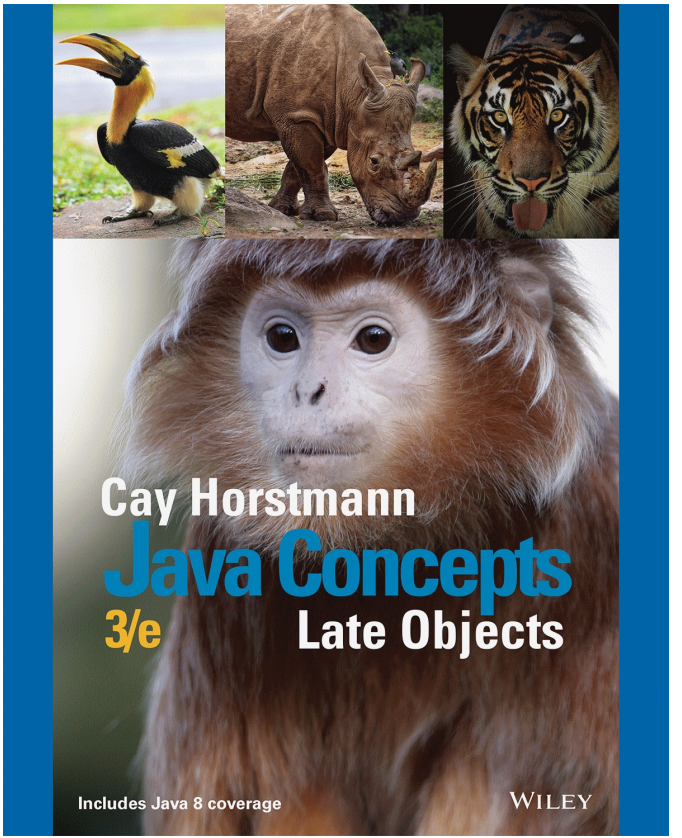
Chapter 13 - Recursion



Chapter Goals

- To learn to “think recursively”
- To be able to use recursive helper methods
- To understand the relationship between recursion and iteration
- To understand when the use of recursion affects the efficiency of an algorithm
- To analyze problems that are much easier to solve by recursion than by iteration
- To process data with recursive structures using mutual recursion

Triangle Numbers

- **Recursion**: the same computation occurs repeatedly.
- Using the same method as the one in this section, you can compute the volume of a Mayan pyramid.

- Problem: to compute the area of a triangle of width *n*
- Assume each [ ] square has an area of 1

- Also called the n^{th} *triangle number*
- The third triangle number is 6

```
[]
[][]
[][][]
```

Outline of Triangle Class

```
public class Triangle
{
   private int width;

   public Triangle(int aWidth)
   {
      width = aWidth;
   }
   public int getArea()
   {
      . . .
   }
}
```

Handling Triangle of Width 1

- The triangle consists of a single square.
- Its area is 1.
- Add the code to `getArea` method for width 1:

```
public int getArea()
{
   if (width == 1) { return 1; }
   . . .
}
```

Handling the General Case

- Assume we know the area of the smaller, colored triangle:

```
[]
[][]
[][][]
[][][][]
```

- Area of larger triangle can be calculated as: `smallerArea + width`
- To get the area of the smaller triangle:
  - Make a smaller triangle and ask it for its area:

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Completed getArea Method

```
public int getArea()
{
   if (width == 1) { return 1; }
   Triangle smallerTriangle = new Triangle(width - 1);
   int smallerArea = smallerTriangle.getArea();
   return smallerArea + width;
}
```

A recursive computation solves a problem by using the solution to the same problem with simpler inputs.

Computing the area of a triangle with width 4

- `getArea` method makes a smaller triangle of width 3
- It calls `getArea` on that triangle
  - That method makes a smaller triangle of width 2
  - It calls `getArea` on that triangle
    - That method makes a smaller triangle of width 1
    - It calls `getArea` on that triangle
      - That method returns 1
    - The method returns `smallerArea + width = 1 + 2 = 3`
  - The method returns `smallerArea + width = 3 + 3 = 6`
- The method returns `smallerArea + width = 6 + 4 = 10`

Recursion

- A recursive computation solves a problem by using the solution of the same problem with simpler values.
- Two key requirements for successful recursion:
  - Every recursive call must simplify the computation in some way
  - There must be special cases to handle the simplest computations directly
- To complete our `Triangle` example, we must handle width <= 0:

```
if (width <= 0) return 0;
```

Other Ways to Compute Triangle Numbers

- The area of a triangle equals the sum:

```
1 + 2 + 3 + . . . + width
```

- Using a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
   area = area + i;
```

- Using math:

```
1 + 2 + . . . + n = n × (n + 1)/2
=> area = width * (width + 1) / 2
```

section_1/Triangle.java

```
1  /**
2     A triangular shape composed of stacked unit squares like this:
3     []
4     [][]
5     [][][]
6     . . .
7  */
8  public class Triangle
```

section_1/TriangleTester.java

```
1  public class TriangleTester
2  {
3     public static void main(String[] args)
4     {
5        Triangle t = new Triangle(10);
6        int area = t.getArea();
7        System.out.println("Area: " + area);
8        System.out.println("Expected: 55");
9     }
```

Program Run:

```
Area: 55
Expected: 55
```

Self Check 13.1

Why is the statement `else if (width == 1) { return 1; }` in the `getArea` method unnecessary?

Answer: Suppose we omit the statement. When computing the area of a triangle with width 1, we compute the area of the triangle with width 0 as 0, and then add 1, to arrive at the correct area.

Self Check 13.2

How would you modify the program to recursively compute the area of a square?

Answer: You would compute the smaller area recursively, then return

`smallerArea + width + width - 1`.

```
[][][][]
[][][][]
[][][][]
[][][][]
```

Of course, it would be simpler to compute the area simply as width * width. The results are identical because

$$1 + 0 + 2 + 1 + 3 + 2 + \cdots + n + n - 1 = \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2$$

Self Check 13.3

In some cultures, numbers containing the digit 8 are lucky numbers. What is wrong with the following method that tries to test whether a number is lucky?

```
public static boolean isLucky(int number)
{
   int lastDigit = number % 10;
   if (lastDigit == 8) { return true; }
   else
   {
      return isLucky(number / 10); // Test the number without the last digit
   }
}
```

Answer: There is no provision for stopping the recursion. When a number < 10 isn't 8, then the method should return false and stop.

Self Check 13.4

In order to compute a power of two, you can take the next-lower power and double it. For example, if you want to compute 2^{11} and you know that $2^{10} = 1024$, then $2^{11} = 2 \times 1024 = 2048$. Write a recursive method `public static int pow2(int n)` that is based on this observation.

Answer:

```
public static int pow2(int n)
{
   if (n <= 0) { return 1; } // 20 is 1
   else { return 2 * pow2(n - 1); }
}
```

Self Check 13.5

Consider the following recursive method:

```
public static int mystery(int n)
{
   if (n <= 0) { return 0; }
   else
   {
      int smaller = n - 1;
      return mystery(smaller) + n * n;
   }
}
```

What is `mystery(4)`?

Answer:

```
mystery(4) calls mystery(3)
   mystery(3) calls mystery(2)
      mystery(2) calls mystery(1)
         mystery(1) calls mystery(0)
            mystery(0) returns 0.
         mystery(1) returns 0 + 1 * 1 = 1
     mystery(2) returns 1 + 2 * 2 = 5
   mystery(3) returns 5 + 3 * 3 = 14
mystery(4) returns 14 + 4 * 4 = 30
```

Tracing Through Recursive Methods

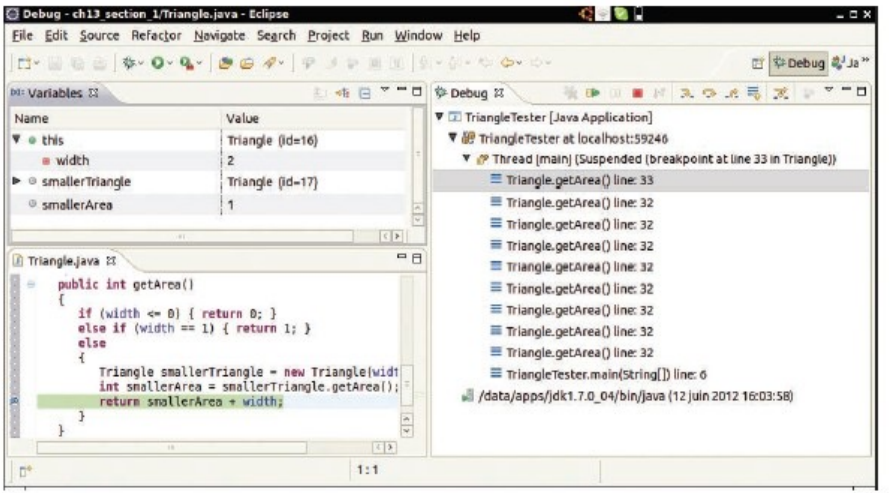


Figure 1 Debugging a Recursive Method

To debug recursive methods with a debugger, you need to be particularly careful, and watch the call stack to understand which nested call you currently are in.cc

Thinking Recursively

© Nikada/iStockphoto.

Thinking recursively is easy if you can recognize a subtask that is similar to the original task.

- Problem: test whether a sentence is a palindrome
- Palindrome: a string that is equal to itself when you reverse all characters
  - A man, a plan, a canal – Panama!
  - Go hang a salami, I'm a lasagna hog
  - Madam, I'm Adam

Implement isPalindrome Method: How To 13.1

```
public class Sentence
{
   private String text;
   /**
      Constructs a sentence.
      @param aText a string containing all characters of the sentence
   */
   public Sentence(String aText)
   {
       text = aText;
   }
   /**
      Tests whether this sentence is a palindrome.
      @return true if this sentence is a palindrome, false otherwise
   */
   public boolean isPalindrome()
   {
      . . .
   }
}
```

Thinking Recursively: How To 13.1

1. Consider various ways to simplify inputs.
Here are several possibilities:

- Remove the first character.
- Remove the last character.
- Remove both the first and last characters.
- Remove a character from the middle.
- Cut the string into two halves.

Thinking Recursively: How To 13.1

2. Combine solutions with simpler inputs into a solution of the original problem.

- Most promising simplification: Remove first and last characters
  adam, I'm Ada, is a palindrome too!
- Thus, a word is a palindrome if
  - The first and last letters match, and
  - Word obtained by removing the first and last letters is a palindrome
- What if first or last character is not a letter? Ignore it.
  - If the first and last characters are letters, check whether they match;
    if so, remove both and test shorter string
  - If last character isn't a letter, remove it and test shorter string
  - If first character isn't a letter, remove it and test shorter string

Thinking Recursively: How To 13.1

3. Find solutions to the simplest inputs.

- Strings with two characters
  - No special case required; step two still applies
- Strings with a single character
  - They are palindromes
- The empty string
  - It is a palindrome

Thinking Recursively: How To 13.1

4. Implement the solution by combining the simple cases and the reduction step:

```
public boolean isPalindrome()
{
   int length = text.length();
   // Separate case for shortest strings.
   if (length <= 1) { return true; }
   // Get first and last characters, converted to lowercase.
   char first = Character.toLowerCase(text.charAt(0));
   char last = Character.toLowerCase(text.charAt(length - 1));
   if (Character.isLetter(first) && Character.isLetter(last))
   {
      // Both are letters.
      if (first == last)
      {
         // Remove both first and last character.
         Sentence shorter = new Sentence(text.substring(1, length - 1));
         return shorter.isPalindrome();
      }
      else
      {
         return false;
      }
   }
   else if (!Character.isLetter(last))
   {
      // Remove last character.
```

```
      Sentence shorter = new Sentence(text.substring(0, length - 1));
      return shorter.isPalindrome();
   }
   else
   {
      // Remove first character.
      Sentence shorter = new Sentence(text.substring(1));
      return shorter.isPalindrome();
   }
}
```

Recursive Helper Methods

- Sometimes, a task can be solved by handing it off to a recursive helper method.
- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

- Consider the palindrome test of previous slide.
  - It is a bit inefficient to construct new Sentence objects in every step
- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:

```
/**
   Tests whether a substring of the sentence is a palindrome.
   @param start the index of the first character of the substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
public boolean isPalindrome(int start, int end)
```

Recursive Helper Methods: isPalindrome

```
public boolean isPalindrome(int start, int end)
{
   // Separate case for substrings of length 0 and 1.
   if (start >= end) { return true; }
   // Get first and last characters, converted to lowercase.
   char first = Character.toLowerCase(text.charAt(start));
   char last = Character.toLowerCase(text.charAt(end));
   if (Character.isLetter(first) && Character.isLetter(last))
   {
      if (first == last)
      {
         // Test substring that doesn't contain the matching letters.
         return isPalindrome(start + 1, end - 1);
      }
      else
      {
         return false;
      }
   }
   else if (!Character.isLetter(last))
   {
      // Test substring that doesn't contain the last character.
      return isPalindrome(start, end - 1);
   }
   else
   {
      // Test substring that doesn't contain the first character.
      return isPalindrome(start + 1, end);
   }
   }
```

Recursive Helper Methods

- Provide a method to call the helper method with positions that test the entire string:

```
public boolean isPalindrome()
{
   return isPalindrome(0, text.length() - 1);
}
```

- This call is **not** recursive

  The `isPalindrome(String)` method calls the helper method `isPalindrome(String, int, int)`.

  An example of overloading

- The public will call `isPalindrome(String)` method.
- `isPalindrome(String, int, int)` is the recursive helper method.

Self Check 13.6

Do we have to give the same name to both `isPalindrome` methods?

Answer: No — the second one could be given a different name such as `substringIsPalindrome`.

Self Check 13.7

When does the recursive `isPalindrome` method stop calling itself?

Answer: When `start >= end`, that is, when the investigated string is either empty or has length 1.

Self Check 13.8

To compute the sum of the values in an array, add the first value to the sum of the remaining values, computing recursively. Of course, it would be inefficient to set up an actual array of the remaining values. Which recursive helper method can solve the problem?

Answer: A `sumHelper(int[] a, int start, int size)`. The method calls `sumHelper(a, start + 1, size)`.

Self Check 13.9

How can you write a recursive method `public static void sum(int[] a)` without needing a helper function?

Answer: Call `sum(a, size - 1)` and add the last element, `a[size - 1]`.

The Efficiency of Recursion: Fibonacci Sequence

- Fibonacci sequence is a sequence of numbers defined by:

  $f_1 = 1$

  $f_2 = 1$

  $f_n = f_{n-1} + f_{n-2}$

- First ten terms:
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55

section_3/RecursiveFib.java

```
1  import java.util.Scanner;
2
3  /**
4     This program computes Fibonacci numbers using a recursive method.
5  */
6  public class RecursiveFib
7  {
8     public static void main(String[] args)
9     {
```

Program Run:

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

The Efficiency of Recursion

- Recursive implementation of `fib` is straightforward.
- Watch the output closely as you run the test program.
- First few calls to `fib` are quite fast.
- For larger values, the program pauses an amazingly long time between outputs.
- To find out the problem, lets insert **trace messages**.

section_3/RecursiveFibTracer.java

```
1  import java.util.Scanner;
2
3  /**
4     This program prints trace messages that show how often the
5     recursive method for computing Fibonacci numbers calls itself.
6  */
7  public class RecursiveFibTracer
8  {
9     public static void main(String[] args)
```

Program Run:

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
```

```
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

Call Tree for Computing `fib(6)`

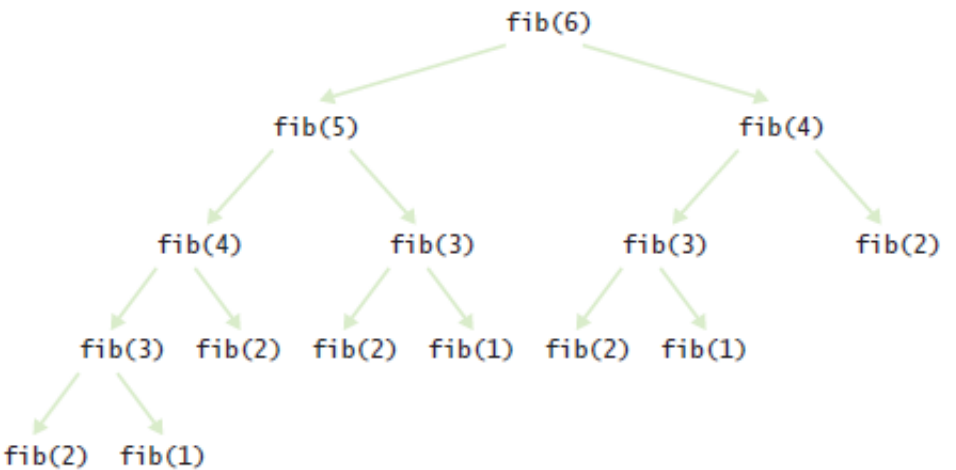


Figure 2 Call Pattern of the Recursive `fib` Method

The Efficiency of Recursion

- Method takes so long because it computes the same values over and over.
- The computation of `fib(6)` calls `fib(3)` three times.
- Imitate the pencil-and-paper process to avoid computing the values more than once.

section_3/LoopFib.java

```
1  import java.util.Scanner;
2
3  /**
4     This program computes Fibonacci numbers using an iterative method.
5  */
6  public class LoopFib
7  {
8     public static void main(String[] args)
9     {
```

Program Run:

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

The Efficiency of Recursion

© pagadesign/iStockphoto.

- In most cases, the iterative and recursive approaches have comparable efficiency.
- Occasionally, a recursive solution runs much slower than its iterative counterpart.
- In most cases, the recursive solution is only slightly slower.
- The iterative `isPalindrome` performs only slightly better than recursive solution.
  - Each recursive method call takes a certain amount of processor time

- Smart compilers can avoid recursive method calls if they follow simple patterns.
- Most compilers don't do that.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Iterative is Palindrome Method

```
public boolean isPalindrome()
{
    int start = 0;
   int end = text.length() - 1;
   while (start < end)
   {
      char first = Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end);
      if (Character.isLetter(first) && Character.isLetter(last))
      {
         // Both are letters.
         if (first == last)
         {
            start++;
            end--;
         }
         else
         {
            return false;
         }
      }
      if (!Character.isLetter(last)) { end--; }
      if (!Character.isLetter(first)) { start++; }
   }
    return true;
}
```

Self Check 13.10

Is it faster to compute the triangle numbers recursively, as shown in Section 13.1, or is it faster to use a loop that computes `1 + 2 + 3 + . . . + width`?

Answer: The loop is slightly faster. Of course, it is even faster to simply compute `width * (width + 1) / 2`.

Self Check 13.11

You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times \ldots \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n - 1)! \times n$. Is the recursive approach inefficient in this case?

Answer: No, the recursive solution is about as efficient as the iterative approach. Both require $n - 1$ multiplications to compute $n!$.

Self Check 13.12

To compute the sum of the values in an array, you can split the array in the middle, recursively compute the sums of the halves, and add the results. Compare the performance of this algorithm with that of a loop that adds the values.

Answer: The recursive algorithm performs about as well as the loop. Unlike the recursive Fibonacci algorithm, this algorithm doesn't call itself again on the same input. For example, the sum of the array 1 4 9 16 25 36 49 64 is computed as the sum of 1 4 9 16 and 25 36 49 64, then as the sums of 1 4, 9 16, 25 36, and 49 64, which can be computed directly.

Permutations

- Using recursion, you can find all arrangements of a set of objects.

  - Design a class that will list all permutations of a string.

- A permutation is a rearrangement of the letters.
- The string `"eat"` has six permutations:

```
"eat"
"eta"
"aet"
"ate"
"tea"
"tae"
```

Permutations

- Problem: Generate all the permutations of "eat".
- First generate all permutations that start with the letter 'e', then 'a' then 't'.
- How do we generate the permutations that start with 'e'?
  - We need to know the permutations of the substring "at". But that's the same problem—to generate all permutations— with a simpler input
- Prepend the letter 'e' to all the permutations you found of 'at'.
- Do the same for 'a' and 't'.
- Provide a special case for the simplest strings.
  - The simplest string is the empty string, which has a single permutation—itself.

section_4/Permutations.java

```
1  import java.util.ArrayList;
2
3  /**
4     This program computes permutations of a string.
5  */
6  public class Permutations
7  {
8     public static void main(String[] args)
9     {
```

Program Run:

```
eat
eta
aet
ate
tea
tae
```

Self Check 13.13

What are all permutations of the four-letter word `beat`?

Answer: They are `b` followed by the six permutations of `eat`, `e` followed by the six permutations of `bat`, `a` followed by the six permutations of `bet`, and `t` followed by the six permutations of `bea`.

Self Check 13.14

Our recursion for the permutation generator stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?

Answer: Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.

Self Check 13.15

Why isn't it easy to develop an iterative solution for the permutation generator?

Answer: An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations `eat`, `eta`, and `aet`, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious —see Exercise P13.12.

Mutual Recursions

- **Problem:** to compute the value of arithmetic expressions such as:

```
3 + 4 * 5
(3 + 4) * 5
1 – (2 – (3 – (4 – 5)))
```

- Computing expression is complicated

  * and / bind more strongly than + and –

  Parentheses can be used to group subexpressions

Syntax Diagrams for Evaluating an Expression

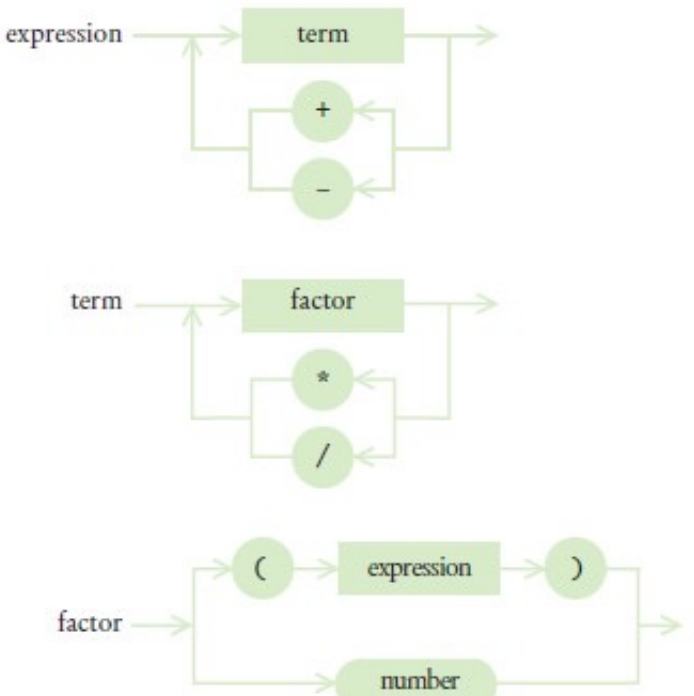


Figure 3

Mutual Recursions

- An expression can broken down into a sequence of terms, separated by + or – .
- Each term is broken down into a sequence of factors, separated by * or / .
- Each factor is either a parenthesized expression or a number.
- The syntax trees represent which operations should be carried out first.

Syntax Tree for Two Expressions

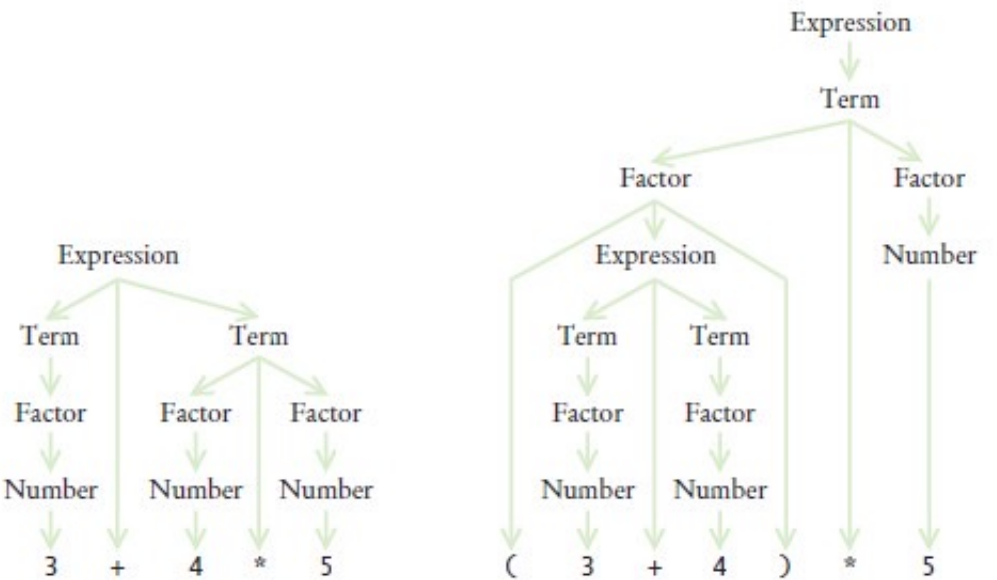


Figure 4 Syntax Trees for Two Expressions

Mutually Recursive Methods

- In a mutual recursion, a set of cooperating methods calls each other repeatedly.
- To compute the value of an expression, implement 3 methods that call each other recursively:

```
getExpressionValue
getTermValue
getFactorValue
```

The getExpressionValue Method

```
public int getExpressionValue()
{
   int value = getTermValue();
   boolean done = false;
   while (!done)
   {
      String next = tokenizer.peekToken();
      if ("+".equals(next) || "-".equals(next))
      {
         tokenizer.nextToken(); // Discard "+" or "-"
         int value2 = getTermValue();
         if ("+".equals(next)) value = value + value2;
         else value = value - value2;
      }
      else
      {
         done = true;
      }
   }
   return value;
}
```

The getTermValue Method

The `getTermValue` method calls `getFactorValue` in the same way, multiplying or dividing the factor values.

The getFactorValue Method

```
public int getFactorValue()
{
   int value;
   String next = tokenizer.peekToken();
   if ("(".equals(next))
   {
      tokenizer.nextToken(); // Discard "("
      value = getExpressionValue();
      tokenizer.nextToken(); // Discard ")"
   }
   else
   {
      value = Integer.parseInt(tokenizer.nextToken());
   }
   return value;
}
```

Using Mutual Recursions

To see the mutual recursion clearly, trace through the expression `(3+4)*5`:

- `getExpressionValue` calls `getTermValue`

  `getTermValue` calls `getFactorValue`

  - `getFactorValue` consumes the `(` input
  - `getFactorValue` calls `getExpressionValue`

    `getExpressionValue` returns eventually with the value of `7`, having consumed `3 + 4`. This is the recursive call.
  - `getFactorValue` consumes the `)` input
  - `getFactorValue` returns `7`

  `getTermValue` consumes the inputs `*` and `5` and returns `35`

- `getExpressionValue` returns `35`
- Recursion terminates when all the tokens of the input string are consumed.

section_5/Evaluator.java

```
1  /**
2     A class that can compute the value of an arithmetic expression.
3  */
4  public class Evaluator
5  {
6     private ExpressionTokenizer tokenizer;
7
8     /**
9        Constructs an evaluator.
```

section_5/ExpressionTokenizer.java

```
1  /**
2     This class breaks up a string describing an expression
3     into tokens: numbers, parentheses, and operators.
4  */
5  public class ExpressionTokenizer
6  {
7     private String input;
8     private int start; // The start of the current token
9     private int end; // The position after the end of the current token
```

section5/ExpressionCalculator.java

```
1  import java.util.Scanner;
2
3  /**
4     This program calculates the value of an expression
5     consisting of numbers, arithmetic operators, and parentheses.
6  */
7  public class ExpressionCalculator
8  {
9     public static void main(String[] args)
```

Program Run:

```
Enter an expression: 3+4*5
3+4*5=23
```

Self Check 13.16

What is the difference between a term and a factor? Why do we need both concepts?

Answer: Factors are combined by multiplicative operators (`*` and `/`), terms are combined by additive operators (+, –). We need both so that multiplication can bind more strongly than addition.

Self Check 13.17

Why does the expression parser use mutual recursion?

Answer: To handle parenthesized expressions, such as `2 + 3*(4 + 5)`. The subexpression `4 + 5` is handled by a recursive call to `getExpressionValue`.

Self Check 13.18

What happens if you try to parse the illegal expression `3+4*)5`? Specifically, which method throws an exception?

Answer: The `Integer.parseInt` call in `getFactorValue` throws an exception when it is given the string `")"`.

Backtracking

- Backtracking is a problem solving technique that builds up partial solutions that get increasingly closer to the goal.

  If a partial solution cannot be completed, one abandons it
  And returns to examining the other candidates.

- Characteristic properties needed to use backtracking for a problem.

  1. A procedure to examine a partial solution and determine whether to
     - Accept it as an actual solution.
     - Abandon it (either because it violates some rules or because it is clear that it can never lead to a valid solution).
     - Continue extending it.

2. A procedure to extend a partial solution, generating one or more solutions that come closer to the goal.

- In a backtracking algorithm, one explores all paths towards a solution. When one path is a dead end, one needs to backtrack and try another choice.

Backtracking

- Backtracking can then be expressed with the following recursive algorithm.

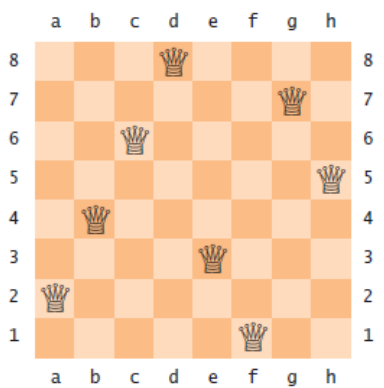
```
Solve(partialSolution)
  Examine(partialSolution).
  If accepted
    Add partialSolution to the list of solutions.
  Else if continuing
    For each p in extend(partialSolution)
      Solve(p).
```

Backtracking - Eight Queens Problem

- The Problem: position eight queens on a chess board so that none of them attacks another according to the rules of chess.

  There are no two queens on the same row, column, or diagonal

- A Solution to the Eight Queens Problem:



Backtracking - Eight Queens Problem

- To examine a partial solution:

  If two queens attack each other, reject it.

  Otherwise, if it has eight queens, accept it.

  Otherwise, continue.

- To extend a partial solution:

  Add another queen on an empty square

  For efficiency, place first queen in row 1, the next in row 2, and so on

Backtracking

- Provide a class `PartialSolution`

  that collects the queens in a partial solution,

  and that has methods to examine and extend the solution

```
public class PartialSolution
{
   private Queen[] queens;

   public int examine() { . . . }
   public PartialSolution[] extend() { . . . }
}
```

Backtracking

- The examine method simply checks whether two queens attack each other:

```
public int examine()
{
   for (int i = 0; i < queens.length; i++)
   {
      for (int j = i + 1; j < queens.length; j++)
      {
         if (queens[i].attacks(queens[j])) { return ABANDON; }
      }
   }
   if (queens.length == NQUEENS) { return ACCEPT; }
   else { return CONTINUE; }
}
```

Backtracking

- The extend method takes a given solution

  And makes eight copies of it.

  Each copy gets a new queen in a different column.

```
public PartialSolution[] extend()
{
   // Generate a new solution for each column
   PartialSolution[] result = new PartialSolution[NQUEENS];
   for (int i = 0; i < result.length; i++)
   {
      int size = queens.length;

      // The new solution has one more row than this one
      result[i] = new PartialSolution(size + 1);

      // Copy this solution into the new one
      for (int j = 0; j < size; j++)
      {
         result[i].queens[j] = queens[j];
      }

      // Append the new queen into the ith column
      result[i].queens[size] = new Queen(size, i);
   }
   return result;
}
```

Backtracking

- To determine if two queens attack each other diagonally

  Compute the slope.

  If it ±1 the queens attack each other diagonally?

- Just check:
  $|row_2 - row_1| = |column_2 - column_1|$

Backtracking

Backtracking in the Four Queens Problem

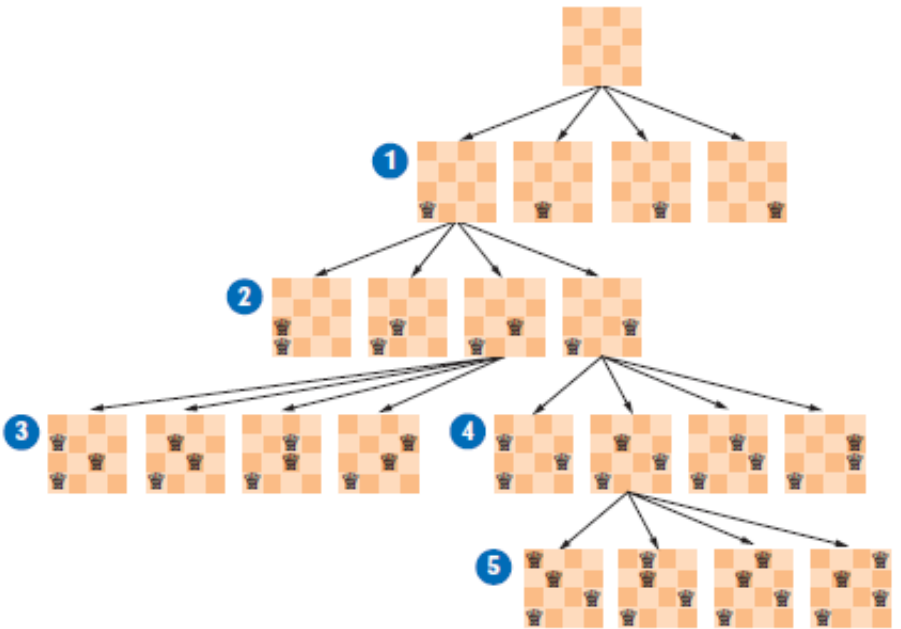


Figure 6 Backtracking in the Four Queens Problem

section_6/PartialSolution.java

```
1  import java.util.Arrays;
2
3  /**
4     A partial solution to the eight queens puzzle.
5  */
6  public class PartialSolution
7  {
8     private Queen[] queens;
9     private static final int NQUEENS = 8;
10
11    public static final int ACCEPT = 1;
12    public static final int ABANDON = 2;
13    public static final int CONTINUE = 3;
14
15    /**
16       Constructs a partial solution of a given size.
17       @param size the size
18    */
19    public PartialSolution(int size)
20    {
21       queens = new Queen[size];
22    }
23
24    /**
25       Examines a partial solution.
26       @return one of ACCEPT, ABANDON, CONTINUE
27    */
```

```
28      public int examine()
29      {
30         for (int i = 0; i < queens.length; i++)
31         {
32            for (int j = i + 1; j < queens.length; j++)
33            {
34               if (queens[i].attacks(queens[j])) { return ABANDON; }
35            }
```

section_6/Queen.java

```
1  /**
2     A queen in the eight queens problem.
3  */
4  public class Queen
5  {
6     private int row;
7     private int column;
8
9     /**
10       Constructs a queen at a given position.
11       @param r the row
12       @param c the column
13    */
14    public Queen(int r, int c)
15    {
16       row = r;
17       column = c;
18    }
19
20    /**
21       Checks whether this queen attacks another.
22       @param other the other queen
23       @return true if this and the other queen are in the same
24       row, column, or diagonal.
25    */
26    public boolean attacks(Queen other)
```

```
27      {
28          return row == other.row
29              || column == other.column
30              || Math.abs(row - other.row) == Math.abs(column - other.col
31      }
32
33      public String toString()
34      {
```

section_6/EightQueens.java

```
1  /**
2     This class solves the eight queens problem using backtracking.
3  */
4  public class EightQueens
5  {
6     public static void main(String[] args)
7     {
8        solve(new PartialSolution(0));
9     }
10
11    /**
12       Prints all solutions to the problem that can be extended from
13       a given partial solution.
14       @param sol the partial solution
15    */
16    public static void solve(PartialSolution sol)
17    {
18       int exam = sol.examine();
19       if (exam == PartialSolution.ACCEPT)
20       {
21          System.out.println(sol);
22       }
23       else if (exam == PartialSolution.CONTINUE)
24       {
25          for (PartialSolution p : sol.extend())
26          {
27             solve(p);
```

```
28              }
29          }
30      }
31  }
32
```

Program Run:

```
[a1, e2, h3, f4, c5, g6, b7, d8]
[a1, f2, h3, c4, g5, d6, b7, e8]
[a1, g2, d3, f4, h5, b6, e7, c8]
. . .
[f1, a2, e3, b4, h5, c6, g7, d8]
. . .
[h1, c2, a3, f4, b5, e6, g7, d8]
[h1, d2, a3, c4, f5, b6, g7, e8]
```

(92 solutions)

Self Check 13.19

Why does `j` begin at `i + 1` in the `examine` method?

Answer: We want to check whether any `queen[i]` attacks any `queen[j]`, but attacking is symmetric. That is, we can choose to compare only those for which `i < j` (or, alternatively, those for which `i > j`). We don't want to call the attacks method when `i` equals `j`; it would return true.

Self Check 13.20

Continue tracing the four queens problem as shown in Figure 6. How many solutions are there with the first queen in position `a2`?

Answer: One solution:

Self Check 13.21

How many solutions are there altogether for the four queens problem?

Answer: Two solutions: The one from Self Check 20, and its mirror image.