12 Recursion

Figure 12.1 credit: modification of work "Rings from the Ocean", by Sue Corbisez/Flickr, CC BY 2.0

Chapter Outline

Introduction

Recursion is a powerful programming technique to write solutions compactly. Recursion simplifies programs by calling smaller versions of a given problem that are used to generate the solution to the overall problem.

In advanced programming, some programs are difficult to write without the use of recursion. So recursion is also a style of programming that enables us to simplify programs.

Recursion enables a piece of code, a function, to call the same piece of code, the same function, with different parameters.

12.1 Recursion basics

Learning objectives

By the end of this section you should be able to

- Describe the concept of recursion.
- Demonstrate how recursion uses simple solutions to build a better solution.

Recursion

Recursion is a problem solving technique that uses the solution to a simpler version of the problem to solve the bigger problem. In turn, the same technique can be applied to the simpler version.

CHECKPOINT

Three Towers

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-1-recursion-basics)

CONCEPTS IN PRACTICE

Three Towers and recursion

1. How is the problem of moving two rings solved using recursion?
   a. Move the small ring to the middle tower, move the bigger ring to the target tower, and move the small ring to the target tower.
   b. Move two rings from the source tower to the target tower.
   c. Cannot be solved with recursion.

2. How is the problem of moving three rings solved using recursion?
   a. Move three rings from the source tower to the target tower.
   b. Move two rings to the middle tower, then move the biggest ring to the target tower, and finally, move two rings to the target tower.
   c. Cannot be solved with recursion.

3. How many times is the two-ring solution used with three rings?
   a. 1
   b. 2
   c. 0

Recursion to find a complete solution

The recursion process continues until the problem is small enough, at which point the solution is known or can easily be found. The larger solution can then be built systematically by successively building ever larger solutions until the complete problem is solved.

CHECKPOINT

Solving Three Towers

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-1-recursion-basics)

CONCEPTS IN PRACTICE

Solving Three Towers

4. How many total steps does it take to solve two rings?
   a. 1
   b. 2

c. 3

5. How many total steps does it take to solve three rings?
   a. 3
   b. 4
   c. 7

6. How many total steps does it take to solve four rings?
   a. 15
   b. 7
   c. 3

12.2 Simple math recursion

Learning objectives

By the end of this section you should be able to

- Identify a recursive case and a base case in a recursive algorithm.
- Demonstrate how to compute a recursive solution for the factorial function.

Calculating a factorial

The factorial of a positive integer is defined as the product of the integer and the positive integers less than the integer.

Ex: `5! = 5 * 4 * 3 * 2 * 1`

Written as a general equation for a positive integer `n`: `n! = n * (n - 1) * (n - 2) * . . . * 1`

The above formula for the factorial of n results in a recursive formula: `n! = n * (n - 1)!`

Thus, the factorial of n depends upon the value of the factorial at `n - 1`. The factorial of n can be found by repeating the factorial of `n - 1` until `(n - 1)! = 1!` (we know that `1! = 1`). This result can be used to build the overall solution as seen in the animation below.

CHECKPOINT

Finding the factorial of 5

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-2-simple-math-recursion)

CONCEPTS IN PRACTICE

Recognizing recursion

Can the following algorithms be written recursively?

1. The summation of `1 + 2 + 3 + . . . + (n - 1) + n` where n is a positive integer.

2. Listing the odd numbers greater than 0 and less than a given number n.

3. Listing the primary numbers (prime numbers) greater than 3.
4. Listing the Fibonacci sequence of numbers.

Defining a recursive function

Recursive algorithms are written in Python as functions. In a recursive function different actions are performed according to the input parameter value. A critical part of a recursive function is that the function must call itself.

A value for which the recursion applies is called the **recursive case**. In the recursive case, the function calls itself with a smaller portion of the input parameter. Ex: In the recursive function `factorial()`, the initial parameter is an integer n. In the function's recursive case, the argument passed to `factorial()` is `n - 1`, which is smaller than n.

A value of n for which the solution is known is called the **base case**. The base case stops the recursion. A recursive algorithm must include a base case; otherwise, the algorithm may result in an infinite computation.

To calculate a factorial, a recursive function, `factorial()` is defined with an integer input parameter, n. When `n > 1`, the recursive case applies. The `factorial()` calls itself with a smaller argument, `n - 1`. When `n == 1`, the solution is known because `1!` is `1`; therefore, `n == 1` is a base case.

Note: `0!` is defined to be `1`; therefore, `n == 0` is a second base case for `factorial()`. When `n < 1`, an error is returned.

CHECKPOINT

Factorial using recursion

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-2-simple-math-recursion)

CONCEPTS IN PRACTICE

Programming recursion

For the questions below, the function `rec_fact()` is another recursive function that calculates a factorial. What is the result of each definition of `rec_fact()` if `n = 17` is the initial input parameter?

5.
```
def rec_fact(n):
  return n * rec_fact(n - 1)
```
a. 355687428096000
b. 0
c. no result / infinite computation

6.
```
def rec_fact(n):
  if n < 0:
    print("error")
  elif n == 0:
    return n
  else:
    return n * rec_fact(n - 1)
```

a. `355687428096000`
b. `0`
c. no result / infinite computation

7.
```
def rec_fact(n):
  if n < 0:
    print("error")
  elif n == 0:
    return 1
  else:
    return n * rec_fact(n - 1)
```

a. `355687428096000`
b. `0`
c. no result / infinite computation

8.
```
def rec_fact(n):
  if n < 0:
    return -1
  else:
    return n * rec_fact(n - 1)
```

a. `355687428096000`
b. `0`
c. no result / infinite computation

TRY IT

Recursive summation

Write a program that uses a recursive function to calculate the summation of numbers from 0 to a user specified positive integer n.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-2-simple-math-recursion)

TRY IT

Digits

Write a program that computes the sum of the digits of a positive integer using recursion.

Ex: The sum of the digits of 6721 is 16.

Hint: There are 10 base cases, which can be checked easily with the right condition.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-2-simple-math-recursion)

12.3 Recursion with strings and lists

Learning objectives

By the end of this section you should be able to

- Demonstrate the use of recursion to solve a string problem.
- Demonstrate the use of recursion to solve a list problem.
- Use the built-in `count()` list function.

Recursion with strings

A word that is spelled the same forward and backward is called a palindrome. Ex: racecar.

Recursion can be used to identify whether a given word is a palindrome.

CHECKPOINT

Identifying a palindrome

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-3-recursion-with-strings-and-lists)

CONCEPTS IN PRACTICE

Recursion with strings

Refer to the animation above.

1. Which of the following does `palindrome()` recognize as a palindrome?
   a. `madamm`
   b. `madaM`
   c. `madame`

2. What would happen if the condition on line 4 of `palindrome()` is changed to `if len(word) == 1:`
   a. Only palindromes with an odd number of letters would be recognized correctly.
   b. Nothing. The function would work the same.
   c. The function would not recognize any palindromes.

3. What would happen if the condition in line 8 `palindrome()` is changed to `if`

palindrome(word.strip(word[0])).

a. Nothing. The function would work the same: the first and last letter would be removed.
b. Some words, such as "madamm", would be incorrectly recognized as a palindrome.

Recursion with lists

The animation below shows a recursive way to check whether two lists contain the same items but in different order.

The `count()` function returns a count of the number of items in a list that match the given item, and returns `0` otherwise. Ex: For `list_num = [1, 3, 3, 4]`, `list_num.count(3)` returns `2`.

CHECKPOINT

Checking list permutations

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-3-recursion-with-strings-and-lists)

CONCEPTS IN PRACTICE

List permutations

Refer to the above animation. What would `permu_check()` return for each pair of lists below?

4.
```
list_num = [1, 7, 99, 2]
other_list = [99, 7, 1, 2]
```
   a. `True`
   b. `False`

5.
```
list_num = [22, 9, 15, 17, 15]
other_list = [17, 9, 22, 22, 15]
```
   a. `True`
   b. `False`

6.
```
honors_list = ["Bo", "Joe", "Sandy"]
first_names_of_team_1 = ["Sandy", "Bo", "Joe"]
```
   a. `True`
   b. `False`

7.
```
this_list = [1, 2, [3], [4], 5]
that_list = [[3], 1, 5, [4], 2]
```

a. `True`
b. `False`

8.
```
list_1 = [1, 2, 3]
list_2 = [1, 2, 3]
```

a. `True`
b. `False`

TRY IT

Remove duplicates

Write a recursive `rem_dup()` function that removes duplicates from a list.

Ex: List `[5, 5, 2, 1, 3, 1, 6]` should result in an output list `[5, 2, 1, 3, 6]`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-3-recursion-with-strings-and-lists)

12.4 More math recursion

Learning objectives

By the end of this section you should be able to

- Define a recursive function to generate Fibonacci numbers.

Fibonacci sequence using recursion

The Fibonacci sequence is a sequence in which each number in the sequence is the sum of the previous two numbers in the sequence. The first two numbers are 0 and 1. Thus, starting from 0 and 1, the third number is 0 + 1 = 1, and the next number is 1 + 1 = 2. The sequence of numbers is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Recursion can be used to calculate the nth Fibonacci number. The base cases are 0 and 1. Thereafter, the nth number is given by adding the (n - 1)th and (n - 2)th number. Thus, `fib(n) = fib(n - 1) + fib(n - 2)`, which is a recursive definition ending with base cases `f(0) = 0`, and `f(1) = 1`.

For a recursion that calls multiple functions in the recursive case, tracing how the recursion proceeds is useful. A structure used to trace the calls made by a recursive function is called a **recursion tree**. The animation below traces the recursion tree for a call to a recursive function to calculate the Fibonacci number for `n = 5`.

EXAMPLE 12.1

Finding the nth Fibonacci number

The Fibonacci function recursion is more complex with the value at n depending on two function calls with smaller values.

```
""" Recursive Fibonacci function """

def fib(n):
  # Base case
  if n == 0 or n == 1:
    return n
  # Recursive case
  elif n > 1:
    return fib(n - 1) + fib(n - 2)
  # Error case
  else:
    print("Fibonacci numbers begin at 0.")
    return

# Test code
print(fib(7))
```

The above code's output is:

```
13
```

CHECKPOINT

Recursive Fibonacci function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-4-more-math-recursion)

CONCEPTS IN PRACTICE

Fibonacci numbers

1. `fib(9)` results in _____.
   a. 13
   b. 21
   c. 34

2. How many calls to the base cases are made as a result of calling `fib(5)`?
   a. 2
   b. 5
   c. 8

3. A structure tracing the function calls made as a result of a complex recursive function call is called a _____.

a. tree
b. recursion tree
c. iteration tree

Greatest common divisor (GCD)

The greatest common divisor (GCD) of two positive integers is an integer that is a divisor for both integers. Ex: The GCD of 6 and 9 is 3 because 3 x 2 = 6, and 3 x 3 = 9.

The GCD is found easily using Euclid's method. Euclid's method recursively subtracts the smaller integer from the larger integer until a base case with equal integers is reached. The greatest common divisor is the integer value when the base case is reached.

EXAMPLE 12.2

Finding the GCD

```
""" Find GCD using recursive implementation of Euclid's method """

def gcd(a, b):
  if a == b: # Base case
    return a
  elif a < b: # Recursive case
    return gcd(a, b - a)
  else:
    return gcd(a - b, a)

# Test code
print(gcd(24, 30))
```

The above code's output is:

```
6
```

CONCEPTS IN PRACTICE

GCD

4. What is the GCD of 15 and 35?
a. 15
b. 20
c. 5

5. How many recursive calls are made with `gcd(24, 30)`?
   a. 4
   b. 1
   c. 24

6. What is the GCD of 13 and 23?
   a. 13
   b. 1
   c. 23

7. How many recursive calls are made with `gcd(13, 23)`?
   a. 1
   b. 8
   c. 13

TRY IT

Recursive power

Write a recursive `power()` function, such that given an integer x and a positive integer y, `power(x, y)` returns x raised to y.

Ex: `power(3, 4)` returns `81`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-4-more-math-recursion)

TRY IT

Recursive power (with any integer power)

Write a `recursive rec_pow()` function such that given two integers, x and y, it returns x raised to y.

(Note: this is an extension of the above problem but now works for any integer value of y, positive or negative. How should the recursive function change to deal with a negative value of y?)

Ex: `rec_pow(2, -4)` returns `0.0625`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-4-more-math-recursion)

12.5 Using recursion to solve problems

Learning objectives

By the end of this section you should be able to

- Use recursion to efficiently search a list.
- Demonstrate a solution to the Three Towers problem.

Binary search

Searching a sorted list usually involves looking at each item. If the item being searched is not found, then the search can take a long time.

A binary search is a recursive algorithm used to efficiently search sorted lists. In each recursive step, about half the items are discarded as not being potential matches, so the search proceeds much faster.

A binary search begins by checking the middle element of the list. If the search key is found, the algorithm returns the matching location (base case). Otherwise, the search is repeated on approximately half the list. If the key is greater than the middle element, then the key must be on the right half, and vice versa. The process continues by checking the middle element of the remaining half of the list.

EXAMPLE 12.3

Binary search

```
""" Binary Search """

def binary_search(search_list, low, high, key):

  # Check base case
  if high > low:
    mid = (high + low) // 2

    # If element is present at the middle itself (base case)
    if search_list[mid] == key:
      return mid

    # Recursive case: check which subarray must be checked
    # Right subarray
    elif key > search_list[mid]:
      return binary_search(search_list, mid + 1, high, key)

    # Left subarray
    else:
      return binary_search(search_list, low, mid - 1, key)

  else:
    # Key not found (other base case)
    return "Not found"

# Test list
in_list = [1, 3, 13, 16, 19, 22, 27, 32, 48, 66, 78, 99, 111, 122]

# Call binary search function
print(binary_search(in_list, 0, len(in_list)-1, 48)) # Key exists at index 8

print(binary_search(in_list, 0, len(in_list)-1, 86)) # Key does not exist
```

The above code's output is:

```
8
Not found
```

CHECKPOINT

Binary search

[Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-5-using-recursion-to-solve-problems)](https://openstax.org/books/introduction-python-programming/pages/12-5-using-recursion-to-solve-problems)

CONCEPTS IN PRACTICE

Binary search

1. Which list can be searched with the binary search algorithm?
   a. `[5, 2, 7, 1, 8]`
   b. `[9, 8, 7, 6, 5]`
   c. `[4, 5, 6, 7, 8]`

2. If the `binary_search()` function is called with `low = 4`, and `high = 7`, `mid` computes to _____.
   a. `5`
   b. `6`
   c. `11`

3. How many calls to the `binary_search()` function occur in the animation?
   a. 3
   b. 4
   c. 8

4. How many calls to the `binary_search()` function occur in the code example when searching `86`?
   a. 4
   b. 14

Solving Three Towers

As discussed in an earlier section, the Three Towers problem can be solved using recursion. The solution depends on calling the solution to the next smaller problem twice. As shown in the code example below, the recursive solution can solve the problem for any number of rings.

EXAMPLE 12.4

Solving N towers

The solution to Three Towers is simple with recursion. In the code below, rings are numbered from the top down. The smallest ring is 1, the next ring is 2, and when solving for three rings, the bottom ring is 3.

```
    """ Solving the towers problem recursively """

    def three_towers(N, source_tower, dest_tower, temp_tower):
        # Base case: simply move the single(bottom) ring from source to destination
tower
        if N==1:
            print("Move ring 1 from tower", source_tower, "to tower", dest_tower)
            return # Exit when the base case is reached

        # Recursive case
        # Call the smaller version of the problem:
        # to move the N-1 stack to the middle tower
        three_towers(N-1, source_tower, temp_tower, dest_tower)

        # Move the N ring to the destination tower
        print("Move ring", N, "from tower", source_tower, "to tower", dest_tower)

        # Call the smaller version of the problem:
        # to now move the N-1 stack from the middle tower
        # to the destination
        three_towers(N-1, temp_tower, dest_tower, source_tower)

    # Test code
    print("Solution for 3 rings:")
    three_towers(3, 't1', 't3', 't2') # t1, t2, t3 are the towers
```

The above code's output is:

```
    Solution for 3 rings:
    Move ring 1 from tower t1 to tower t3
    Move ring 2 from tower t1 to tower t2
    Move ring 1 from tower t3 to tower t2
    Move ring 3 from tower t1 to tower t3
    Move ring 1 from tower t2 to tower t1
    Move ring 2 from tower t2 to tower t3
    Move ring 1 from tower t1 to tower t3
```

CONCEPTS IN PRACTICE

Solving Three Towers

5. For four rings, how many total lines will be printed by `three_towers()`?
   a. 7
   b. 15

6. Would an iterative solution to the Three Towers problem be more complex or less complex?
   a. more complex
   b. less complex

TRY IT

Coin combinations

Write a recursive function `print_H_T()` that produces all possible combinations of heads (`"H"`) and tails (`"T"`) for a given number of coin tosses.

Ex: For three coins, the program should print the output shown below.

```
HHH
HHT
HTH
HTT
THH
THT
TTH
TTT
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/12-5-using-recursion-to-solve-problems)

12.6 Chapter summary

Highlights from this chapter include:

- Describe the concept of recursion.
- Demonstrate how recursion uses simpler solutions to build a bigger solution.
- Identify a recursive case and a base case in a recursive algorithm.
- Demonstrate how to compute a recursive solution for the factorial function.
- Demonstrate the use of recursion to solve a string problem.
- Demonstrate the use of recursion to solve a list problem.
- Use the built-in `count()` list function.
- Define a recursive function to generate Fibonacci numbers.
- Use recursion to efficiently search a list.
- Demonstrate a solution to the Three Towers problem.

At this point, you should be able to solve problems using recursion.

Function	Description
`count(element)`	Returns the number of times the element exists on the list on which the `count()` function is called.

Table 12.1 Chapter 12 reference.