplicate elements (since it is a bag). The *element uniqueness problem* is to determine whether all elements in the given array are distinct.

One straightforward algorithm is shown below:

Code Segment 2.6

```
1   public static boolean unique1(int[] data) {
2     int n = data.length;
3     for (int j=0; j < n-1; j++)
4       for (int k=j+1; k < n; k++)
5         if (data[j] == data[k])
6           return false; // found duplicate pair
7     return true;        // if we reach this, elements are unique
8   }
```

This algorithm returns *true* if all elements in the array are unique.

The outer *for* loop in line 3 considers the first $n-1$ elements, one at a time. Each element—say X—in the outer loop is then compared with all elements that come after X in the array, one at a time, in lines 4 and 5. For example, the first element *data*[0] is compare with *data*[1] through *data*[*n*-1], the second element *data*[1] is compare with *data*[2] through *data*[*n*-1], and so on. If any comparison results in *true*, there is a duplicate in the array and the algorithm returns *false* in line 6. If no duplicate is found, then the algorithm returns *true* in line 7.

In the worst case, therefore, the number of times the body of the inner *for* loop is executed is as follows:

$$f(n) = (n-1) + (n-2) + \cdots + 1$$

Or,

$$f(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$$

So, this algorithm has a quadratic running time.

There is a more efficient way of solving this problem. First, we sort the elements in the array. Then, since duplicate elements are stored in consecutive locations in the array, we can easily identify duplicate elements while scanning the whole array. A code is shown below:

Code Segment 2.7

```
1   public static boolean unique2(int[] data) {
2     int n = data.length;
3     int[] temp = Arrays.copyOf(data, n); // make copy of data
4     Arrays.sort(temp);                   // and sort the copy
5     for (int j=0; j < n-1; j++)
6       if (temp[j] == temp[j+1])     // check neighboring entries
7         return false;               // found duplicate pair
8     return true;                   // if we reach this, elements are unique
9   }
```

Line 4 sorts the elements in the array. The running time of this sorting depends on which sorting algorithm is used. The running time of the *Arrays.sort* method of Java, in line 4, takes $O(n \log n)$ in the worst case. Once the array is sorted, the *for loop* of lines 5 through 7 is executed n times. Lines 2, 3, and 8 run in constant amounts of time and are ignored. So, the running time of this algorithm, *unique*2, is $O(n \log n) + O(n) = O(n \log n)$ (we discard the lower-order term of $O(n)$).

Prefix Averages

Given a sequence of n numbers—$X_0, X_1, \ldots, X_{n-1}$—a prefix average a_j is the average of elements $X_0, X_1, \ldots, X_j$.

The *prefix average* problem is to calculate and return prefix averages a_0 through a_{n-1}. A general expression for calculating a_j, $0 \le j \le n-1$, is as follows:

$$a_j = \frac{\sum_{i=0}^{j} x_i}{j+1}$$

We describe two algorithms to solve this problem. One takes a quadratic time, or $O(n^2)$, and the second one takes a linear time, or $O(n)$.

The first algorithm calculates each prefix average $a[j]$ independently and is shown below:

Code Segment 2.8

```
1   public static double[] prefixAverage1(double[] x) {
2     int n = x.length;
3     double[] a = new double[n];  // filled with zeros by default
4     for (int j=0; j < n; j++) {
5       double total = 0;          // begin computing x[0] + ... + x[j]
6       for (int i=0; i <= j; i++)
7         total += x[i];
8       a[j] = total / (j+1);      // record the average
9     }
10    return a;
11  }
```

Line 2 takes a constant amount of time, or $O(1)$. Line 3, which initializes all n elements with 0, takes $O(n)$.

For each j of the outer loop (line 4), the inner loop (line 6) calculates the sum of elements x_0 through x_j, as illustrated below:

$j = 0 : a[j] = x[0]$ // line 7 executed once
$j = 1 : a[j] = x[0] + x[1]$ // line 7 executed twice
$j = 0 : a[j] = x[0] + x[1] + x[2]$ // line 7 executed three times
...
$j = 0 : a[j] = x[0] + x[1] + x[2] + \cdots + x[n-1]$ // line 7 executed n times

So, the calculation of all prefix averages $a[0]$ through $a[n-1]$ takes $1 + 2 + \cdots n = O(n^2)$. The total running time of the algorithm is $O(1) + O(n) + O(n^2) = O(n^2)$.

Next, we discuss an improved version of the *prefix average* algorithm, which runs in linear time.

In *prefixAverage*1, a prefix sum for each j is calculated in each iteration of the *while* loop and is stored in $a[j]$. Then it is used to calculate the prefix average for that j value. Note that, in this implementation, each time we calculate $a[j]$, we start from the beginning, adding $x[0]$ through $x[j]$, which takes $O[j]$. But this can be simplified. Suppose that we keep the current prefix sum—say, $a[j-1]$, which is the sum of $x[0]$ through $x[j-1]$. Then, when we compute the next prefix sum $a[j]$, instead of adding all elements from $x[0]$ to $x[j]$, we can simply add $x[j]$ to $a[j-1]$—i.e., $a[j] = a[j-1] + x[j]$—and this takes only $O(1)$. This improved implementation is shown below:

Code Segment 2.9

```
1   public static double[] prefixAverage2(double[] x) {
2     int n = x.length;
3     double[] a = new double[n]; // filled with zeros by default
4     double total = 0;        // compute prefix sum as x[0] + x[1] + ...
5     for (int j=0; j < n; j++) {
6       total += x[j];        // update prefix sum to include x[j]
7       a[j] = total / (j+1); // compute average based on current sum
8     }
9     return a;
10  }
```

The difference between this implementation and *prefixAverage*1 is as follows. In *prefixAverage*1, in each iteration, *total* is first initialized to 0 and calculated

by adding $x[0]$ through $x[j]$. But, in *prefixAveragei*2, *total* is initialized to 0 only once before the *for* loop begins (in line 4), and $x[j]$ is added to *total* in each iteration. So, *total* always keeps the current prefix sum and is used when calculating the next prefix sum (in line 6).

Let's analyze the running time of *prefixAverage*2. We begin by analyzing the running time of each line except the *for* loop. Line 1 takes $O(1)$, line 2 takes $O(n)$, line 3 takes $O(1)$, and line 9 takes $O(1)$.

The body of the *for* loop (lines 6 and 7) takes $O(1)$ and is executed n times. So, the execution of the *for* loop takes $O(n)$.

The total running time of *prefixAverage*2 is $O(1) + O(n) + O(1) + O(1) + O(n) = O(n)$.

Section 2.1.4. Simple Proof Techniques

In this section, we briefly and informally discuss proof techniques that are sometimes necessary in studying data structures and algorithms.

We discuss five proof techniques: direct proof, exhaustive proof, proof by contraposition, proof by contradiction, and induction. We also discuss a proof method called *loop invariant method*, which is used to prove the correctness of a code segment that involves a loop.

Before that, we briefly discuss disproving by a counterexample.

Section 2.1.4.1. Disproving by Counterexample

To prove a statement $P \rightarrow Q$, in which P and Q are statements, it is necessary to show that if P is true, Q is also true for all objects in the domain under consideration. However, if you need to disprove the argument, you don't need to show that the argument is false for all objects. It is sufficient to show that the argument is false for one object. In other words, we disprove the statement by showing a *counterexample.*

Example: Consider the statement "If n is an even number, then n^2 is an odd number." To disprove this statement, it is sufficient to show one counterexample. Let $n = 2$. Then $n^2 = 4$, which is not an odd number. So, we conclude that the statement is false.

Section 2.1.4.2. Exhaustive Proof

If the domain of objects under consideration is small, we can prove a statement by considering all objects in the domain. We can use this proof method when the domain has a finite set of objects and the number of objects is relatively small.

Example: Suppose you want to prove that, if n is a positive integer smaller than 10, then $2n$ is smaller than 20. You can prove this by evaluating the truth value of the statement for $n = 1$, $n = 2$, ..., and $n = 9$. Obviously, the truth value is true for all nine integer values, thus proving the statement.

Section 2.1.4.3. Direct Proof

To prove $P \rightarrow Q$, we begin with P, which is assumed to be true, and deduce another statement P_1, which is a logical consequence of P. Next, we deduce another statement P_2, which is true if P_1 is true. We continue this derivation of a sequence of statements until we reach Q.

Example: If an even integer is added to another even integer, the result is an even integer.

Proof:

Let x and y be two even integers. Let $z = x + y$.
Since x is even, it can be rewritten as $x = 2n$, for some integer n.
Since y is even, it can be rewritten as $y = 2m$, for some integer m.
Then, $z = x + y = 2n + 2m = 2(n + m)$. So, z is even.

Section 2.1.4.4. Proof by Contraposition

Given a statement $P \rightarrow Q$, the contraposition of the statement is $Q' \rightarrow P'$. In logic, if the contraposition of a statement is true, the statement is true, or $(Q' \rightarrow P') \rightarrow (P \rightarrow Q)$. So, if we prove $(Q' \rightarrow P')$ is true, then we can conclude $(P \rightarrow Q)$ is also true.

Example: If n candies are distributed to *m* children, where $n > m$, then at least one child has more than one candy. The contraposition of this statement

is, "If every child has at most one candy, then the total number of candies is less than or equal to the number of children." Proof of this contraposition is trivial (if there are *m* children and each child has at most one candy, then the total number of candies cannot exceed m and, therefore, the original statement is true.

Section 2.1.4.5. Proof by Contradiction

To prove $P \rightarrow Q$ is true, we assume that Q is false (or negate Q) and find a contradiction.

Example: If an even integer is added to another even integer, the result is an even integer. This is the same statement we proved using the direct-proof method.

Proof:

Let x and y be two even integers. Let $z = x + y$.

Let's assume that z is an odd integer (negating the conclusion of the given statement).

Since x is even, it can be rewritten as $x = 2n$, for some integer n.

Since y is even, it can be rewritten as $y = 2m$, for some integer m.

Since z is odd, it can be rewritten as $z = 2k + 1$, for some integer k.

Then we have the following:

$$x + y = z$$

$$2n + 2m = 2k + 1$$

$$2n + 2m - 2k = 1$$

$$2(n + m - k) = 1$$

This is a contradiction because the left hand side is an even number, and the right-hand side is 1, which is odd.

Section 2.1.4.6. Induction

Suppose that we want to prove a statement $P(n)$ is true for all positive integers n. Maybe we will be able to prove this statement using one of the proof methods discussed above. However, for this type of proof problem, there is an effective method, which works as follows.

First, we show that $P(1)$ is true.
Second, we show that if $P(k)$ is true for any positive integer k, then $P(k + 1)$ is also true.

By the first step, we know that $P(1)$ is true. By the second step, we can conclude that $P(2)$ is true (because $P(1)$ is true). Then, for the same reason, $P(3)$ is true, and so on. In this way, we can argue that $P(n)$ is true for all positive integers n.

The first step is called the *basis*, *base step*, or *base case*. The second step is called the *inductive step*. The $P(k)$ in the second step is referred to as the *inductive hypothesis*.

Example: Prove that, for any positive integer n, $2^n > n$.

Base case: $n = 1$

$LHS = 2^1 = 2, RHS + 1$; so, LHS > RHS

Induction step: Assume it is true for $n = k$ $(k \geq 1)$—i.e., $2^k > k$ for $k \geq 1$.

We show that it is also true for $n = K + 1$—i.e.,$2^{k+1} > k + 1$

$$\begin{aligned} LHS &= 2^{k+1} \\ &= 2 \times 2^k \\ &> 2k \\ &= k + k \geq k + 1 \\ &= RHS \end{aligned}$$

So, LHS > RHS.

Example: Prove that $1 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ for all positive integers.

Base case: $n = 1$

$LHS = 1 + 2 = 3, RHS = 2^{1+1} - 1 = 3$; so, LHS = RHS

Inductive step: Assume it is true for $n = k$, i.e., $1 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$

We show it is also true for $n = k + 1$, i.e., $1 + 2^1 + 2^2 + \cdots + 2^k + 2^{k+1} = 2^{k+2} - 1$

$$\begin{aligned} LHS &= 1 + 2^1 + 2^2 + \cdots + 2^k + 2^{k+1} \\ &= 2^{k+1} - 1 + 2^{k+1} \\ &= 2 \times 2^{k+1} - 1 \\ &= 2^{k+2} - 1 \\ &= RHS \end{aligned}$$

Section 2.1.4.7. Loop Invariant

To prove that the execution of a loop in an algorithm (or a program) is correct, we can use the *loop invariant method*. Note that this method does not prove the correctness of the whole algorithm. Instead it proves that the execution of the loop is correct. However, when a loop is a main part of the algorithm, often the correctness of its execution leads to the correctness of the algorithm.

More specifically, a loop invariant is a predicate (or a property) that is true during the execution of all iterations of the loop. In other words, a loop invariant is true before the first iteration of the loop, before the second iteration of the loop, and so on, and it is also true at the termination of the loop.

To prove that a loop invariant is true with regard to a loop, we use an induction.

Assume that we want to prove that a loop invariant $\mathcal{C}$ about a loop is true. Also assume that the loop goes through k iterations. We proceed as follows:

1. Base case—We show that $\mathcal{C}$ is true at the beginning of iteration 0, which is the first iteration.
2. Induction step—We assume that $\mathcal{C}$ is true at the beginning of iteration j, and we show that $\mathcal{C}$ is also true at the beginning of iteration $(j + 1)$.

To illustrate how to use the *loop invariant* method to prove the correctness of an algorithm, we use a simple program that searches an array for a given value. A pseudocode of the algorithm is shown below.

Input:

data—an array of integers
val—the value that we are looking for

Output:

If the value *val* is found in the array, the index of the value is returned. If the value is not in the array, –1 is returned.

```
1   arrayFind
2     n = data.length
3     j = 0
4     while (j < n) {
5       if data[j] == val
6         return j
7       j = j + 1
8     } // end of while
```

```
9     return –1
```

Note that, in the pseudocode, $data[0]$ is compared with *val* during iteration 0 (again, this is the first iteration). In general, $data[j]$ is compared with *val* during iteration *j,* and first j elements $(data[0 \cdots j-1])$ have been compared with *val* before iteration j.

The loop invariant for the while loop (lines 4 through 8) is as follows:

$\mathcal{L}$—val is not equal to any of the elements in $data[0 \cdots j-1]$ (first j elements of data) at the beginning of iteration j.

We first prove that this loop invariant is true for all iterations using induction.

Base case: Before iteration 0 (the first iteration), $j = 0$, and there is no element in $data[0 \cdots j-1] = data[0 \cdots -1]$. So, the loop invariant $\mathcal{L}$ is trivially true (this is also said to be *vacuously* true).

Induction step: Assume that $\mathcal{L}$ is true at the beginning of iteration j. This means val is not equal to any element in $data[0 \cdots j-1]$. During the execution of the iteration j, $data[j]$ is compared with val. If $data[j] = val$, the value is found and the index j is returned. If they are not identical, then j is incremented and execution proceeds to iteration $(j+1)$. Since $data[j] \neq val$, at the beginning of iteration $(j+1)$, val is not equal to any of elements in $data[0 \cdots j]$. So, the loop invariant $\mathcal{L}$ is also true at the beginning of iteration $(j+1)$.

This completes the proof of the loop invariant. Now we prove the correctness of the algorithm using this loop invariant. The *while* loop terminates in one of two ways: (1) If val is in $data$, the loop terminates prematurely, and the algorithm returns the index of the array of which the value is identical to val. (2) If val is not in $data$, the *while* loop runs all n iterations, and the algorithm returns –1. So, the algorithm does what it is intended to do—i.e., the algorithm is correct.

Test Yourself 2.1

Prove that the sum of two odd integers is an even integer using the proof by contradiction method.

Please think carefully, write your answer, and then click "Show Answer" to compare yours to the suggested answer.

Suggested answer:

Let x and y be two odd integers. Let $z = x + y$.

Let's assume that z is an odd integer (negating the conclusion of the given statement).

Since x is odd, it can be rewritten as $x = 2n + 1$, for some integer n.

Since y is odd, it can be rewritten as $y = 2m + 1$, for some integer m.

Since z is odd (this we assumed), it can be rewritten as $z = 2k + 1$, for some integer k.

Then, we have

$$x + y = z$$
$$2n + 1 + 2m + 1 = 2k + 1$$
$$2n + 2m - 2k + 2 = 1$$
$$2(n + m - k + 1) = 1$$

This is a contradiction because the left hand side is an even number and the right hand side is odd.

Section 2.2 Recursion

Section 2.2. Recursion

Overview

In mathematics, a recursive function is a function that is defined in terms of itself. In programming, a recursive method is a method that calls itself. Recursion is a powerful programming technique that can be used to describe repeated execution of statements. So, recursion is an alternative to iterative

constructs, which include *while* and *for* loops.

We first illustrate the mechanism of recursion using examples. Then we discuss how to analyze recursive algorithms. We also discuss some common forms of design.

Section 2.2.1. Examples

In this section, four examples are used to illustrate the concept and mechanisms of recursion.

Section 2.2.1.1. Factorial

The *factorial function* is defined as follows:

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

By this definition, $(n-1) \times (n-2) \times \ldots \times 2 \times 1$ can be rewritten as $(n-1)!$. So, we can rewrite the definition recursively as follows:

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

A typical recursive definition has two parts: one or more *base cases* and one or more *recursive cases*. The function is expressed in terms of itself in the recursive case. In the above definition, there is one base case (when $n = 0$) and one recursive case (when $n \geq 1$), where $n!$ is defined in terms of $(n-1)!$.

Implementing the factorial function recursively is straightforward. A Java implementation is shown below:

Code Segment 2.10

```
1   public static int factorial(int n) throws IllegalArgumentException {
2     if (n < 0)
3       throw new IllegalArgumentException(); // argument must be nonnegative
4     else if (n == 0)
5       return 1;                            // base case
6     else
7       return n * factorial(n-1);          // recursive case
8   }
```

Line 2 checks that *n* is not a negative number. Lines 4 and 5 implement the base case, and lines 6 and 7 implement the recursive case.

One thing we need to make sure of with recursion is that the recursion eventually stops, without going through infinite recursive calls. Usually, there is an expression (or a quantity) in a recursive program that is decreased by a fixed amount in each recursive call. When this expression becomes smaller than (or equal to) a threshold, the recursion stops. In this example, the expression is *n* and it is decreased by one in each call (line 7). When it becomes 0 (line 4), the recursion stops.

The following figure illustrates how the factorial method is executed by successive calls to itself:

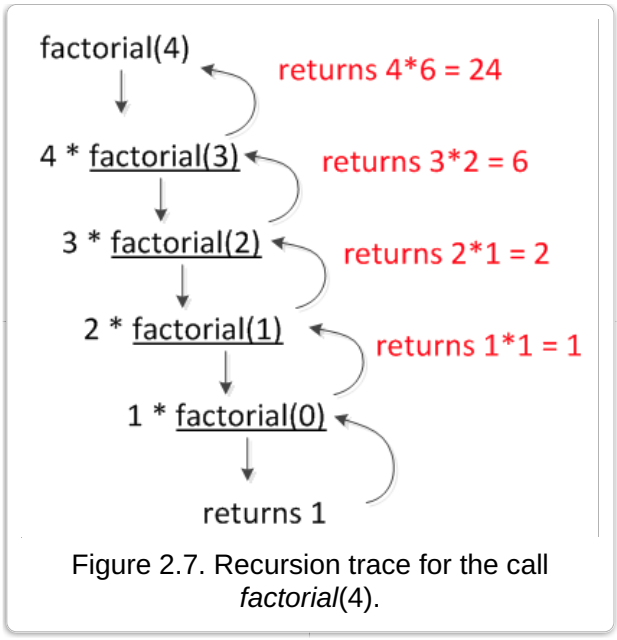
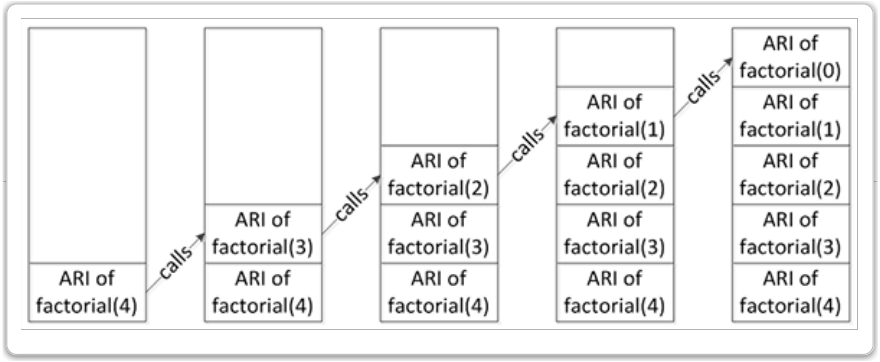


Figure 2.7. Recursion trace for the call *factorial*(4).

In a typical programming-language system, when a method is invoked, a data structure called *Activation Record Instance* (ARI) is created for the method. The ARI stores parameters and the local variable and other information necessary to execute the method. When a recursive program is executed, an ARI is created for each recursive call and the ARI's of successive calls are created on a stack. For example, when *factorial*(4) is invoked, an ARI is created for it and is placed in a stack. Next, when *factorial*(3) is invoked, the ARI for this recursive call is created and placed on top of the ARI of *factorial*(4), and so on. When the execution of a method call is finished, the corresponding ARI is removed from the stack, and execution returns to the ARI below it.

The change of stack content when *factorial*(4) is invoked is shown below:



Returning from recursive calls.

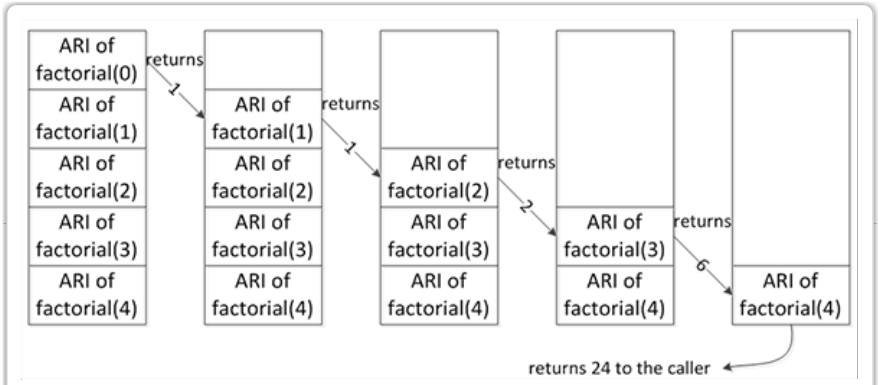


Figure 2.8. Change of stack contents when *factorial*(4) is called.

Section 2.2.1.2. Binary Search

When searching a sequence of elements for a target element, we can use either *linear search* or *binary search*, depending on whether the elements in the sequence are sorted or not.

Suppose we have a sequence of *n* elements. When elements in the sequence are not sorted, then we need to examine all elements one at a time, beginning with the first element. This type of search is called a *linear search*. In the worst case, all *n* elements need to be examined. In the best case, only one comparison is needed. On average, *n*/2 elements are examined and the running time is $O(n)$.

If elements in the sequence are sorted, we can use a more efficient method, called *binary search*. The idea is as follows. Let *target* be the element that is being searched for. First, we compare *target* with the element that is at the "middle" position in the sequence. Let's call this element the *median candidate* of the sequence. If *target* is identical to the *median candidate*, then we have found *target*. If *target* is smaller than the *median candidate*, then *target* must be in the left half of the sequence, so we recursively search the left half of the sequence. If *target* is greater than the *median candidate*, then we recursively search the right half of the sequence.

A pseudocode of the algorithm is shown below. In the pseudocode, elements are integers, and *low* and *high* are the lowest and highest indexes of a subarray, respectively. Initially, *low* = 0 and *high* = *n* – 1.

```
Algorithm binarySearch(int[] data, int target, int low, int high)
  If low > high             // target is not found
    return false
  else
    mid = floor((low + high)/2) // median candidate
    if target = data[mid]       // target found
      return true
    else if target < data[mid]
      search data[low .. mid-1] recursively
    else
      search data[mid+1 .. high] recursively
```

A Java code implementing the pseudocode is shown below:

Code Segment 2.11

```
1   public static boolean binarySearch(int[] data, int target,
                                       int low, int high) {
2   if (low > high)
3     return false;                // interval empty; no match
4   else {
5     int mid = (low + high) / 2;
6     if (target == data[mid])
7       return true;               // found a match
8     else if (target < data[mid])
        // recurse left of the middle
9       return binarySearch(data, target, low, mid - 1);
10    else
        // recurse right of the middle
11      return binarySearch(data, target, mid + 1, high);
12    }
13  }
```

The following figure illustrates a search process:

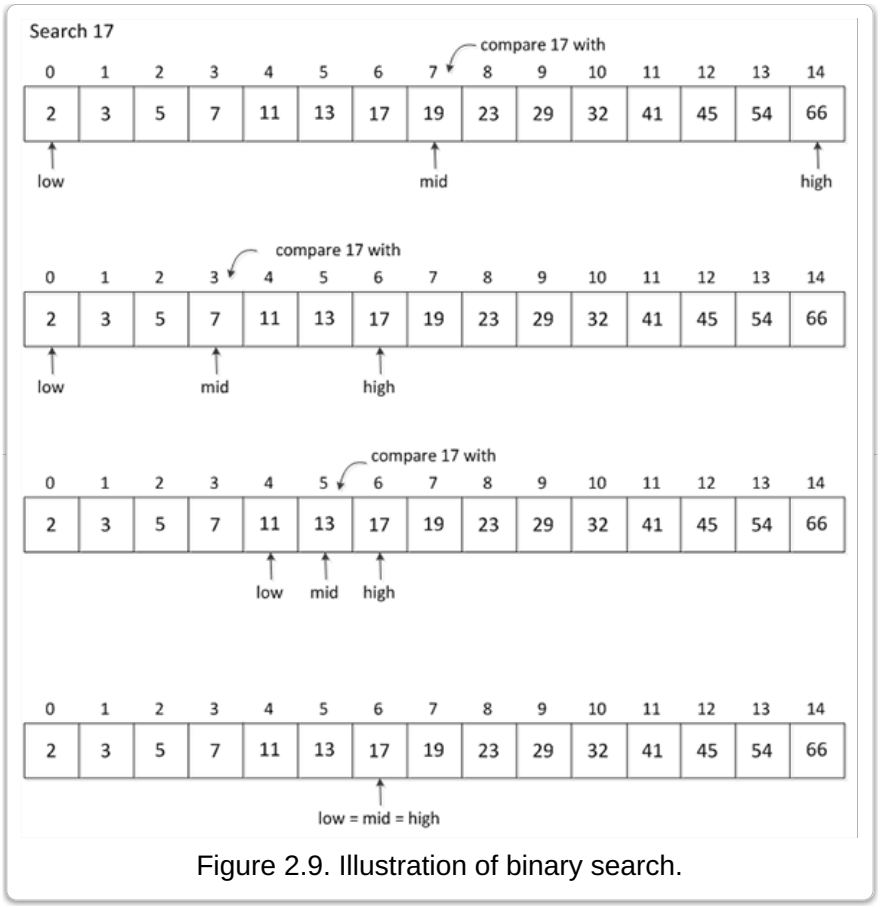


Figure 2.9. Illustration of binary search.

Section 2.2.2. Analysis of Recursive Algorithms

One way of analyzing the running time of a recursive algorithm is as follows. First, we analyze the running time of one execution of a recursive method. Next, we count the number of times the recursive method is invoked. Then we multiply the two.

We now show how to analyze the running time of the three recursive algorithms we discussed in the previous section – *factorial* and *binary search*.

Factorial

When $factorial(n)$ is invoked, it is recursively invoked n more times. So, the total number of invocations is $n + 1$. Each invocation of $factorial(n)$ takes a constant amount of time, or $O(1)$. So, the total running time of $factorial(n)$ is $O(n)$.

Binary Search

Execution of each invocation of *binary search* takes a constant amount of time. We show that the total number of times *binary search* is invoked is at most $\lfloor \log n \rfloor + 1$.

When *binary search* is invoked (recursively), the number of elements to be searched is reduced by at least half of the elements in the current subarray. Initially, the number of elements to be searched is n. In the first recursive invocation, the number of elements to be searched is at most $\frac{n}{2}$. In the second recursive invocation, the number of elements to be searched is at most $\frac{n}{4}$, and so on. So, in the j^{th} invocation, at most $\frac{n}{2^j}$ are searched. In the worst case, the *target* is not in the initial array, and the recursion stops when there are no more elements to be searched. In the example used in Section 2.2.1.3, $n = 15$. The numbers of elements to be searched for in each recursive call are shown below:

```
Initial call           15
First recursive call   7
Second recursive call  3
Third recursive call   1 (recursion terminates)
```

The total number of invocations of *binary search*, including the initial call, is 4. In general, the maximum number of invocations of *binary search* is the smallest integer r, such that the following is true:

$$\frac{n}{2^r} < 1$$

The inequality can be rewritten as $n < 2^r$. Taking the log of both sides, we have $\log n < r$. So, the smallest integer that satisfies this inequality is as follows:

$$r = \lfloor \log n \rfloor + 1$$

So, the total number of times *binary search* is invoked is at most $\lfloor \log n \rfloor + 1$. Therefore, the running time of *binary search* is $O(\log n)$.

Section 2.2.3. More Recursion Examples

In general, in a single invocation of a recursive program, the next recursion step may include only one recursive call, two recursive calls, or more than two recursive calls. So, recursions can be categorized in the following three types:

- *Linear recursion*: At most one recursive call is made.
- *Binary recursion*: Two recursive calls are made.
- *Multiple recursion*: Three or more recursive calls are made.

Section 2.2.3.1. Linear Recursion

The factorial function we discussed earlier is an example of linear recursion because, in each invocation of the function, one recursive call is made. The binary-search algorithm is also an example of linear recursion. Even though the given array (or a subarray) is divided into two subarrays, a recursive call is made on only one of the two. So, it is, by definition, linear recursion.

Printing Array Elements Recursively

We can print the elements of an array one at a time while scanning the whole array, beginning with the first element. We can also print them in a recursive way. The following pseudocode describes how to print array elements recursively. Here, *data* is an array, *i* is the index of an array element to be printed, and *n* is the number of elements in the array. The value of *i* in the initial call is 0.

```
Algorithm printArrayRecursively(data, i)
if i = n, return
else
print data[i]
i = i + 1
printArrayRecursively(data, i)
```

A Java implementation is given below:

Code Segment 2.12

```
1   public static void printArrayRecursive(int[ ] data, int i){
2     if (i == data.length)
3       return;
4      else{
5        System.out.print(data[i++] + " ");
6        printArrayRecursive(data, i);
7      }
8   }
```

Printing Array Elements in Reverse Order Recursively

The following pseudocode prints the elements of an array in the reverse order. Here, *data* is an array of integers and *n* is the number of elements in the array. Note that the index of the first element is 0.

```
Algorithm printReverse(data, n)
if n = 0, return
else
print data[n-1]
printReverse(data, n-1)
```

A Java implementation of the pseudocode is given below:

Code Segment 2.13

```
1   public static void printReverse(int[ ] data, int n){
2     if (n == 0)
3       return;
4     else{
5       System.out.print(data[n-1] + "  ");
6       printReverse(data, n-1);
7     }
8   }
```

Reversing a Sequence with Recursion

Suppose we want to reverse the order of elements in a given sequence. One way of doing this is to swap the first element with the last, swap the second element with the second to last, and so on. This can be implemented with linear recursion. A pseudocode is given below:

Code Segment 2.14

```
1   Algorithm reverseArray(data, low, high)
2     if low >= high, return
3     else
4       swap data[low] with data[high]
5       reverseArray(data, low+1, high-1)
```

In the pseudocode, *data* is an array, *low* is the index of the first element of an array (or a subarray), and *high* is the index of the last element of an array (or a subarray).

A Java implementation follows:

Code Segment 2.15

```
1   public static void reverseArray(int[] data, int low, int high){
2     if (low < high) {
3       int temp = data[low];
4       data[low] = data[high];
5       data[high] = temp;
6       reverseArray(data, low + 1, high - 1);
7     }
8   }
```

The initial call to this method is made with *reverseArray*(*data*, 0, *n*-1), where *n* is the number of elements in the array. Line 2 checks that there are at least two elements (because otherwise no swapping is needed). Lines 3, 4, and 5 swap the first and last elements of a subarray. Line 6 invokes itself recursively with a subarray excluding the first and the last elements.

The following figure illustrates how this method works:

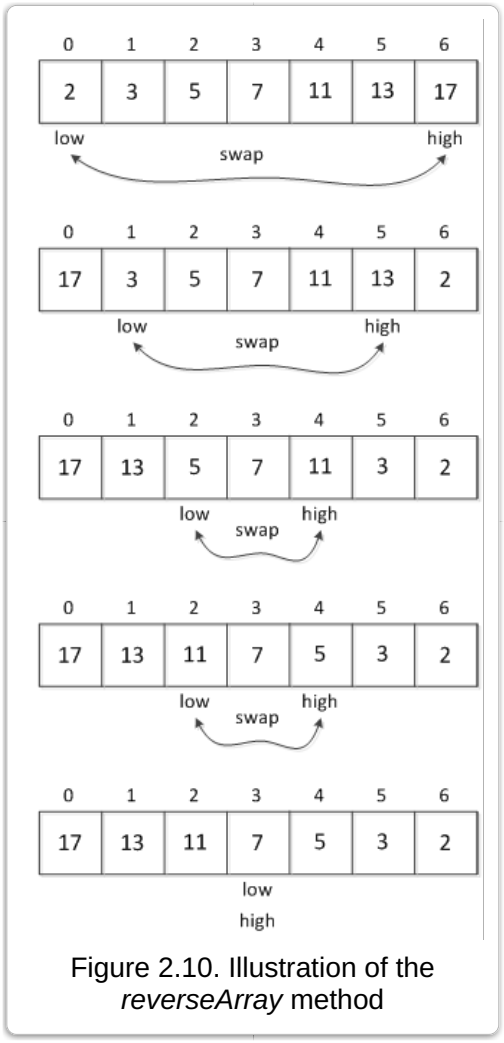


Figure 2.10. Illustration of the *reverseArray* method

Computing Powers Recursively

The *power* function is defined as follows:

$$power(x, n) = x^n$$

It can be redefined recursively as follows:

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

Implementation of this recursive definition is straightforward. A Java implementation is shown below:

Code Segment 2.16

```
1   public static double power(double x, int n) {
2     if (n == 0)
3       return 1;
4     else
5       return x * power(x, n-1);
6   }
```

The running time of this implementation is $O(n)$ because the *power* function is invoked $n + 1$ times (including the initial call).

There is a more efficient implementation. Suppose that $k = \lfloor \frac{n}{2} \rfloor$. When n is even, $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$.

When n is odd, $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$. So, the following are true:

$$(x^k)^2 = \left(x^{\frac{n}{2}}\right)^2 = x^n \qquad \text{when } n \text{ is even}$$

$$(x^k)^2 = \left(x^{\frac{n-1}{2}}\right)^2 = x^{n-1} \qquad \text{when } n \text{ is odd}$$

Based on this, we can redefine $power(x, n)$ as follows:

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

A Java implementation of this redefined *power* method is given below:

Code Segment 2.17

```
1   public static double power(double x, int n) {
2     if (n == 0)
3       return 1;
4     else {
5       double partial = power(x, n/2); // use integer division of n
6       double result = partial * partial;
7       if (n % 2 == 1)      // if n is odd, include extra factor of x
8         result *= x;
9       return result;
10    }
11  }
```

In line 5, $power\left(x, \lfloor \frac{n}{2} \rfloor\right)$ is computed, and in line 6, it is squared. If *n* is odd, *x* is multiplied in line 8.

The execution of $power(2, 5)$ is illustrated in the following figure:

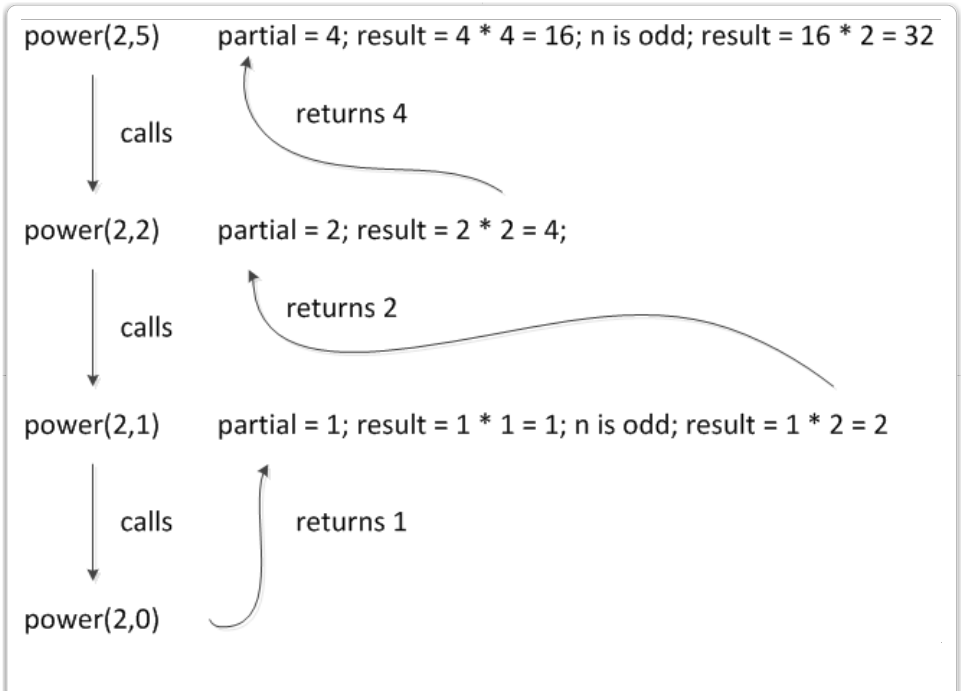


Figure 2.11. Recursion trace for the call *power*(2, 5).

Section 2.2.3.2. Binary Recursion

When a method makes two recursive calls in a single invocation, it is called *binary recursion*.

As an example, we consider summing integers in a given array. This problem can be solved by *linear recursion*. But, in this section, we use *binary recursion* to solve the problem.

The underlying idea is as follows.

1. The given array is split into a left half and a right half.
2. The sum of elements in each half is calculated in a recursive manner.
3. The sum of the left half and the sum of the right half are added to give the total sum.

Note that, in step 2, recursive calls are made on both halves.

A Java implementation is given below:

Code Segment 2.18

```
1   public static int binarySum(int[] data, int low, int high) {
2     if (low > high)            // zero elements in subarray
3       return 0;
4     else if (low == high)      // one element in subarray
5       return data[low];
6     else {
7       int mid = (low + high) / 2;
8       return binarySum(data, low, mid) + binarySum(data, mid+1, high);
9     }
10  }
```

When there is no element in the subarray, it returns 0 (lines 2 and 3), and when there is one element in the subarray, it returns that element (lines 4 and 5). Otherwise, two recursive calls are made, one on the left half and the other on the right half, and their results are added (in line 8).

The following figure shows the execution of the code with the initial array of size 8.

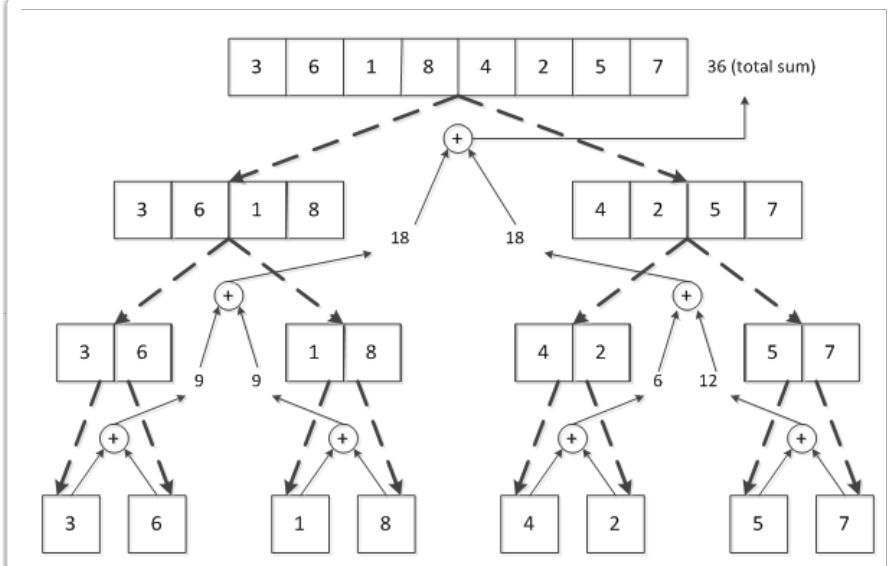


Figure 2.12. Recursively calculating the sum of array elements.

In the figure, dashed, thick lines denote recursive calls, and solid, thin lines represent partial sums returned by calls.

Section 2.2.3.3. Multiple Recursion

When a method makes more than two recursive calls, it is referred to as *multiple recursion*. In the textbook, the *summation puzzles* problem is used to illustrate multiple recursion. In this section, we generate a permutation of a sequence of characters using multiple recursion. For example, if a given string is "XYZ," then the output is as follows:

XYZ, XZY, YXZ, YZX, ZXY, ZYX

A pseudocode of the algorithm is shown below:

```
1   Algorithm Permutation
2     Input: sequence of characters S
3     Output: print all permutations of characters in S

4     sequence P = ∅; // will hold each permutation; initially empty
5     PrintPermutation(S.size, P, S);

6     PrintPermutation(int n, sequence P, sequence S)
7       for each c in S
8         remove c from S
9         append c to P
10        if n = 1
11          print P
12        else
13          PrintPermutation(n-1, P, S)
14        remove c from P
15        add c to S
```

The algorithm *Permutation* receives a sequence of characters and prints all permutations of the sequence. It invokes a helper method *PrintPermutation*, which prints all permutations by recursively calling itself.

The following figure shows a part of the trace of recursive calls made by the execution of $Permutation(3, \emptyset, \text{XYZ})$.

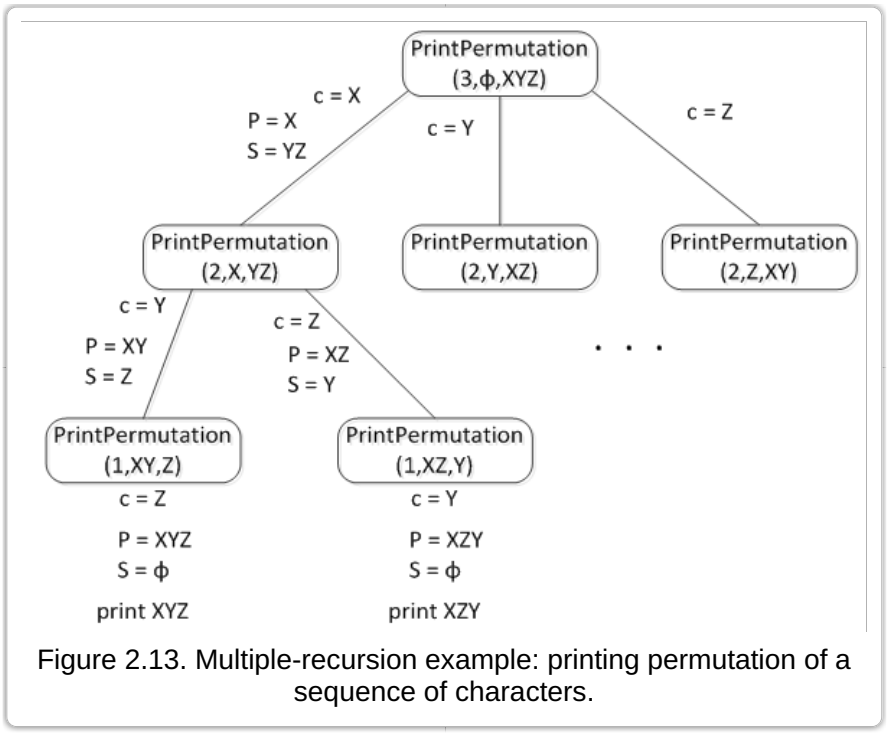


Figure 2.13. Multiple-recursion example: printing permutation of a sequence of characters.

Section 2.2.4. Designing Recursive Algorithms

A recursive algorithm includes two components:

- Base case—A recursive algorithm calls itself repeatedly and stops when a certain condition is met. This is usually referred to as the *base case*. In the *Permutation* example of Section 2.2.3.3, the base case occurs when $n = 1$. We have to ensure that the execution of each recursive call eventually reaches this base case so that it may not go into an infinite sequence of calls.
- Recursion—When the condition of the base case is not met, then the algorithm invokes itself recursively.

When designing a recursive algorithm, first we need to identify whether the given problem can be decomposed into smaller subproblems that have the same, repetitive structure as that of the initial problem.

Often, it is not easy to find a repetitive structure from the given problem. In such a case, we can use a helper method that has a repetitive structure. For example, the initial problem of *Permutation* has one input argument, which is a sequence of characters, and the signature of the algorithm would be *Permutation*(sequence S). But, it is not obvious how to design a recursive algorithm with that signature as it is. For this problem, we can use a helper method with additional parameters, with which it becomes easy to develop a recursive algorithm. The helper method that was used in Section 2.2.3.3 is *PrintPermutation*(*S.size*, *P*, *S*). Note that two additional parameters are used. The first parameter, which is the size of a sequence *S*, is used to test for a base case. The second parameter *P* will eventually hold each permutation that will be printed. Usually, a helper method is declared as a private method inside the class of the original problem.

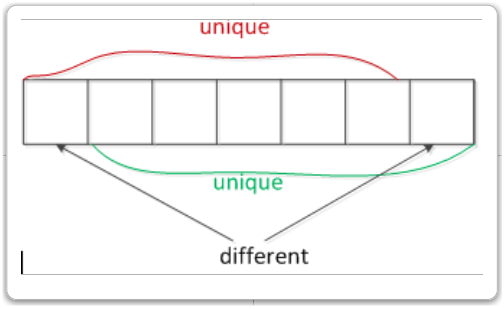
Section 2.2.5. Caution

We may encounter two problems when designing recursive algorithms. The first problem is that, if a recursive algorithm is poorly designed, its efficiency is very low. The second problem is called *infinite recursion*. A recursive algorithm continuously makes a recursive call and never stops.

We illustrate the first problem using the *element uniqueness problem* (discussed in Section 2.1.3.4) and the *Fibonacci numbers problem* (discussed in Section 1.2.1.3 in Module 1).

Element Uniqueness Problem

This problem is intended to determine whether all elements in the given array are distinct. Consider the following recursive algorithm that solves this problem. Given an array with n elements, we conclude that all elements are unique if (1) the first $n - 1$ elements are unique, (2) the last $n - 1$ elements are unique, and (3) the first and last elements are different from each other, as shown below:



Based on this observation, we can design the following recursive algorithm:

Code Segment 2.19

```
1   public static boolean unique3(int[] data, int low, int high) {
2     if (low >= high) return true;                       // at most one item
3     else if (!unique3(data, low, high-1)) return false;// duplicate in first n-1
4     else if (!unique3(data, low+1, high)) return false; // duplicate in last n-1
5     else return (data[low] != data[high]);          // do first and last differ?
6   }
```

Note that each invocation of *unique3* passes a subarray by passing its lower bound *low* and its upper bound *high*. The number of elements in the subarray is *n* = *high* – *low* + 1.

The problem with this design is that it is very inefficient. When *unique3* is invoked, it calls itself twice, in lines 3 and 4, with *n* – 1 elements, each of which makes two recursive calls with *n* – 2 elements; each of these makes two recursive calls with *n* – 3 elements, and so on. It terminates when there is only one element in the subarray (line 2). So, the number of method calls, in the worst case, is as follows:

$$1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1$$

So, the running time of *unique3* is $O(n^2)$, which is very inefficient.

Fibonacci Numbers Problem

Fibonacci sequence is defined recursively as follows:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F - n - 2 + F_{n-1} \text{ for } n > 1$$

A direct implementation of this recursive definition is shown below:

Code Segment 2.20

```
1   public static long fibonacciBad(int n) {
2     if (n <= 1)
3       return n;
4     else
5       return fibonacciBad(n-2) + fibonacciBad(n-1);
6   }
```

This implementation is very inefficient. When $fibonacciBad(n)$ is invoked, it invokes $fibonacciBad(n-2)$ and $fibonacciBad(n-1)$ in line 5. Note that $fibonacciBad(n-1)$ will invoke $fibonacciBad(n-2)$ again when it is executed. So, the same recursive call with the same argument is invoked multiple times. To illustrate this, let's assume that we want to calculate F_4 and we invoke $fibonacciBad(4)$. The trace of recursive calls is shown below:

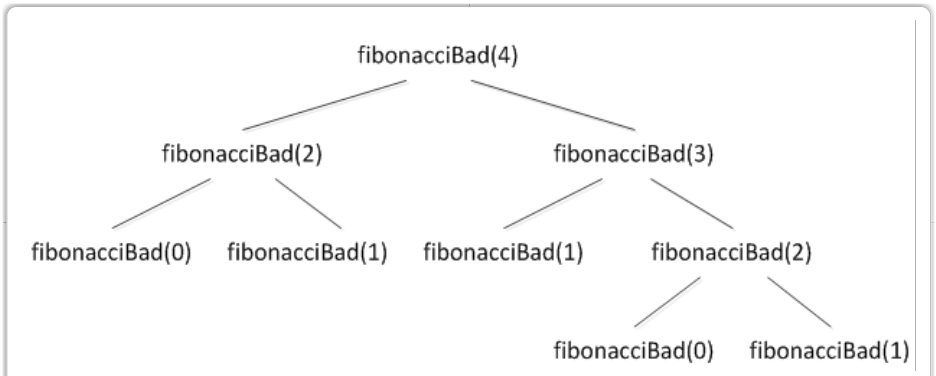


Figure 2.14. Recursive call trace for the call *fibonacciBad*(4).

We can see that $fibonacciBad(2)$ is invoked twice, $fibonacciBad(1)$ is invoked three times, and $fibonacciBad(0)$ is invoked twice. The total number of method invocations, including $fibonacciBad(3)$ and $fibonacciBad(4)$, is nine.

We can generalize the number of method invocations by enumerating it for each n as follows. Let C_n denote the number of calls made when $fibonacciBad(n)$ is invoked. Then, we have the following:

$C_0 = 1$
$C_1 = 1$
$C_2 = 1 + C_0 + C_1 = 3$
$C_3 = 1 + C_1 + C_2 = 5$
$C_4 = 1 + C_2 + C_3 = 9$
$C_5 = 1 + C_3 + C_4 = 15$
$C_6 = 1 + C_4 + C_5 = 25$
...

We can observe that, for every other n, the number of calls more than doubles. For example, C_4 is more than twice C_2, C_5 is more than twice C_3, C_6 is more than twice C_4, and so on. In general, we have $C_n > 2^{\frac{n}{2}}$. So, the running time of $fibonacciBad$ is an exponential function of n, which is extremely inefficient.

There is an efficient way of solving the *Fibonacci numbers problem*. We can design a recursive algorithm in such a way that each method invocation makes only one recursive call. This can be achieved by making a method to return two values, instead of a single value.

In $fibonacciBad$, F_n (a call to $fibonacciBad(n)$) returns a single value, which is F_n. In the new design, it will return two values, F_n and F_{n-1}, as an array of two elements. For consistency, we define $F_{-1} = 0$ (then we can say that F_0 returns F_0 and F_{-1}). A new recursive algorithm based on this observation is shown below:

Code Segment 2.21

```
1   public static long[] fibonacciGood(int n) {
2     if (n <= 1) {
3       long[] answer = {n, 0};
4       return answer;
5     } else {
6       long[] temp = fibonacciGood(n - 1);    // returns {Fn-1, Fn-2}
7       long[] answer = {temp[0] + temp[1], temp[0]};  // we want {Fn, Fn-1}
8       return answer;
9     }
10  }
```

In this design, $fibonacciGood(n)$ makes only one recursive call with one fewer element, $fibonacciGood(n-1)$. So, the method is invoked n times in total. Since all operations in the code except the recursive call, take a constant amount of time, the running time of $fibonacciGood$ is $O(n)$.

The second problem, *infinite recursion*, occurs if a recursive algorithm fails to implement or incorrectly implements the base-case test (a test for the stopping condition) of recursion. For example, the following code does not have a test for the base case and, thus, keeps calling itself. Theoretically, it never ends.

```
1   public static int Fibonacci(int n) {
2     return Fibonacci(n)
3   }
```

In typical recursive algorithms, there is a certain variable of which the value is changed with each recursive call. A test for the base case, then, checks whether the variable satisfies a predefined condition and, if it does, the algorithm terminates. For example, in the $factorial(n)$ method (refer to Section 2.2.1.1), n is decremented in each invocation of the method, and the method terminates when it becomes 0. If the variable is not changed in the method body or is incorrectly changed (for example, n gets incremented in the *factorial* method), then *infinite recursion* occurs.

So, programmers must make sure when implementing a recursive algorithm that the program will eventually terminate.

In actual programming-language systems, *infinite recursion* does not occur. Each invocation of a recursive call consumes CPU time and uses the call's memory space (which includes memory spaces for local variables, parameters, and return address, for example). So, *infinite recursion* would drain all system resources and, thus, is prevented by programming-language systems. In Java, if recursive calls are made more than a certain number of times (typically about 1,000 times), a *StackOverflowError* exception is thrown and the program is terminated.

Section 2.2.6. Eliminating Tail Recursion

An advantage of recursion is that, if a computational problem has an inherent recursive structure, a solution to the problem can be clearly and concisely described using a recursive algorithm. So, it can simplify algorithm analysis and avoid potential, complex program structures such as nested loops. However, as mentioned earlier, recursive programs incur overhead in terms of CPU time and memory space. If a program (or a part of a program) that which is implemented using recursion is executed frequently, then it may be desirable to convert it to an equivalent nonrecursive one. There is a standard technique for converting any recursive algorithm to an equivalent nonrecursive algorithm. Instead of this general technique, we discuss how to eliminate tail recursions.

A recursion is a *tail recursion* if a recursive call made in an invocation of the recursion is the last operation performed in that invocation. An example of a tail recursion is the method reversing the order of elements in an array, as discussed in Section 2.2.3. The Java code is copied below:

```
1   Algorithm reverseArray(data, low, high)
2     if low >= high, return
3     else
4       swap data[low] with data[high]
5       reverseArray(data, low+1, high-1)
```

In line 5, a recursive call is made and is the last operation.

Certain algorithms have the appearance of a tail recursion when actually they are not. An example is the *factorial* algorithm. The last statement of the algorithm is as follows:

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

So, a recursive call is made in the last statement. But, this is not a tail recursion because the recursive call is not the last *operation*. In this example, the last operation is that of multiplying *n* with the value returned by the recursive call.

Typically, a tail recursion is desirable because compilers can easily optimize it to use less stack space. Some compilers even automatically eliminate tail recursions by converting them to nonrecursive ones.

In this section, we illustrate the elimination of a tail recursion by converting the recursive version of the binary search method to a nonrecursive one that uses iteration. The code is shown below:

Code Segment 2.22

```
1   public static boolean binarySearchIterative(int[] data, int target) {
2     int low = 0;
3     int high = data.length - 1;
4     while (low <= high) {
5       int mid = (low + high) / 2;
6       if (target == data[mid])   // found a match
7         return true;
8       else if (target < data[mid])
9         high = mid - 1;         // only consider values left of mid
10      else
11       low = mid + 1;           // only consider values right of mid
12    }
13    return false;              // loop ended without success
14  }
```

Initially, *low* and *high* are set to the indexes of the first and last elements, respectively, of the entire array (in lines 2 and 3). These two indexes are updated in the subsequent *while* loop. In line 5, the index of the *middle* element is computed. Then the *middle* element is compared with the *target* and the following is performed. Case 1 (line 6): If they are identical, we have found the match and terminate. Case 2 (line 8): If *target* is smaller than the *middle* element, we update *low* and continue to the top of the *while* loop. In the next iteration, the left half of the array is searched. Case 3(line 10): If *target* is larger than the *middle* element, we update *high* and continue to the top of the *while* loop. In the next iteration, the right half of the array is searched. Note that, in the recursive version, Case 2 is implemented as a recursive call to the left half, and Case 3 is implemented as a recursive call to the right half. If we exit the

while loop, this means *target* was not found, and we return *false* (line 13).

As another example, a nonrecursive version of *reverseArray* is shown below:

Code Segment 2.23

```
1   public static void reverseIterative(int[] data) {
2     int low = 0, high = data.length - 1;
3     while (low < high) { // swap data[low] and data[high]
4       int temp = data[low];
5       data[low++] = data[high];  // post-increment of low
6       data[high--] = temp;       // post-decrement of high
7     }
8   }
```

Section 2.3 Stacks and Queues

Section 2.3. Stacks and Queues

Overview

Stacks and queue are the most fundamental and simplest data structures. They are also very important data structures that are used by many algorithms and applications.

Section 2.3.1. Stacks

A stack is a collection of objects with the following properties:

- A stack has a linear data structure.
- An object is added to one end of a stack and removed from the same end of the stack. This end is usually called the *stack top*, or simply the *top*.

Accordingly, when an object is removed, the object that was added last is removed first. Because of this, a stack is referred to as *LIFO* (*last-in, first-out*). Adding an object is called *push* and removing an object is called *pop*. The following figure illustrates the push and pop operations, which reverses the characters in the input:

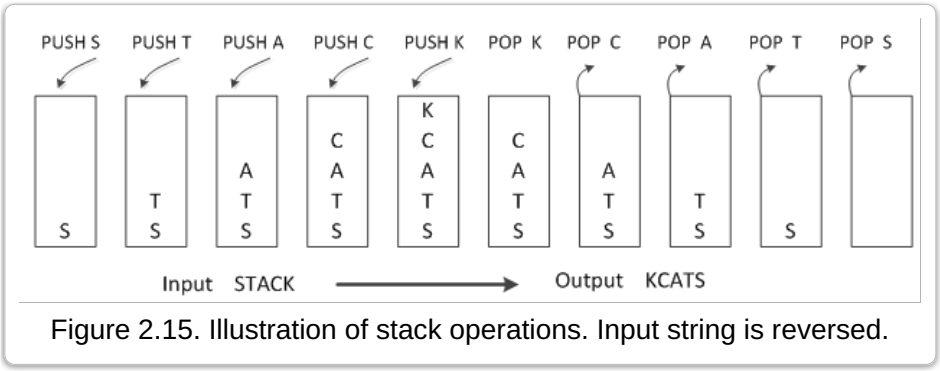


Figure 2.15. Illustration of stack operations. Input string is reversed.

Section 2.3.1.1. The Stack ADT

The stack ADT supports the following operations:

- push(*e*)—Adds element *e* to the stack top.

- pop( )—Removes and returns the top element from the stack. Returns *null* if the stack is empty.
- top( )—Returns the top element of the stack without removing it. Returns *null* if the stack is empty.
- size( )—Returns the number of elements in the stack.
- isEmpty( )—Returns *true* if the stack is empty and *false* otherwise.

The following table demonstrates stack operations. It is assumed that, in the *Stack Contents* column, the right side of a sequence is the stack top.

Figure 2.16. Demonstration of a Series of Stack Operations

Operation	Return Value	Stack Contents
push(10)	-	(10)
push(20)	-	(10, 20)
push(5)	-	(10, 20, 5)
size( )	3	(10, 20, 5)
top( )	5	(10, 20, 5)
pop( )	5	(10, 20)
push(30)	-	(10, 20, 30)
pop( )	30	(10, 20)
pop( )	20	(10)
pop( )	10	( )
isEmpty( )	true	( )
pop( )	null	( )

The Java interface for the stack ADT is shown below:

Code Segment 2.24

```
1   public interface Stack<E> {
2     int size();
3     boolean isEmpty();
4     void push(E e);
5     E top();
6     E pop();
7   }
```

Java provides a concrete implementation of the ADT in the *java.util.Stack* class. Java's stack class uses names for some operations that are different from the ones we used in the above ADT. The following table summarizes the differences:

Figure 2.17. Comparison of names used in the textbook ADT and those used in Java's *Stack* class.

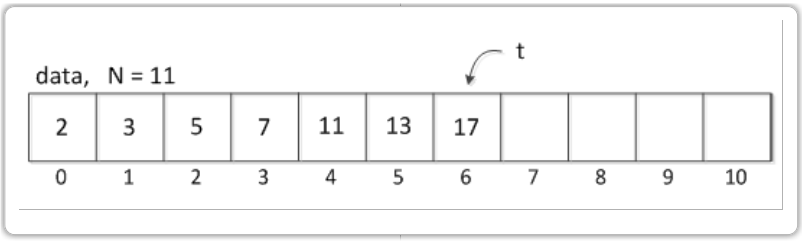
Stack ADT in Textbook	Class java.util.Stack
size( )	size( )

isEmpty( )	empty( )
push(*e*)	push(*e*)
pop( )	pop( )
top( )	peek( )

Section 2.3.1.2. Array-Based Stack Implementation

A stack's elements are stored in an array, *data*, with size *N*. The bottom element is stored in *data*[0], and the top element is stored in $data[t]$, $0 \leq t \leq N$. When the stack is empty, by convention, $t = -1$.

An example of an array-based stack is shown below:



The following code segment is a Java implementation of a stack, which uses an array for storage:

Code Segment 2.25

```
1   public class ArrayStack<E> implements Stack<E> {
2     public static final int CAPACITY=1000;   // default array capacity
3     private E[] data;                        // generic array used for storage
4     private int t = -1;                      // index of the top element in stack
5     public ArrayStack() { this(CAPACITY); }  // constructor with default capacity
6     public ArrayStack(int capacity) {        // constructor with given capacity
7       data = (E[]) new Object[capacity];
8     }
9     public int size() { return (t + 1); }
10    public boolean isEmpty() { return (t == -1); }
11    public void push(E e) throws IllegalStateException {
12      if (size() == data.length) throw new IllegalStateException("Stack is full");
13      data[++t] = e;              // increment t before storing new item
14    }
15    public E top() {
16      if (isEmpty()) return null;
17      return data[t];
18    }
19    public E pop() {
20      if (isEmpty()) return null;
21      E answer = data[t];
22      data[t] = null;                 // dereference to help garbage collection
23      t--;
24      return answer;
25    }
26  }
```

The array-based implementation is simple and very efficient. However, since the size of an array is fixed when created, the stack size is also limited. This problem can be resolved in two ways: (1) we can use a linked list as a storage, or (2) we can use a dynamic array, the size of which automatically increases

as more objects are added.

In the array-based implementation, all five methods run in $O(1)$ time, as summarized in the following table:

Figure 2.18. Performance of array-based implementation of a stack.

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
push(e)	$O(1)$
pop( )	$O(1)$
top( )	$O(1)$

In Java, when objects are not used by the program that created them anymore, a *garbage collection* mechanism automatically reclaims their storage space. So, a program does not need to explicitly deallocate storage of objects that are not used any more. However, it is a good practice to manually deallocate storage of such objects. In the *pop* method of the *ArrayStack* class, after the last element (stack top) in the *data* array is returned, *null* is assigned to that array slot, effectively deallocating the storage. This reduces a burden on the garbage collector and can increase overall system performance.

Section 2.3.1.3. Implementing a Stack with a Singly Linked List

In this section, we describe how to implement a stack using a singly linked list. An advantage is that there is no size limit any more because the list grows automatically as more objects are added.

The top element of a stack is stored at the front of the list. This makes stack operations efficient and all operations run in constant time.

A stack implemented using a singly linked list is illustrated below:

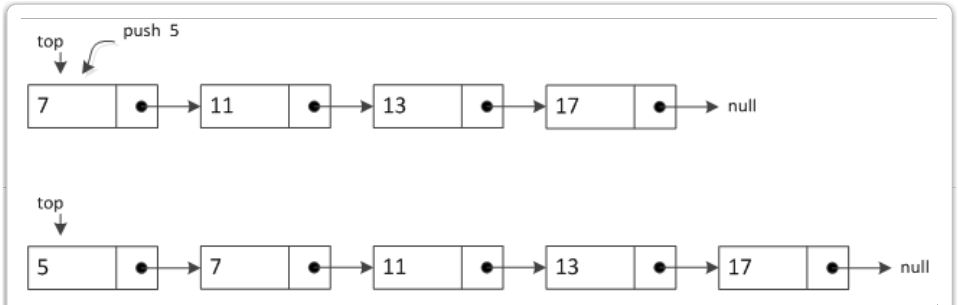


Figure 2.19. A stack is implemented using a singly linked list.

In this implementation, we use the singly linked list (discussed in Section 1.3.7) as the storage for a stack. Stack operations are implemented using the methods defined for the singly linked list. The following table shows the correspondence between the stack operations and the list operations:

Figure 2.20. Correspondence between stack operations and singly linked list operations.

Stack Method	Singly Linked-List Methods
size( )	list.size( )
isEmpty( )	list.isEmpty( )

push(*e*)	list.addFirst(*e*)
pop( )	removeFirst( )
top( )	list.first( )

A Java code of the implementation is shown below:

Code Segment 2.26

```
1   public class LinkedStack<E> implements Stack<E> {
2     private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list
3     public LinkedStack() { }  // new stack relies on the initially empty list
4     public int size() { return list.size(); }
5     public boolean isEmpty() { return list.isEmpty(); }
6     public void push(E element) { list.addFirst(element); }
7     public E top() { return list.first(); }
8     public E pop() { return list.removeFirst(); }
9     public String toString() {
10      return list.toString();
11    }
12  }
```

Section 2.3.1.4. Reversing an Array Using a Stack

Suppose we add a sequence of objects to a stack. When we remove those objects from the stack, they will come out in the reverse order. We can use this stack property to reverse elements in an array.

The following is a Java generic method that reverses array elements:

Code Segment 2.27

```
1   public static <E> void reverse(E[] a) {
2     Stack<E> buffer = new ArrayStack<>(a.length);
3     for (int i=0; i < a.length; i++)
4       buffer.push(a[i]);
5     for (int i=0; i < a.length; i++)
6       a[i] = buffer.pop();
7   }
```

In line 2, the variable *buffer* is declared as a stack of *ArrayStack* type. Since this is a generic method, we can create a stack of any type by passing that type as an actual type parameter. In the first *for* loop of lines 3 and 4, the elements in the array *a* are pushed into the stack *buffer*. In lines 5 and 6, the contents of *buffer* are popped and copied back to the array *a*, but in reverse order.

The *reverse* method is illustrated below. In the left column of the figure, elements are copied from the array *a* to the stack *buffer* through a series *push* operations. On the right, elements are popped from the stack and copied back to the array *a*. Note that, in the figure, the arrays on the right side have empty spaces. But actually, they are all filled with previous contents. They are now shown to make the effects of *pop* operations more obvious.

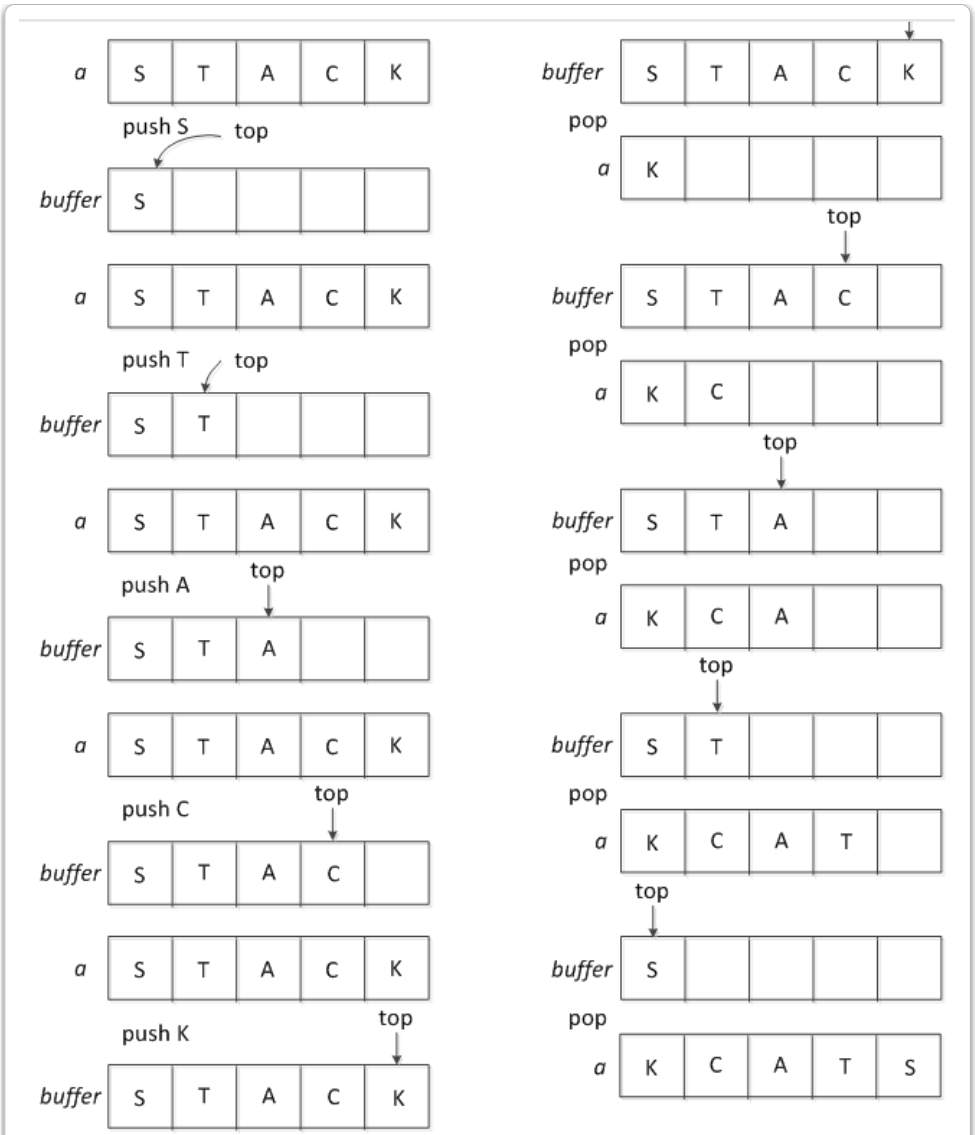


Figure 2.21. Reversing array elements using a stack.

Section 2.3.1.5. Matching Parentheses

In mathematical expressions, each opening parenthesis must be matched by a corresponding closing parenthesis. The following shows some matching pairs of parentheses:

- Parentheses: "(" and ")"
- Braces: "{" and "}"
- Brackets: "[" and "]"

We call these *delimiting symbols*. When we process an arithmetic expression, we first make sure that it is valid by checking that all delimiting symbols have matching counterparts, among other things. We briefly discuss how to check matching delimiters using a stack. Note that the algorithm described here is slightly different from the one in the textbook.

The sketch of the algorithm is given below:

- Scan the expression one character at a time from left to right.
- If the character is an opening delimiter, push it to the stack.
- If the character is a closing delimiter, do the following:
  - Pop a delimiter from the stack.
  - Compare that with the closing delimiter being scanned.

- If they are identical, continue.
- If they are different, the expression is invalid.

Note that, while the expression is being scanned, all other characters are ignored.

The following figure illustrates the algorithm:

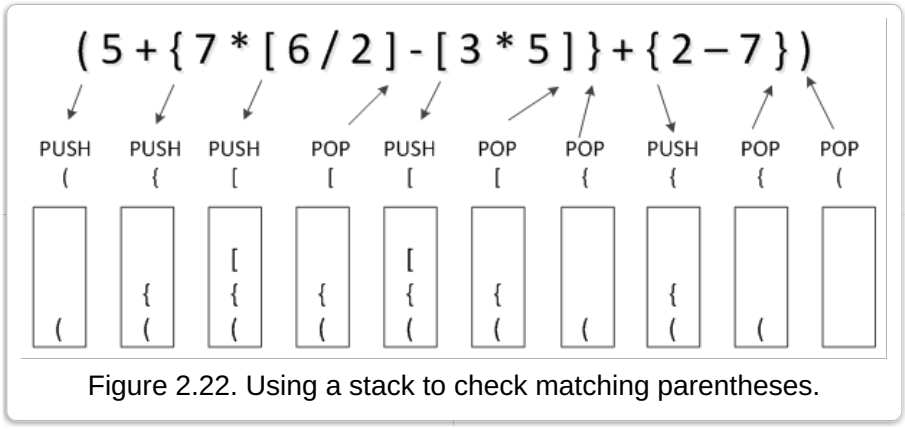


Figure 2.22. Using a stack to check matching parentheses.

Section 2.3.2. Queues

A queue is similar to a stack, but objects are added and removed in a different way. When an object is added, it is added to the *rear* of a queue. When an object is removed, it is taken from the *front* of a queue. This behavior is called *FIFO* (*first-in, first-out*). To add an object is referred to as to *enqueue*, and to remove an object is referred to as to *dequeue*. When we implement a queue, we usually (but not always) maintain two pointers: *front* and *rear*. The front pointer always references the first element in a queue, and the rear pointer always references the last element in a queue. The following figure illustrates the *enqueue* and *dequeue* operations:

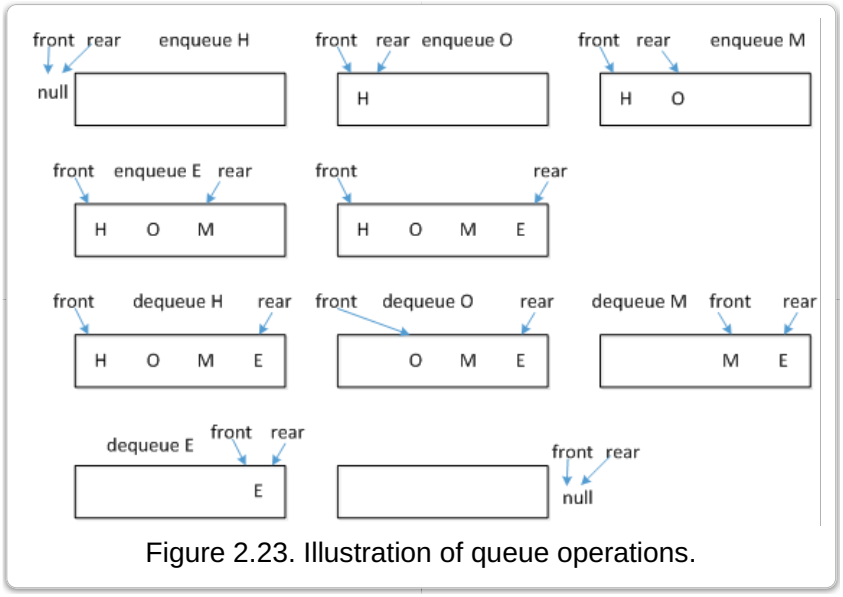


Figure 2.23. Illustration of queue operations.

Section 2.3.2.1. The Queue Abstract Data Type (ADT)

The queue ADT supports the following operations:

- enqueue(*e*)—Adds element *e* to the back of queue.
- dequeue( )—Remove and returns the first element from the queue. Returns *null* if the queue is empty.
- first( )—Returns the first element of the queue, without removing it. Returns *null* if the queue is empty.
- size( )—Returns the number of elements in the queue.

- isEmpty( )—Returns *true* if the queue is empty and *false* otherwise.

A Java interface of the queue ADT is shown below:

Code Segment 2.28

```
1   public interface Queue<E> {
2     int size();
3     boolean isEmpty();
4     void enqueue(E e);
5     E first();
6     E dequeue();
7   }
```

The following table demonstrates the queue operations:

Figure 2.24. Demonstration of a series of queue operations.

Operation	Return Value	first $\leftarrow Q \leftarrow$ last
enqueue(10)	-	(10)
enqueue(20)	-	(10, 20)
enqueue(5)	-	(10, 20, 5)
size( )	3	(10, 20, 5)
dequeue( )	10	(20, 5)
enqueue(30)	-	(20, 5, 30)
dequeue ( )	20	(5, 30)
dequeue( )	5	(30)
dequeue( )	30	( )
isEmpty( )	true	( )
enqueue( )	null	( )

Java provides the *java.util.Queue* interface to support queue operations. Java's queue interface has methods to implement all operations we discussed and includes some additional methods.

The following table compares the traditional queue ADT operations, which we discussed, and Java's methods:

Figure 2.25. Comparison of queue operations.

Queue ADT	Interface java.util.Queue	
	throws exception	returns special value
enqueue(*e*)	add(*e*)	offer(*e*)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )

size( )	size( )
isEmpty( )	isEmpty

As we can see in the table, Java's queue interface has two sets of methods for some operations. The differences in their behavior are described below:

- When a queue is empty, the following occurs:
  - The *remove* method throws a *NoSuchElementException* exception.
  - The *poll* method returns null.
  - The *element* method throws a *NoSuchElementException* exception.
  - The *peek* method returns null.
- When a queue is full (when implemented with bounded capacity), the following occurs:
  - The *add* method throws an *IllegalStateException* exception.
  - The *offer* method returns false.

Section 2.3.2.2. Array-Based Queue Implementation

In this section, we describe how the queue ADT can be implemented using an array. Objects are stored in the array *data* and the first element is at index 0. The following figure illustrates *enqueue* and *dequeue* operations. In the figure, f points to the first element.

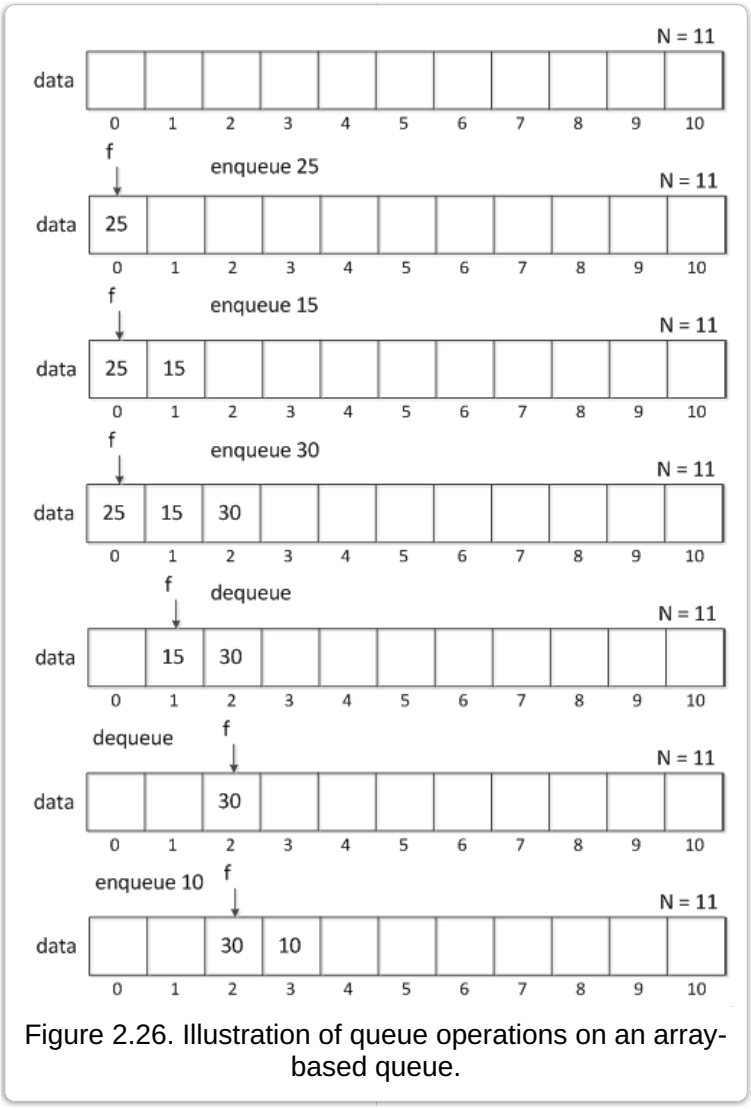
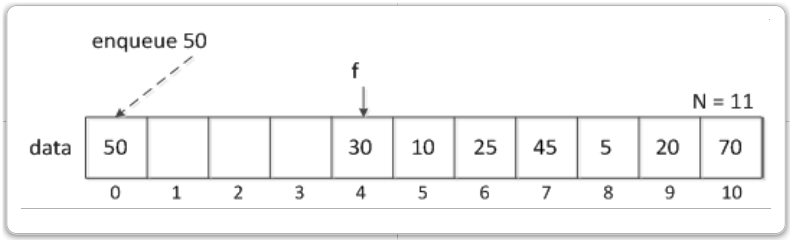


Figure 2.26. Illustration of queue operations on an array-based queue.

As we enqueue elements, the rear of the queue moves to the right and will eventually reach the end of the *data* array. When we enqueue one more

element, we would like it to be added to the beginning of the array if there is a space. In other words, we want to use the array in a circular manner. This is illustrated below. In the figure, a new element *50* is added to the queue. But there is no space at the end of the array. So, we go around the array and add it into the first slot of the array.



This can be implemented using the *modulo* operator. When we advance f or add a new element at rear of a queue, we first increment the current index and we apply the "modulo *N*" operation on that incremented index. In Java, the "%" symbol is the modulo operator. For example, suppose that *N* = 11, the current rear of a queue is 10 (so there is no more space at the end of the array), and a new element is being added. The index of the next slot in the array is calculated as (10 + 1) % *N* = 11 % 11 = 0. So, the new element is added to *data*[0].

A Java implementation of a queue ADT using an array in a circular fashion is presented in the *ArrayQueue* class. The following code segment shows the declaration part of the *ArrayQueue* class, along with instance variables and constructors:

Code Segment 2.29

```
1   public class ArrayQueue<E> implements Queue<E> {
2     public static final int CAPACITY = 1000;  // default array capacity
3     private E[] data;                         // generic array used for storage
4     private int f = 0;                        // index of the front element
5     private int sz = 0;                       // current number of elements
6     public ArrayQueue() {this(CAPACITY);} // constructs queue with default capacity
7     public ArrayQueue(int capacity) {     // constructs queue with given capacity
8       data = (E[]) new Object[capacity];
9     }
```

There are three instance variables:

- $data$—The array storing the elements of the queue
- f—The index of the front element of the queue
- sz—The number of current elements in the queue

The following is a Java implementation of the *enqueue* method. Note that, in line 3, the index of the next available slot is calculated in a circular way.

Code Segment 2.30

```
1   public void enqueue(E e) throws IllegalStateException {
2     if (sz == data.length) throw new IllegalStateException("Queue is full");
3     int avail = (f + sz) % data.length;   // use modular arithmetic
4     data[avail] = e;
5     sz++;
6   }
```

A Java code that implements the *dequeue* method is shown below. The next index of f is also calculated in a circular fashion in line 5.

Code Segment 2.31

```
1   public E dequeue() {
2     if (isEmpty()) return null;
```

```
3     E answer = data[f];
4     data[f] = null;   // dereference to help garbage collection
5     f = (f + 1) % data.length;
6     sz--;
7     return answer;
8   }
```

Both the *enqueue* and *dequeue* operations take $O(1)$ running time. Other queue operations also run in $O(1)$ time.

A complete Java code implementing the queue ADT using an array in a circular fashion can be found at the *ArrayQueue.java file*.

Section 2.3.2.3. Implementing a Queue with a Singly Linked List

As discussed earlier, a drawback of an array-based implementation is the fixed size of the array used to store queue elements. In this section, we show an implementation that uses a singly linked list as a storage (refer to Section 1.3.7 for *SinglyLinkedList*). The following is a Java code for the implementation:

Code Segment 2.32

```
1   public class LinkedQueue<E> implements Queue<E> {
2     private SinglyLinkedList<E> list = new SinglyLinkedList<>();
3     public LinkedQueue() { }
4     public int size() { return list.size(); }
5     public boolean isEmpty() { return list.isEmpty(); }
6     public void enqueue(E element) { list.addLast(element); }
7     public E first() { return list.first(); }
8     public E dequeue() { return list.removeFirst(); }
9   }
```

As shown above, methods in the class are implemented using the methods defined in the *SinglyLinkedList* class. All methods run in $O(1)$ time.

Section 2.3.2.4. A Circular Queue

The linked list-based implementation, described in the previous section, can be easily extended to implement a circular queue. First, we define both an interface that inherits all behavior of the *Queue* interface and an additional method *rotate*( ). This *rotate*( ) method moves the first element to the end of the list, effectively creating a circular queue. A Java code for the circular queue interface is shown below:

Code Segment 2.33

```
1   public interface CircularQueue<E> extends Queue<E> {
2     /**
3      * Rotates the front element of the queue to the back of the queue.
4      * This does nothing if the queue is empty.
5      */
6     void rotate();
7   }
```

Test Yourself 2.2

Consider the *LinkedQueue* class, whose definition is shown below:

Please think carefully, practice and write your own program, and then click "Show Answer" to compare yours to the suggested possible solution.

```
1   public class LinkedQueue<E> implements Queue<E> {
```

```
2     private SinglyLinkedList<E> list = new SinglyLinkedList<>();
3     public LinkedQueue() { }
4     public int size() { return list.size(); }
5     public boolean isEmpty() { return list.isEmpty(); }
6     public void enqueue(E element) { list.addLast(element); }
7     public E first() { return list.first(); }
8     public E dequeue() { return list.removeFirst(); }
9   }
```

Note that the *LinkedQueue* is discussed in section 2.3.2.3 and it is also described in page 245 of the textbook.

Write a Java method with signature *concatenate*(*LinkedQueue*<*E*> *Q*2) that takes all elements of *Q*2 and appends them to the end of the original queue. The operations should run in *O*(1) time and *Q*2 must be empty after the execution of the method.

Answer – One possible solution:

```
public void concatenate(SinglyLinkedList<E> Q2) {
  if (head == null) {
    head = Q2.head;
  } else {
    tail.setNext(Q2.head);
  }
  tail = Q2.tail;
  size += Q2.size;
  // clear Q2
  Q2.head = Q2.tail = null;
  Q2.size = 0;
}
```

Section 2.3.3. Double-Ended Queues

In this section, we describe another queue-like data structure called *deque*, which is more flexible than either a stack or a queue.

A *deque*, pronounced as "deck" to avoid the confusion with the *dequeuer* operation, is a data structure that allows insertion and deletion at both the front and rear of a queue. The deque ADT support the following operations:

- addFirst(*e*)—Inserts a new element *e* at the front of the deque.
- addLast(*e*)—Inserts a new element *e* at the rear of the deque.
- removeFirst()—Removes and returns the first element of the deque. Returns *null* if the deque is empty.
- removeLast()—Removes and returns last element of the deque. Returns *null* if the deque is empty.
- first( )—Returns the first element of the deque, without removing it. Returns *null* if the deque is empty.
- last( )—Returns the last element of the deque, without removing it. Returns *null* if the deque is empty.
- size( )—Returns the number of elements in the deque.
- isEmpty( )—Returns *true* if the deque is empty and *false* otherwise.

A Java interface formalizing the deque ADT is shown below:

Code Segment 2.34

```
1   public interface Deque<E>{
2     int size();
3     boolean isEmpty();
4     E first();
5     E last();
6     void addFirst(E e);
7     void addLast(E e);
```

```
8     E removeFirst();
9     E removeLast();
10  }
```

The following table demonstrate some of the deque operations:

Figure 2.27. Demonstration of a Series of deque Operations

Method	Return Value	D
addLast(10)		(10)
addFirst(5)		(5, 10)
addFirst(30)		(30, 5, 10)
addLast(50)		(30, 5, 10, 50)
fitst( )	30	(30, 5, 10, 50)
last( )	50	(30, 5, 10, 50)
removeFirst( )	30	(5, 10, 50)
removeLast( )	50	(5, 10)
size( )	2	(5, 10)
isEmpty( )	false	(5, 10)

We can implement a deque using a circular array or a doubly linked list. Of the two, a doubly linked list is more appropriate because it naturally supports insertion and deletion at both ends.

The doubly linked list (*DoublyLinkedList.java*), which we discussed in Section 1.3.9 of Module 1, has already implemented all methods in the *Deque* interface. It is only necessary to add "implements Deque<E>" to the declaration part of the *DoublyLinkedList* class definition.

Java defines its own deque in the *java.util.Deque* interface. The *java.util.ArrayDeque* is a concrete class implementing the *Deque* interface using a resizable array. The *java.util.LinkedList* class also implements the *Deque* interface, as well as the *List* interface.

The following table compares the methods in the deque ADT operations described in this section with the methods in Java's *Deque* interface:

Figure 2.28. Comparison of deque Operations

Deque ADT	Interface java.util.Deque	
	throws exceptions	returns special value
first( )	getFirst( )	peekFirst( )
last( )	getLast( )	peekLast( )
addFirst(*e*)	addFirst(*e*)	offerFirst(*e*)
addLast(*e*)	addLast(*e*)	offerLast(*e*)
removeFirst( )	removeFirst( )	pollFirst( )
removeLast( )	removeLast( )	pollLast( )

size( )	size( )
isEmpty( )	isEmpty( )

Java provides two sets of methods for some operations. The differences are described below:

- When a deque is empty, the following occurs:
  - The *getFirst*, *getLast*, *removeFirst*, and *removeLast* methods throw a *NoSuchelementException*.
  - The *peekFirst*, *peekLast*, *pollFirst*, and *pollLast* methods return null.
- When a deque is full (implemented with a capacity limit), the following occurs:
  - The *addFirst* and *addLast* methods throw an *IllegalStateException*.
  - The *offerFirst* and *offerLast* methods return false.

Module 2 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer or write your own program first, and then click "Show Answer" to compare yours to the suggested answer or the possible solution.

Test Yourself 2.3

Arrange the following functions in increasing order of running times:

$$n^2, \log n, n \log n, a^n, n, n^3, constant$$

Suggested answer: $constant, \log n, n, n \log n, n^2, n^3, a^n$

Test Yourself 2.4

Express each of the following functions using a *big-oh*:

Please think carefully about each function, respond, and click each one to compare yours to its suggested answer.

▶ **(1)** $T(n) = 7n^3 - 4n^2 + 5n - 10$

Suggested answer: $T(n) = O(n^3)$

▶ **(2)** $T(n) = 8 \log n - n \log (n + 2)$

Suggested answer: $T(n) = O(n \log n)$

▶ **(3)** $T(n) = 3n^3 + 4^{2n}$

Suggested answer: $T(n) = O(c^n)$, where c is a constant.

▶ **(4)** $T(n) = 4n + \log (n - 5)$

Suggested answer: $T(n) = O(n)$

▶ **(5)** $T(n) = \log (4n + 3) + c$, **where c is a constant**

Suggested answer: $T(n) = O(\log n)$

Test Yourself 2.5

The number of operations executed by algorithms A and B is $8n \log n$ and $2n^2$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

Suggested answer:

$8n \log n \leq 2n^2$

$4\log n \leq n$

This inequality is always true for $n \geq 16$. So, $n_0 = 16$.

Test Yourself 2.6

Express the running time of the following method using a *big-oh* notation:

```
public static int method1(int[] a) {
int n = a.length;
int total = 0;
for (int j=0; j<n; j += 2)
  total += a[j];
return total;
}
```

Suggested answer: $O(n)$

Test Yourself 2.7

Express the running time of the following method using a *big-oh* notation:

```
public static int method1(int[] a, int[] b){
int n = a.length;
int count = 0;
for (int i=0; i<n; i++) {
  int total = 0;
  for (int j=0; j<n; j++)
    for (int k=0; k<=j; k++)
      total += a[k]
  if (b[i] == total) count++
}
return count;
}
```

Suggested answer: $O(n^3)$

Test Yourself 2.8

Prove that the sum of two odd integers is an even integer using the direct proof method.

Suggested answer:

Let x and y be two odd integers. Let $z = x + y$.

Since x is odd, it can be rewritten as $x = 2n + 1$, for some integer n.

Since y is odd, it can be rewritten as $y = 2m + 1$, for some integer m.

Then, $z = x + y = 2n + 1 + 2m + 1 = 2(n + m + 1)$. So, z is even.

Test Yourself 2.9

Prove that $1 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ for all positive n, using the induction method.

Suggested answer:

Base case:

Assume it is true for $n = k$, i.e., $1 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$

Show it is also true for $n = k + 1$, i.e., $1 + 2^1 + 2^2 + \cdots + 2^k + 2^{k+1} = 2^{k+2} - 1$

$$\begin{aligned}\text{LHS} &= 1 + 2^1 + 2^2 + \cdots + 2^k + 2^{k+1} \\ &= 2^{k+1} - 1 + 2^{k+1} \\ &= 2 \times 2^{k+1} - 1 \\ &= 2^{k+2} - 1 = \text{RHS}\end{aligned}$$

Test Yourself 2.10

Write a recursive Java method that computes the product of two positive integers m and n, using only addition and subtraction.

Answer - one possible solution:

```
int productRecursive(int n, int m){
  if (m == 1) return n;
  return n + productRecursive(n, m-1);
}
```

Test Yourself 2.11

Write a recursive Java method that computes the integer part of the base-two logarithm of n, using only addition and subtraction.

Answer - one possible solution:

```
int logTwoInteger(int n){
      if (n < 2) return 0;
      return 1 + logTwoInteger(n/2);
}
```

Here, the *greaterOfTwo* method returns the greater of the two arguments.

Test Yourself 2.12

Write a recursive Java method that finds the maximum element in an array *A* of *n* elements.

Answer - one possible solution:

```
int recursiveMax(int[] a){
    if (a.length == 1) return a[0];
    int[] temp = Arrays.copyOf(a, a.length-1);
    return greaterOfTwo(a[a.length-1], recursiveMax(temp));
}
```

Test Yourself 2.13

Write a recursive Java method that receives a character string and outputs its reverse. For example, if an input argument is "abc" then "cba" is the output.

Answer - one possible solution:

```
void printReverse(String s, int n) {
  if (n >= 0) {
  System.out.print(s.charAt(n))
  printReverse(s, n-1);
  }
}

void printReverse(String s) {
  printReverse(s, s.length( ) - 1);
```

```
}
```

Test Yourself 2.14

Rewrite the following method without using recursion:

```
double power(double x, int n){
  if (n == 0) return 1;
  else return x * power(x, n-1);
}
```

Answer - one possible solution:

```
public static double iterativePower(double x, int n){
  double result = x;
  while (n > 1){
    result *= x;
     n--;
  }
  return result;
}
```

Test Yourself 2.15

Is the following recursive method a *tail recursion*?

```
double power(double x, int n){
  if (n == 0) return 1;
  else return x * power(x, n-1);
}
```

Suggested answer: No, because there is an additional multiplication in the last statement.

Test Yourself 2.16

Consider the stack ADT discussed in Section 2.3.1.1 (the stack ADT is also described in page 227 of the textbook). Suppose that the following operations are performed on an initially empty stack *S*: 12 *push* operations, 5 *top* operations, an 4 *pop* operations, 2 of which returned *null*. What is the size of the resulting stack *S*?

Suggested answer: 12 – 4 + 2 = 10.

Test Yourself 2.17

Write a generic Java method with signature *transfer*(*Stack* <*E*> *S*, *Stack* <*E*> *T*) that transfers all elements from stack *S* to stack *T*, so that element that starts at the top of *S* is the first to be inserted on to *T*, and the last element at the bottom of *S* ends up at the top of *T*.

Answer - one possible solution:

```
static <E> void transfer(Stack<E> S, Stack<E> T){
  while (!S.isEmpty( )){
  T.push(S.pop( ));
  }
}
```

Test Yourself 2.18

Consider the queue ADT discussed in Section 2.3.2.1 (the queue ADT is also described in page 239 of the textbook). Show the sequence of values returned by the following sequence of queue operations. Assume that the queue is initially empty.

enqueue(10), enqueue(7), dequeue(), enqueue(21), dequeue(), dequeue(), dequeue(), enqueue(4), enqueue(5), dequeue(), enqueue(25), dequeue()

Suggested answer: none, none, 10, none, 7, 21, null, none, none, 4, none, 5

Test Yourself 2.19

Suppose there are three nonempty stacks *R*, *S*, and *T*. Show a sequence of stack operations that will move all elements in *T* below all of *S*'s original elements. Order of elements in both T and S should be the same in the resulting *S*. You may use *R* as a temporary storage but the original elements in *R* should remain unchanged in the original order. For example, if *R* = (1, 2, 3), S = (4, 5), and *T* = (6, 7, 8, 9), when ordered from bottom to top, then the final stack contents should be *R* = (1, 2, 3), *S* = (6, 7, 8, 9, 4, 5), *T* = ( ).

Suggested answer:

Let r, s and t denote the original sizes of the stacks.

```
Make s calls to R.push(S.pop( ))
Make t calls to R.push(T.pop( ))
Make s+t calls to S.push(R.pop( ))
```

Test Yourself 2.20

Java's *java.util.Deque* specifies a deque ADT. It provides duplicative methods for some operations. Describe the difference between *addFirst*(*e*) and *offerFirst*(*e*).

Suggested answer: When a deque is full, the *addFirst* method throws *IllegalStateException* and the *offerFirst* method returns false.

References

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.) Wiley.
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). *The Java® language specification, Java SE 8 edition.* Oracle America Inc.
- Oracle. *The Java Tutorials*.