Recursion

CHAPTER 13

Recursive Solution

- **A recursive solution**
  - solves a problem by solving a *smaller* instance of the problem.
- Example
  - How do we go about looking for a word in a dictionary?
  - Two methods
    - Sequential search
    - Binary search

Recursive Solutions

- Sequential search
  - Starts at the beginning of the collection
  - examine every element until a match is found
  - could be time consuming
  - what if the list contained millions of elements?
- Binary search
  - Repeatedly halves the collection and determines which half could contain the item
  - Uses a divide and conquer strategy

Dictionary

- **Divide and conquer** strategy
- First dividing the dictionary into two halves
- Chose the correct half
- Divide again into half (smaller problem)
- *Continue* until *a base case* is reached.
  - we have reached a single page containing the word.

```
//Search a dictionary for a word using a recursive binary search
if (the dictionary contains only one page)
{
    Scan the page for the word
}
else
{
    Open the dictionary to a point near the middle.
    Determine which half of the dictionary contains the word.
    if (the word is in the first half of the dictionary)
    {
            Search the first half of the dictionary for the word.
    }
    else
    {
            Search the second half of the dictionary for the word.
    } //end if
} //end if
```

Recursive Solutions

- Facts about a recursive solution
  - A recursive method calls itself
  - Each recursive call solves an identical, but smaller, problem
  - A test for the base case enables the recursive calls to stop
    - Base case: a known case in a recursive definition
  - Eventually, one of the smaller problems must be the base case

4 Question for Recursive Solutions

1. How can you define the problem in terms of a smaller identical problem?
2. Do you diminish the size of the problem?
3. What instance of the problem can serve as the base case?
4. As the problem size diminishes, will you reach this base case?

Recursive Solutions

- Theory - can solve a large variety of problems.
- In practice
  - *computationally expensive*
  - can only be used to solve small instances of such problems in a reasonable amount of time.

Triangle Problem

- Want to computer the area of the triangle
- Has a width n
- Each [] has area 1

- []
- [][]
- [][][]

- The third triangle number is 6.

Triangle Example Outline

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

        public int getArea()
        {
                .......
        }
private int width;
}
```

Complete getArea()

Take care of the case where the width is 1 or []

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

Now that we know that we computer the area of the larger triangle as

        smallerArea + width;

Complete getArea()

How do we get the smaller area?
Make a smaller area and calculate.

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

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

Process

The 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 the getArea on that triangle.
That method returns 1.
That method returns smallerArea + width = 1 + 2 = 3.
That method returns smallerArea + width = 3 + 3 = 6.
That method returns smallerArea + width = 6 + 4 + 10.

A Recursive Method: The Factorial of n

- Problem
  - Compute the factorial of an integer n
- An iterative definition of factorial(n)

  factorial(n) = n * (n-1) * (n-2) * … * 1 for any integer n > 0

  factorial(0) = 1

A Recursive Method: The Factorial of n

- A recursive definition of factorial(n)

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

- A recurrence relation
  - A mathematical formula that generates the terms in a sequence from previous terms
  - Example

$$\text{factorial}(n) = n * [(n-1) * (n-2) * \ldots * 1]$$
$$= n * \text{factorial}(n-1)$$

Factorial

- factorial(4) = 4*factorial(3)
- factorial(3) = 3*factorial(2)
- factorial(2) = 2*factorial(1)
- factorial(1) = 1*factorial(0)
- factorial(0) = 1

- now we can solve it

Factorial

- factorial(0) = 1
- factorial(1) = 1*factorial(0) = 1*1 = 1
- factorial(2) = 2*factorial(1) = 2*1 = 2
- factorial(3) = 3*factorial(2) = 3*2 = 6
- factorial(4) = 4*factorial(3) = 4*6 = 24

Code


```
public class Factorial
{
        public static void main(String [] arg)
         {
                fact(4);
        }
        public static int fact(int n)
        {
                if (n==0)
                        return 1;
                else
                {
                        int num = (n*fact(n-1));
                        System.out.println(num);
                        return num;
                }
        }
}
```

Permutations

- More complex example
- Write a class that lists all permutations of a string.
- How could we use such a class?
  - Password cracker
  - We have a given set of letters (all the alphabet)
  - We simple write a program that will allow us to get all permutations of that alphabet.
  - The larger the original set – the more permutations – the longer to crack

Permutations of “eat”

- Begin by removing the first letter. (e in this case)
- Generate all the combinations of the remaining letters
  - at and ta
- Therefore all the possible combinations of beginning with e are eat and eta
- Now get second letter (a in this case)
- Generate all the combinations of the remaining letters
  - et and te
- Repeat for each remaining letter

Permutation GeneratorDemo

```
import java.util.ArrayList;

public class PermutationGeneratorDemo

{
        public static void main(String[] agrs)
        {
                PermutationGenerator generator = new
                        PermutationGenerator("eat");
                ArrayList<String> permutations = generator.getPermutations();
                for (String s: permutations)
                {
                        System.out.println(s);
                }
        }
}
```

PermutationGenerator

```
public class PermutationGenerator
{
        public PermutationGenerator(String aWord)
        {
                word = AWord;
        }

        public ArrayList<String> getPermutations()
        {
                ArrayList<String> result = new ArrayList<String>();

                // The empty string has a single permutations: iself

                if (word.length() == 0)
                {
                        result.add(word);
                        return result;
                }
```

PermutationGenerator

```
// Loop through all character postions

                for (int i=0; i<word.length(); i++)
                {
                        // Form a simpler work by removing the ith character
                        String shorterWord = work.substring(0,i)
                                + word.substring(i+1);

                        //Generate all permutations of the simpler word
                        PermutationGenerator shorterPermutationGenerator
                                = new PermutationGenerator(shorterWord);
                        ArrayList<String> shorterWordPermutations  =
                                shorterPermutationGenerator.getPermutations();
```

PermutationGenerator

```
// Add the removed charater to the front of
                        // each permutation of the simpler word

                        for String s: shorterWordPermutations)
                        {
                                result.add(word.charAt(i) + s);
                        }

                        return result;
                }

                private String word;

}
```

Palindrome

- Test whether a string is equal to itself when you reverse all the characters.
  - Racecar
- Step 1: Consider ways to simplify inputs.
  - Remove first character
  - Remove last character
  - Remove a character from the middle
  - Cut the string into two halves.

Palindrome

- Step 2: See if the solutions you found in step 1, alone or in combination, work in solving your problem.
  - For palindrome try this
  - If the first and last characters are both letter, then check to see if they match. If so remove both and test again.
  - If the last character isn't a letter, remove it and test the shorter string.
  - If the first character isn't a letter, remove it and test the shorter string.

Palindrome

- Step 3: Find solutions to the simplest inputs.
  - Simplest strings
    - String with two characters
    - Strings with single character
    - The empty string
- Step 4: Implement the solution by combining the simple cases and the reduction step.

Code for IsPalindrome

```
pubic boolean isPalindrome()
{
      int length = text.length();

       if (length <= 1) return true;

        char first = Character.toLowerCase(text.charAt(0));
         char last = Character.toLowerCase(text.charAt(length -1));
```

Code for IsPalindrome

```
// works only if both are letters)
if ((Character.isLetter(first) && Character.isLetter(last))
{
        if (first==last)
        {

        // remove both first and last character.
        Sentence shorter = new Sentence(text.substring 1, length -1));
        return shorter.isPalindrome();
        }

        else return false;
}
```

Code for IsPalindrome

```
else if(! character.isLetter(last))
{
        Sentence shorter = new Sentence(text.substring(0, length -1));
        return shoter.isPalindrome();
}
else
{
        Sentence shorter = new Sentence(text.substring(1));
        return shorter.isPalindrome();
}
}
```

Helper Methods

- Assist by changing the original problem slightly
- Palindrome – create new sentence in every step
- Rather than testing whether the entire sentence is a palindrome, check whether a substring is a palindrome.

```
pubic boolean isPalindrome(int start, int end)
{
        if (start >= end) return true;

        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            if (first == last)
            {
                return isPalindrome(start +1, end -1)
            }
            else return false;
        }
        else if (!Charater.isLetter(last))
        {
            return isPalindrome(start,end-1);
        }
        else
        {
            return isPalindrome(start+1, end)
        }
}
```

Method to Solve the Whole Problem

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

- Public sees this one.
- Work is performed by other one

Efficiency

- Occasionally runs much slower
- Most cases only slightly slower
- Easier to understand
- Easier to implement

Fibonacci Sequence

- Series of equations
  - Definitions
  - $f_1 = 1$
  - $f_2 = 1$
  - $f_n = f_{n-1} + f_{n-2}$

Code

```
public class Recursive Fib
{
        public static void main(String[]  args)
        {
                Scanner in = new Scanner(System.in);
                System.out.println("Enter n: ");
                int n = in.nextInt();
                for (int i = 1; k <=n; i++)
                {
                        long f = fib(i);
                        System.out.println("fib( " + i + ") = " + f);
                }
        }
                public static long fib(int n)
        {
                if (n <=2) return 1:
                else return fib(n - 1) + fib(n - 2);
        }
}
```