Chapter 11

RECURSION

Chapter Goals

- To learn to "think recursively"
- To be able to use recursive helper functions
- 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

Contents

11.1 Triangle Numbers Revisited

- Triangle shape of side length 4:

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

- Will use recursion to compute the area of a triangle of width *n* , assuming each `[]` square has an area of 1
- Also called the *n*th *triangle number*
- The third triangle number is 6, the fourth is 10

Handling Triangle of Width 1

- The triangle consists of a single square
- Its area is 1
- Take care of this case first:

```
def triangleArea(sideLength) :
    if sideLength == 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

```
area = smallerArea + sideLength
```

- To get the area of the smaller triangle
  - Call the `triangleArea()` function*:*

```
smallerSideLength = sideLength - 1
smallerArea = triangleArea(smallerSideLength)
```

Computing the Area of a Triangle With Width 4

- `triangleArea()` function makes a smaller triangle of width 3
- It calls `triangleArea()` on that triangle
  - That function makes a smaller triangle of width 2
  - It calls `triangleArea()` on that triangle
    - That function makes a smaller triangle of width 1
    - It calls `triangleArea()` on that triangle
      - That function returns 1
    - The function returns `smallerArea + sideLength` = 1 + 2 = 3
  - The function returns `smallerArea + sideLength` = 3 + 3 = 6
- The function returns `smallerArea + sideLength` = 6 + 4 = 10

Recursive Computation

- A **recursive computation** solves a problem by using the solution to the same problem with simpler inputs
- Call pattern of a **recursive function** is complicated
    - Key: *Don't think about it*

Successful Recursion

- Every recursive call must simplify the computation in some way
- There must be special cases to handle the simplest computations directly

Other Ways to Compute Triangle Numbers

- The area of a triangle equals the sum:

```
1 + 2 + 3 + ... + sideLength
```

- Using a simple loop:

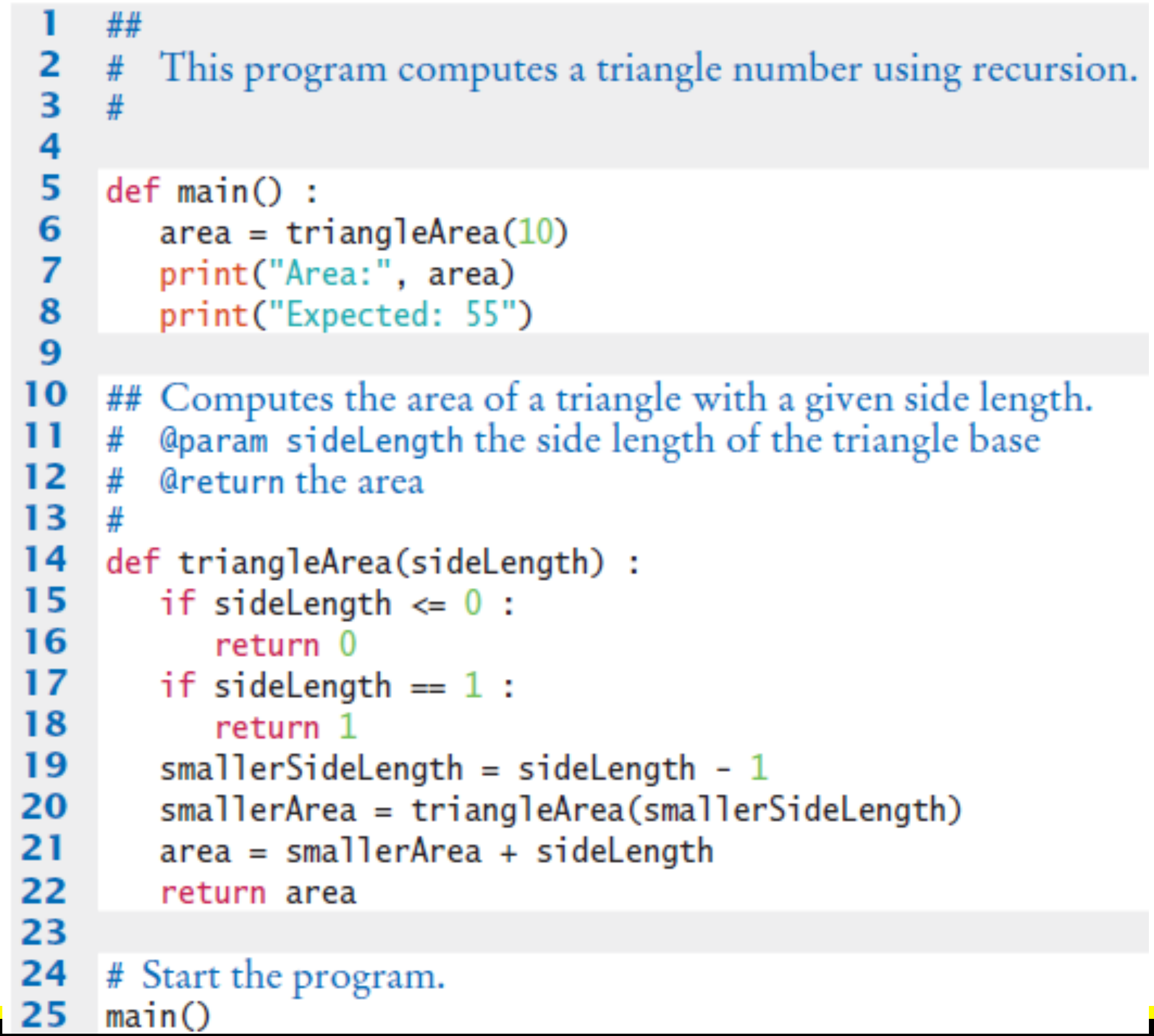
```
area = 0;
for i in range (1, (sideLength+1), 1) :
   area = area + i
```

- Using math:

$1 + 2 + \ldots + n = n \times (n + 1)/2$

=> `n * (n + 1) / 2`

Trianglenumbers.py



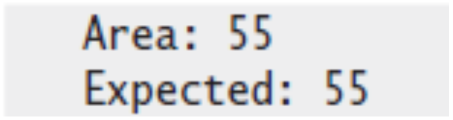
```
##
#   This program computes a triangle number using recursion.
#

def main() :
   area = triangleArea(10)
   print("Area:", area)
   print("Expected: 55")

## Computes the area of a triangle with a given side length.
#  @param sideLength the side length of the triangle base
#  @return the area
#
def triangleArea(sideLength) :
   if sideLength <= 0 :
      return 0
   if sideLength == 1 :
      return 1
   smallerSideLength = sideLength - 1
   smallerArea = triangleArea(smallerSideLength)
   area = smallerArea + sideLength
   return area

# Start the program.
main()
```

Program Run



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

Special Topic 11.1

- An Object-Oriented version of the triangles program

```
class Triangle
    def _ _init_ _ (self, sideLength) :
        self._sideLength = sideLength
    def getArea(self) :
        if self._sideLength == 1 :
            return 1
        . . .
```

- General case: compute the area of the larger triangle as `smallerArea + self._sideLength`.
- To get the smaller area:

```
smallerTriangle = Triangle(self._sideLength - 1)
smallerArea = smallerTriangle.getArea()
area = smallerArea + self._sideLength
```

Common Error 11.1

- Infinite recursion:
  - A function calling itself over and over with no end in sight.
  - The computer needs some amount of memory for bookkeeping during each call.
  - After some number of calls, all memory that is available for this purpose is exhausted.
  - Your program shuts down and reports a “stack overflow”.
- Causes:
  - The arguments don’t get simpler or because a special terminating case is missing.

Thinking Recursively

- 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()` Function

```
## Tests whether a string is a palindrome.
# @param text a string that is being checked
# @return True if text is a palindrome, False otherwise
#
def isPalindrome(text) :
    . . .
```

Thinking Recursively: Step 1

- Consider various ways to simplify inputs.
- 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: Step 2 (1)

- Combine solutions with simpler inputs into a solution of the original problem.
- Most promising simplification: *Remove both 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*

Thinking Recursively: Step 2 (2)

- 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: Step 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: Step 4 (1)

- Implement the solution by combining the simple cases and the reduction step.

```
def isPalindrome(text) :
    length = len(text)
    # Separate case for shortest strings.
    if length <= 1 :
        return True
    else :
        # Get first and last characters, converted to
        # lowercase.
        first = text[0].lower()
        last = text[length - 1].lower()
```

Continued

Thinking Recursively: Step 4 (2)

```
# Non base case
if first.isalpha() and last.isalpha() :
    # Both are letters.
    if first == last :
        # Remove both first and last character.
        shorter = text[1 : length - 1]
        return isPalindrome(shorter)
    else :
        return False
elif not last.isalpha() :
    # Remove last character.
    shorter = text[0 : length - 1]
    return isPalindrome(shorter)
else :
    # Remove first character.
    shorter = text[1 : length]
    return isPalindrome(shorter)
```

Recursive Helper functions

- 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 section.
- It is a bit inefficient to construct new string objects in every step.

Substring Palindromes (1)

- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:

```
## Recursively tests whether a substring is
# a palindrome.
# @param text a string that is being checked
# @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
#
def substringIsPalindrome(text, start, end) :
```

Substring Palindromes (2)

- Then, simply call the helper function with positions that test the entire string:

```
def isPalindrome(text) :
    return substringIsPalindrome(text, 0, len(text) – 1)
```

Recursive Helper function

```
def substringIsPalindrome(text, start, end) :
    # Separate case for substrings of length 0 and 1.
    if start >= end :
        return True
    else :
    # Get first and last characters, converted to lowercase.
        first = text[start].lower()
        last = text[end].lower()
```

Continued

Recursive Helper Function

```
if first.isalpha() and last.isalpha() :
    if first == last :
        # Test substring that doesn’t contain the matching
        # letters.
        return substringIsPalindrome
           (text, start + 1, end - 1)
    else :
        return False
elif not last.isalpha() :
    # Test substring that doesn’t contain the last character.
    return substringIsPalindrome(text, start, end - 1)
else :
    # Test substring that doesn’t contain the first
    # character.
    return substringIsPalindrome(text, start + 1, end)
```