14.1 Recursive functions

A function may call other functions, including calling itself. A function that calls itself is known as a ***recursive function***. The following program illustrates.

PARTICIPATION ACTIVITY 14.1.1: A recursive function example.

Animation captions:

1. count_down is called and count = 2.
2. count_down is recursively called and count = 1.
3. count_down is recursively called and count = 0.

The function is mostly useful for demonstrating recursion; counting down is easily done instead of using a loop. Each call to count_down creates a new namespace for the local scope of the function. The script makes the first call to count_down(), creating a namespace with the count argument bound to the integer value 2. That first function call prints 2, and calls count_down() with an argument of 1. A new namespace is created again for the local variables in count_down()'s local scope with the count argument bound to the integer value 1. That second function call prints 1, and calls count_down() with an argument of 0. That third function call prints GO!, and then because count == 0 is true, returns. The second function call is then done so it returns. The first function call is then done so it returns. Finally, the script finishes.

PARTICIPATION ACTIVITY 14.1.2: Recursive functions.

1) How many times is count_down() called if the script calls count_down(5)?

Check Show answer

2)

How many times is count_down() called if the script calls count_down(0)?

Check Show answer

3) Is there a difference in how we define the parameters of a recursive versus non-recursive function? Answer yes or no.

Check Show answer

CHALLENGE ACTIVITY 14.1.1: Calling a recursive function.

Write a statement that calls the recursive function backwards_alphabet() with input starting_letter.

Sample output with input: 'f'

```
f
e
d
c
b
a
```

Run

View solution (Instructors only)

Download student submissions

14.2 Recursive algorithm: Search

An algorithm is a sequence of steps for solving a problem. For example, an algorithm for making lemonade is:

- Make lemonade
  - Add sugar to pitcher
  - Add lemon juice
  - Add water
  - Stir

Each step is distinct. Alternatively, an algorithm, for mowing the lawn is:

- *Mow* the lawn
  - *Mow* the frontyard
    - *Mow* the left front
    - *Mow* the right front
  - *Mow* the backyard
    - *Mow* the left back
    - *Mow* the right back

The mowing algorithm is defined *recursively*, i.e., the mowing algorithm's steps themselves consist of mowing, but of a smaller region.

Consider a guessing game program where a friend thinks of a number from 0-100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly. What algorithm would you use to minimize the number of guesses? An algorithm that simply guesses in increments of 1 -- Is it 0? Is it 1? Is it 2? -- requires too many guesses (50 on average). An algorithm that guesses by 10s and then by 1s -- Is it 10? Higher: Is it 20? Higher: Is it 30? Lower: Is it 21? 22? 23? -- does better but still requires about 10 guesses on average (5 to find the correct tens digit and 5 to guess the correct ones digit). An even better algorithm uses a binary search approach, guessing the midpoint of the range and halving the range after each guess -- Is it 50 (the middle of 0-100)? Lower: Is it 25 (the middle of 0-50)? Higher: Is it 37 (the middle of 25-50)? Lower: Is it 31 (the middle of 25-37). After each guess, the binary search algorithm is applied again, just on a smaller range, i.e., the algorithm is recursive. The following animation illustrates.

PARTICIPATION ACTIVITY 14.2.1: Binary search: A well-known recursive algorithm.

Animation captions:

1. The midpoint of 0 and 100 is 50.
2. 31 is lower than 50, so the window is halved and the midpoint of 0 and 50 is found.
3. 31 is greater than 25, so the window is halved and the midpoint of 25 and 50 is found.
4. 31 is less than 37, so the window is halved and the midpoint of 25 and 37, which is 31, is found.

A recursive function is a natural match for the recursive binary search algorithm. We can define a function find(low, high) whose parameters indicate the low and high sides of the guessing range. The function guesses at the midpoint of the range. If the user says lower, the function calls find(low, mid). If the user says higher, the function calls find(mid+1, high)[Note_mid]. The following program illustrates.

Figure 14.2.1: A recursive function find() carrying out a binary search algorithm.

```
def find(low, high):
    mid = (high + low) // 2  # Midpoint of
low..high
    answer  = input('Is it {}? (l/h/y):
'.format(mid))

    if (answer != 'l') and (answer != 'h'):  #
Base case
        print('Got it!')
    else:
        if answer == 'l':
            find(low, mid)
        else:
            find(mid+1, high)

print('Choose a number from 0 to 100.')
print('Answer with:')
print('   l (your num is lower)')
print('   h (your num is higher)')
print(' any other key (guess is right).')

find(0, 100)
```

```
Choose a number from 0 to
100.
Answer with:
   l (your num is lower)
   h (your num is higher)
 any other key (guess is
right).
Is it 50? (l/h/y): l
Is it 25? (l/h/y): h
Is it 38? (l/h/y): h
Is it 44? (l/h/y): l
Is it 41? (l/h/y): y
Got it!
```


The recursive function has an if-else statement, where the if branch is the end of the recursion, known as the ***base case***. The else part has the recursive calls. Such an if-else pattern is quite common in recursive functions.

Consider the following program, in which a recursive algorithm is used to find an item in a sorted list. This example is for demonstration purposes only, a programmer would be better off using the list.index() or "in" operator to find a specific list element. Consider having a list of attendees at a conference, whose names have been stored in alphabetical order in a list. The following program determines whether a particular person is in attendance.

Figure 14.2.2: Recursively searching a sorted list.

```
Enter person's name: Last,
First: Simpson, Homer
Not found.
...
Enter person's name: Last,
First: Domer, Hugo
Found at position 2.
```

```
def find(lst, item, low, high):
    """
    Finds index of string in list of
strings, else -1.
    Searches only the index range low to
high
    Note: Upper/Lower case characters matter
    """
    range_size = (high - low) + 1
    mid = (high + low) // 2

    if item == lst[mid]:  # Base case 1:
Found at mid
        pos = mid
    elif range_size == 1:  # Base case 2:
Not found
        pos = -1
    else:  # Recursive search: Search lower
or upper half
        if item < lst[mid]:  # Search lower
half
            pos = find(lst, item, low, mid)
        else:  # Search upper half
            pos = find(lst, item, mid+1,
high)

    return pos

attendees = []

attendees.append('Adams, Mary')
attendees.append('Carver, Michael')
attendees.append('Domer, Hugo')
attendees.append('Fredericks, Carlo')
attendees.append('Li, Jie')

name = input("Enter person's name: Last,
First: ")
pos = find(attendees, name, 0,
len(attendees)-1)

if pos >= 0:
    print('Found at position
{}.'.format(pos))
else:
    print('Not found.')
```


The find() function restricts its search to elements within the range "low" to "high". The script passes a range encompassing the entire list, namely 0 to (list length - 1). find() compares to the middle element, returning that element's position if matching. If not matching, then find() checks if the window's size is just one element, returning -1 in that case to indicate the item was not found because there is nothing left to search in the window. If neither of those two base cases are satisfied, then find() uses recursive binary search, recursively searching either the lower or upper half of the range as appropriate. Use the below tool to step through execution of the above program.

In general, any recursive solution can also be done using loops. However, in some cases using a recursive algorithm may make a solution more clear, concise, and understandable. Candidates for recursion are problems that can be reduced into smaller and identical problems, and then solved. Above, the binary search algorithms iteratively reduced the problem by half, eventually reached a base case where the problem could be solved (i.e., the desired element was located).

PARTICIPATION ACTIVITY 14.2.2: Recursive search algorithm.

1) If a sorted list has elements numbers 0 to 50 and the item being searched for happens to be at location 6, how many times will the find() function be called?

Check **Show answer**

2) If an alphabetically sorted list (ascending) has elements numbered 0 to 50, and the item at element 0 is "Bananas", how many recursive calls to find() will be made during the failed search for "Apples"?

Check **Show answer**

3) A list of 5 elements is: A B D E F. A is element 0 and F is element 4. Write each call to find() that would occur when searching for item C. The first is find(0,4).

Check   Show answer

(*Note_mid) Because mid has already been checked, it need not be part of the new window, so mid+1 rather than mid can be used for the window's new low side, or mid-1 for the window's new high side. But the mid-1 can have the drawback of a non-intuitive base case (i.e., mid < low, because if the current window is say 4..5, mid is 4, so the new window would be 4..4-1, or 4..3). We believe range==1 is more intuitive, and thus use mid rather than mid-1. However, we still have to use mid+1 when searching higher, due to integer rounding. In particular, for window 99..100, mid is 99 ((99+100)//2=99.5, rounded to 99 due to truncation of the fraction). So the next window would again be 99..100, and the algorithm would repeat with this window forever. mid+1 prevents the problem, and doesn't miss any numbers because mid was checked and thus need not be part of the window.

14.3 Adding output statements for debugging

Recursive functions can be particularly challenging to debug. Adding output statements can be helpful. Furthermore, an additional trick is to indent the print statements to show the current depth of recursion. The following program adds a parameter indent to a find() function that searches a sorted list for an item. All of the find() function's print statements start with "print indent, ...". The indent variable is typically some number of spaces. The script sets indent to three spaces " ". Each recursive call *adds* three more spaces. Note how the output now clearly shows the recursion depth.

Figure 14.3.1: Output statements can help debug recursive functions, especially if indented based on recursion depth.

```
def find(lst, item, low, high, indent):
    """
    Finds index of string in list of strings,
else -1.
    Searches only the index range low to high
    Note: Upper/Lower case characters matter
    """
    print(indent, 'find() range', low, high)
    range_size = (high - low) + 1
    mid = (high + low) // 2

    if item == lst[mid]:  # Base case 1: Found
at mid
        print(indent, 'Found person.')
        pos = mid
    elif range_size == 1:  # Base case 2: Not
found
        print(indent, 'Person not found.')
        pos = -1
    else:  # Recursive search: Search lower or
upper half
        if item < lst[mid]:  # Search lower
half
            print(indent, 'Searching lower
half.')
            pos = find(lst, item, low, mid,
indent + '   ')
        else:  # Search upper half
            print(indent, 'Searching upper
half.')
            pos = find(lst, item, mid+1, high,
indent + '   ')

    print(indent, 'Returning pos =
{}.'.format(pos))
    return pos

attendees = []

attendees.append('Adams, Mary')
attendees.append('Carver, Michael')
attendees.append('Domer, Hugo')
attendees.append('Fredericks, Carlo')
attendees.append('Li, Jie')

name = input("Enter person's name: Last,
First: ")
pos = find(attendees, name, 0,
len(attendees)-1, '   ')

if pos >= 0:
    print('Found at position {}.'.format(pos))
else:
    print( 'Not found.')
```

```
Enter person's name: Last,
First: Meeks, Stan
    find() range 0 4
    Searching upper half.
       find() range 3 4
       Searching upper half.
          find() range 4 4
          Person not found.
          Returning pos =
-1.
       Returning pos = -1.
    Returning pos = -1.
Not found.
...
Enter person's name: Last,
First: Adams, Mary
    find() range 0 4
    Searching lower half.
       find() range 0 2
       Searching lower half.
          find() range 0 1
          Found person.
          Returning pos = 0.
       Returning pos = 0.
    Returning pos = 0.
Found at position 0.
```


©zyBooks 03/05/20 10:41 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Some programmers like to leave the output statements in the code, commenting them out with "#" when not in use. The statements actually serve as a form of comment. More advanced techniques for handling debug output exist too, such as the *logging* Python standard library (beyond this section's scope).

zyDE 14.3.1: Output statements in a recursive function.

Run the recursive find program having the output statements for debugg "Aaron, Joe", and observe the correct output indicating the person is not an error in the algorithm by changing "pos = -1" to "pos = 0" in the base c is found. Run the program again and notice how the indented print state isolate the error; in particular, note how the "Person not found" output is pos = 0", which may lead one to realize the wrong value is being returned introducing different errors and seeing how the indented print statement

Load default template...

```

def find(lst, item, low, high, indent):
    """
    Finds index of string in list of strings
    Searches only the index range low to hig
    Note: Upper/Lower case characters matte
    """
    print(indent, 'find() range', low, high
    range_size = (high - low) + 1
    mid = (high + low) // 2
    if item == lst[mid]:  # Base case 1: Fou
        print(indent, 'Found person.')
        pos = mid
    elif range_size == 1:  # Base case 2: No
        print(indent, 'Person not found.')
        pos = -1
    else:  # Recursive search: Search lower
        if item < lst[mid]:  # Search lower
            print(indent, 'Searching lower h
            pos = find(lst, item, low, mid
```

Pre-enter any input fo run.

Run

PARTICIPATION ACTIVITY 14.3.1: Recursive function debug statements.

1) The above debug approach requires an extra parameter be passed to indicate the amount of indentation.

- True
- False

2) Each recursive call should add a few spaces to the indent parameter.

- True
- False

3) The function should remove a few spaces from the indent parameter before returning.

- True
- False

14.4 Creating a recursive function

Creating a recursive function can be accomplished in two steps.

- *Write base case* -- Every recursive function must have a case that returns a value without performing a recursive call. That case is called the ***base case***. A programmer may write that part of the function first, and then test. There may be multiple base cases.
- *Write recursive case* -- The programmer then adds the recursive case to the function.

The following illustrates for a simple function that computes the factorial of N (N!). The base case is n=1 -- 1! is 1, which is written and tested. The recursive case is n*nfact(n-1), which is written and tested. *Note:* Factorial is not necessarily a good candidate for a recursive function, because a non-recursive version using a loop is so simple; however, factorial makes a simple example for demonstrating recursion. Actually useful cases for recursion are rarer in Python than for other programming languages, since Python programmers tend to prefer more natural iterative loop structures. Typically, recursion is useful when dealing with data structures of unknown size and connectivity, properties most commonly associated with tree-shaped data structures.

PARTICIPATION ACTIVITY 14.4.1: Writing a recursive function for factorial: First writing the base case, then adding the recursive case.

Animation captions:

1. The base case (non-recursive case) has to be written and tested.
2. The recursive case has to be added and tested.

Before writing a recursive function, a programmer should determine: (1) Whether the problem has a naturally recursive solution, *and* (2) whether that solution is better than a non-recursive solution. For example, computing E = M*C*C doesn't seem to have a natural recursive solution. Computing n! (n factorial) does have a natural recursive solution, but a recursive solution is not better than a non-recursive solution that simply uses a loop, as in `for i in range(n, 0, -1): result *= i`[factorial] Binary search has a natural recursive solution, and that solution may be easier to understand than a non-recursive solution.

A common error is to not cover all possible base cases in a recursive function. Another common error is to write a recursive function that doesn't always reach a base case. Both errors may lead to infinite recursion, causing the program to fail.

Commonly, programmers will use two functions for recursion. An "outer" function is intended to be called from other parts of the program, like the function "factorial(n)". An "inner" function is intended only to be called from that outer function, like the function " _factorial(n)" (note the "_"). The outer function may check for a valid input value, e.g., ensuring n is not negative, and then calling the inner function. Commonly, the inner function has parameters that are mainly of use as part of the recursion, and need not be part of the outer function, thus keeping the outer function more intuitive.

PARTICIPATION ACTIVITY 14.4.2: Creating a recursive function.

1) A recursive function with parameter n counts up from any negative number to 0. An appropriate base case would be n==0.

- True
- False

2) A recursive function can have two base cases, such as n==0 returning 0, and n==1 returning 1.

- True
- False

3) n factorial (n!) is commonly implemented as a recursive function due to being easier to understand and executing faster than a loop implementation.

- True
- False

CHALLENGE ACTIVITY 14.4.1: Recursive function: Writing the base case.

Add an if branch to complete double_pennies()'s base case.

Sample output with inputs: 1 10

```
Number of pennies after 10 days: 1024
```

Note: If the submitted code has an infinite loop, the system will stop running the code after a few seconds, and report "Program end never reached." The system doesn't print the test case that caused the reported message.

Run

View solution (*Instructors only*)

Download student submissions

CHALLENGE ACTIVITY

14.4.2: Recursive function: Writing the recursive case.

Write code to complete print_factorial()'s recursive case.

Sample output with input: 5

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

```python
def print_factorial(fact_counter, fact_value):
    output_string = ''

    if fact_counter == 0:      # Base case: 0! = 1
        output_string += '1'
    elif fact_counter == 1:    # Base case: print 1 and result
        output_string += str(fact_counter) +  ' = ' + str(fact_value)
    else:                       # Recursive case
        output_string += str(fact_counter) + ' * '
        next_counter = fact_counter - 1
        next_value = next_counter * fact_value
        output_string += ''' Your solution goes here '''

    return output_string

user_val = int(input())
print('{}! = '.format(user_val), end="")
print(print_factorial(user_val, user_val))
```


Run

View solution (Instructors only)

Download student submissions

(*factorial) In this discussion, we ignore the fact that the math module has a very convenient math.factorial(n) function.

14.5 Recursive math functions

Recursive functions can be used to solve certain math problems, such as computing the Fibonacci sequence. The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc. The pattern is to compute the next number by adding the previous two numbers. The sequence starts with 0 and 1.

Below is a program that outputs the Fibonacci sequence step-by-step for a user-entered number of steps. The program starts after the first 0 and 1 of the Fibonacci sequence. The base case is that the program has output the requested number of steps. The recursive case computes the next step.

Figure 14.5.1: Fibonacci sequence step-by-step.

```
This program outputs the
Fibonacci sequence step-by-
step,
starting after the first 0
and 1.

How many steps would you
like?10
0 + 1 = 1
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13
8 + 13 = 21
13 + 21 = 34
21 + 34 = 55
34 + 55 = 89
```

```
"""
Output the Fibonacci sequence step-by-step.
Fibonacci sequence starts as:
0 1 1 2 3 5 8 13 21 ... in which the first
two numbers are 0 and 1 and each additional
number is the sum of the previous two numbers
"""
def fibonacci(v1, v2, run_cnt):
    print(v1, '+', v2, '=', v1+v2)

    if run_cnt <= 1:  # Base case:
                      # Ran for user's number of
steps
        pass  # Do nothing
    else:             # Recursive case
        fibonacci(v2, v1+v2, run_cnt-1)


print ('This program outputs the\n'
       'Fibonacci sequence step-by-step,\n'
       'starting after the first 0 and 1.\n')

run_for = int(input('How many steps would you
like?'))

fibonacci(0, 1, run_for)
```


zyDE 14.5.1: Recursive Fibonacci.

Write a program that outputs the nth Fibonacci number, where n is a use
if the user enters 4, the program should output 3 (without outputting the
Use a recursive function compute_nth_fib that takes n as a parameter ar
Fibonacci number. The function has two base cases: input 0 returns 0, ai

Load default template... Run

Recursion can be used to solve the greatest common divisor (GCD) problem. The GCD is the largest number that divides evenly into two numbers, e.g. GCD(12, 8) = 4. A simple algorithm to compute the GCD subtracts the smaller number from the larger number until both numbers are equal. For example, GCD(12, 8) = GCD(12-8=4, 8) = GCD(4, 8-4=4). The equal numbers are the GCD. Euclid described this algorithm around 300 BC.

The below program recursively computes the GCD of two numbers. The base case is that the two numbers are equal, so that number is returned. The recursive case subtracts the smaller number from the larger number and then calls GCD with the new pair of numbers.

Figure 14.5.2: Calculate greatest common divisor of two numbers.

```
"""
Determine the greatest common
divisor
of two numbers, e.g. GCD(8, 12) = 4
"""

def gcd(n1, n2):
    if n1 % n2 == 0:            # n2
is a common factor
        return n2
    else:
        return gcd(n2,n1%n2)


print ('This program outputs the
greatest '
       'common divisor of two
numbers.\n')

num1 = int(input('Enter first
number: '))
num2 = int(input('Enter second
number: '))

if (num1 < 1) or (num2 < 1):
    print('Note: Neither value can
be below 1.')
else:
    my_gcd = gcd(num1, num2)
    print('Greatest common divisor
=', my_gcd)
```

```
def compute_nth_fib(num):
    # if base case ...
        # return base case value ...
    # else ...
        # recursively call compute_nth_fib().
```

```
This program outputs the greatest
common divisor of two numbers.

Enter first number:12
Enter second number:8
Greatest common divisor = 4
...
This program outputs the greatest
common divisor of two numbers.

Enter first number:456
Enter second number:784
Greatest common divisor = 8
```


The ***depth*** of recursion is a measure of how many recursive calls of a function have been made, but have not yet returned. Each recursive call requires the Python interpreter to allocate more memory, and eventually all of the system memory could be used. Thus, a recursion depth limit exists, accessible using the function sys.getrecursionlimit(). The default recursion depth limit is typically 1000. The limit can be changed use sys.setrecursionlimit(). Exceeding the depth limit causes a RuntimeError to occur; for example, calling gcd() above with very large numbers can cause more than 1000 recursive calls:

Figure 14.5.3: Limit on recursion depth.

```
This program outputs the greatest common divisor of two numbers.

Enter first number:213432189235798743
Enter second number:213004998585443

Traceback (most recent call last):
  File "test.py", line 28, in <module>
    my_gcd = gcd(num1, num2)
  File "test.py", line 14, in gcd
    greatest_common_divisor = gcd(n1-n2, n2)
  File "test.py", line 14, in gcd
    greatest_common_divisor = gcd(n1-n2, n2)

... (repeated 1000 times)

  File "test.py", line 9, in gcd
    if n1 == n2:  # Base case: Numbers are equal
RuntimeError: maximum recursion depth exceeded in comparison
```


PARTICIPATION ACTIVITY 14.5.1: Recursive GCD.

1) How many calls are made to the gcd function for gcd(12, 8)?

Check Show answer

2) How many calls are made to the gcd function for gcd(5, 3)?

Check Show answer

Exploring further:

- More on the Fibonacci sequence from wikipedia.org
- More on the GCD algorithm from wikipedia.org

CHALLENGE ACTIVITY 14.5.1: Writing a recursive math function.

Write code to complete raise_to_power(). Note: This example is for practicing recursion; a non-recursive function, or using the built-in function math.pow(), would be more common.

Sample output with inputs: 4 2


```
4^2 = 16
```

```
def raise_to_power(base_val, exponent_val):
   if exponent_val == 0:
      result_val = 1
   else:
      result_val = base_val * ''' Your solution goes here '''

   return result_val

user_base = int(input())
user_exponent = int(input())

print('{}^{} = {}'.format(user_base, user_exponent,
      raise_to_power(user_base, user_exponent)))
```

Run

View solution (*Instructors only*)

Download student submissions

14.6 Recursive exploration of all possibilities

Recursion is a powerful tool for exploring all possibilities, such as all possible reorderings of a word's letters, all possible subsets of items, all possible paths between cities, etc. This section provides several examples of using recursion for such exploration.

Consider the problem of printing all possible combinations (or "scramblings") of a word's letters. For example, the letters of "abc" can be scrambled in 6 ways: abc, acb, bac, bca, cab, cba. Those possibilities can be obtained by thinking of three choices: Choosing the first letter ("a", "b", or "c"), then choosing the second letter (if "a" was the first choice, then second possible choices are "b" or "c"; if "b" was the first choice, then second possible choices are "a" and "c"; etc.), then choosing the third letter. The choices can be depicted using a tree. Each level represents a choice. Each node in the tree shows the unchosen letters on the left, and the chosen letters on the right, as in the animation figure below.

Such a tree forms the basis for a recursive exploration function to generate all possible combinations of a string's letters. The function will take two parameters, one for the unchosen letters, and one for the already chosen letters. The base case will be when no letters exist in the unchosen letters, in which case the chosen letters are printed. The recursive case will call the function once for each letter in the unchosen letters. The following animation depicts how such a recursive algorithm would traverse the tree. The leaves of the tree (the bottommost nodes) represent the base case.

PARTICIPATION ACTIVITY 14.6.1: Exploring all possibilities viewed as a tree of choices.

Animation captions:

1. "a" is chosen from "abc", then "b" is chosen from "bc". Finally, "c" is chosen from "c".
2. "b" has already been chosen from "bc". "c" can also be chosen. "acb" is chosen from "b".
3. "b" is chosen from "abc".
4. "c" is chosen from "abc".

The program below receives a word from the user then jumbles all of its letters in to every possible ordering. The base case is that all letters have been used. In the recursive case, a remaining letter is moved to the scrambled letters, recursively explored, then put back. This is done for each remaining letter.

Figure 14.6.1: Scramble a word's letters in every possible way.

```
def scramble(r_letters, s_letters):
    """
    Output every possible combination of a word.
    Each recursive call moves a letter from
    r_letters (remaining letters) to
    s_letters (scrambled letters)
    """
    if len(r_letters) == 0:
        # Base case: All letters used
        print(s_letters)
    else:
        # Recursive case: For each call to
scramble()
        # move a letter from remaining to scrambled
        for i in range(len(r_letters)):
            # The letter at index i will be
scrambled
            scramble_letter = r_letters[i]

            # Remove letter to scramble from
remaining letters list
            remaining_letters = r_letters[:i] +
r_letters[i+1:]

            # Scramble letter
            scramble(remaining_letters, s_letters +
scramble_letter)

word = input('Enter a word to be scrambled: ')
scramble(word, '')
```



```
Enter a word to be
scrambled: cat
cat
cta
act
atc
tca
tac
```

Recursion is useful for finding all possible subsets of a set of items. The following example is a shopping spree in which you may select a 3-item subset from a larger set of items. The program should print all possible 3-item subsets given the larger set. The program also happens to print the total price value of those items.

The ShoppingBagCombinations() function has a parameter for the current bag contents, and a parameter for the remaining items from which to choose. The base case is that the current bag already has 3 items. The recursive case is to move one of the remaining items to the bag, recursively call the function, then move the item back from the bag to the remaining items.

Figure 14.6.2: Shopping spree in which you can fit 3 items in your shopping bag.

```
max_items_in_bag = 3

def shopping_bag_combinations(curr_bag,
remaining_items):
    """
    Output every combination of items that fit
    in a shopping bag. Each recursive call moves
    one item into the shopping bag.
    """
    if len(curr_bag) == max_items_in_bag:
        # Base case: Shopping bag full
        bag_value = 0
        for item in curr_bag:
            bag_value += item['price']
            print(item['name'], ' ', end=' ')
        print('=', bag_value)
    else:
        # Recursive case: Move one of the remaining
items
        # to the shopping bag.
        for index, item in enumerate(remaining_items):
            # Move item into bag
            curr_bag.append(item)
            remaining_items.pop(index)

            shopping_bag_combinations(curr_bag,
remaining_items)

            # Take item out of bag
            remaining_items.insert(index, item)
            curr_bag.pop()

items = [
    {
        'name': 'Milk',
        'price': 1.25
    },
    {
        'name': 'Belt',
        'price': 23.55
    },
    {
        'name': 'Toys',
        'price': 19.05
    },
    {
        'name': 'Cups',
        'price': 11.85
    }
]

bag = []
shopping_bag_combinations(bag, items)
```

```
Milk   Belt   Toys   =
43.85
Milk   Belt   Cups   =
36.65
Milk   Toys   Belt   =
43.85
Milk   Toys   Cups   =
32.15
Milk   Cups   Belt   =
36.65
Milk   Cups   Toys   =
32.15
Belt   Milk   Toys   =
43.85
Belt   Milk   Cups   =
36.65
Belt   Toys   Milk   =
43.85
Belt   Toys   Cups   =
54.45
Belt   Cups   Milk   =
36.65
Belt   Cups   Toys   =
54.45
Toys   Milk   Belt   =
43.85
Toys   Milk   Cups   =
32.15
Toys   Belt   Milk   =
43.85
Toys   Belt   Cups   =
54.45
Toys   Cups   Milk   =
32.15
Toys   Cups   Belt   =
54.45
Cups   Milk   Belt   =
36.65
Cups   Milk   Toys   =
32.15
Cups   Belt   Milk   =
36.65
Cups   Belt   Toys   =
54.45
Cups   Toys   Milk   =
32.15
Cups   Toys   Belt   =
54.45
```


Recursion is useful for finding all possible paths. In the following example, a salesman must travel to 3 cities: Boston, Chicago, and Los Angeles. The salesman wants to know all possible paths among those three cities, starting from any city. A recursive exploration of all travel paths can be used. The base case is that the salesman has traveled to all cities. The recursive case is to travel to a new city, explore possibilities, then return to the previous city.

Figure 14.6.3: Find distance of traveling to 3 cities.


```
num_cities = 3
city_names = []
distances = []

def travel_paths(curr_path, need_to_visit):
    if len(curr_path) == num_cities:  # Base case:
Visited all cities
        total_distance = 0
        for i in range(len(curr_path)):
            print(city_names[curr_path[i]], '  ',
end=' ')

            if i > 0:
                total_distance +=
distances[curr_path[i-1]][curr_path[i]]

        print('=', total_distance)
    else:  # Recursive case: Travel to each city
        for i in range(len(need_to_visit)):
            # Visit city
            city = need_to_visit[i]
            need_to_visit.pop(i)
            curr_path.append(city)

            travel_paths(curr_path, need_to_visit)

            # Take item out of bag
            need_to_visit.insert(i, city)
            curr_path.pop()

distances.append([0])
distances[0].append(960)  # Boston-Chicago
distances[0].append(2960) # Boston-Los Angeles
distances.append([960])   # Chicago Boston
distances[1].append(0)
distances[1].append(2011) # Chicago-Los Angeles
distances.append([2960])  # Los Angeles-Boston
distances[2].append(2011) # Los Angeles-Chicago
distances[2].append(0)

city_names = ["Boston", "Chicago", "Los Angeles"]

path = []
need_to_visit = [0, 1, 2] # (Need to visit all 3
cities)
travel_paths(path, need_to_visit)
```

```
Boston    Chicago
Los Angeles    = 2971
Boston    Los Angeles
Chicago    = 4971
Chicago    Boston
Los Angeles    = 3920
Chicago    Los Angeles
Boston    = 4971
Los Angeles    Boston
Chicago    = 3920
Los Angeles    Chicago
Boston    = 2971
```

PARTICIPATION ACTIVITY 14.6.2: Recursive exploration.

1) What is the output of: scramble("xy", "")? Determine your answer by manually tracing the code, not by running the program.

Check Show answer

2) You wish to generate all possible 3-letter subsets from the letters in an N-letter word (N>3). Which of the above recursive functions is the closest (just enter the function's name)?

Check Show answer

Exploring further:

- More on recursion trees from Wikipedia.org.

14.7 LAB: All permutations of names

Write a program that lists all ways people can line up for a photo (all permutations of a list of strings). The program will read a list of one word names, then use a recursive method to create and output all possible orderings of those names, one ordering per line.

When the input is:

```
Julia Lucas Mia
```


then the output is (must match the below ordering):

```
Julia Lucas Mia
Julia Mia Lucas
Lucas Julia Mia
Lucas Mia Julia
Mia Julia Lucas
Mia Lucas Julia
```

LAB ACTIVITY 14.7.1: LAB: All permutations of names 0 / 10

main.py Load default template...

```
def all_permutations(permList, nameList):
    # TODO: Implement method to create and output all permutations of the list of na

if __name__ == "__main__":
    nameList = input().split(' ')
    permList = []
    all_permutations(permList, nameList)
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → Output (shown be

Program output displayed here

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**

14.8 LAB: Number pattern

Write a recursive function called print_num_pattern() to output the following number pattern.

Given a positive integer as input (Ex: 12), subtract another positive integer (Ex: 3) continually until 0 or a negative value is reached, and then continually add the second integer until the first integer is again reached.

Ex. If the input is:

```
12
3
```

the output is:

```
12 9 6 3 0 3 6 9 12
```


LAB ACTIVITY 14.8.1: LAB: Number pattern 0 / 10

main.py Load default template...

```python
# TODO: Write recursive print_num_pattern() function

if __name__ == "__main__":
    num1 = int(input())
    num2 = int(input())
    print_num_pattern(num1, num2)
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → Output (shown be

Program output displayed here

Lab statistics and submissions

Show

Solution

Show

Tests

Show