

CIS 61 :: Lab 04 - Recursion - Template

Student Name:

Instructions: Use Recursion to find solutions to below functions. Attached screenshots of your code and your test run. Make sure the screenshots are readable.

Q1: Skip Add

Write a function `skip_add` that takes a single argument `n` and computes the sum of every other integer between `0` and `n`. Assume `n` is non-negative.

```
Question1.py
1 def skip_add(n):
2     """ Takes a number x and returns x + x-2 + x-4 + x-6 + ... + 0.
3     """
4     >>> skip_add(5) # 5 + 3 + 1 + 0
5     9
6     >>> skip_add(10) # 10 + 8 + 6 + 4 + 2 + 0
7     30
8     >>> # Do not use while/for loops!
9     """
10    if n<=0:
11        return 0
12    else:
13        return n+skip_add(n-2)

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>python -m doctest Question1.py -v
Trying:
    skip_add(5) # 5 + 3 + 1 + 0
Expecting:
    9
ok
Trying:
    skip_add(10) # 10 + 8 + 6 + 4 + 2 + 0
Expecting:
    30
ok
1 item had no tests:
    Question1
1 item passed all tests:
    2 tests in Question1.skip_add
2 tests in 2 items.
2 passed.
Test passed.
C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>
```

Code:

Test Run:

Q2: Hailstone

Recall the `hailstone` function from Lab 2. First, pick a positive integer `n` as the start. If `n` is even, divide it by 2. If `n` is odd, multiply it by 3 and add 1. Repeat this process until `n` is 1. Write a recursive version of `hailstone` that prints out the values of the sequence and returns the number of steps.

Hint: When taking the recursive leap of faith, consider both the return value and side effect of this function.

```
Question1.py  Question2.py
1 def hailstone(n):
2     """ Print out the hailstone sequence starting at n, and return the number of elements in the sequence.
3     """
4     >>> a = hailstone(10)
5     10
6     5
7     16
8     8
9     4
10    2
11    1
12    >>> a
13    7
14    """
15    print (n)
16    if (n==1):
17        return 1
18    elif (n%2==0):
19        return 1 + hailstone(n//2)
20    else:
21        return 1 + hailstone(n*3+1)

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>python -m doctest Question2.py -v
Trying:
    a = hailstone(10)
Expecting:
    10
    5
    16
    8
    4
    2
    1
ok
Trying:
    a
Expecting:
    7
ok
1 item had no tests:
    Question2
1 item passed all tests:
    2 tests in Question2.hailstone
2 tests in 2 items.
2 passed.
Test passed.
C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>
```

Q3: Summation

Write a recursive implementation of `summation`, which takes a positive integer `n` and a function `term`. It applies `term` to

```
Question3.py
1 def summation(n, term):
2     """Return the sum of the first n terms in the sequence defined by term.
3     Implement using recursion!
4     """
5     >>> summation(5, lambda x: x * x * x) # 1^3 + 2^3 + 3^3 + 4^3 + 5^3
6     225
7     >>> summation(9, lambda x: x + 1) # 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
8     54
9     >>> summation(5, lambda x: 2**x) # 2^1 + 2^2 + 2^3 + 2^4 + 2^5
10    62
11    >>> # Do not use while/for loops!
12    """
13    if n==0:
14        return 0
15    else:
16        return term(n) + summation(n-1, term)

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>python -m doctest Question3.py -v
Trying:
    summation(5, lambda x: x * x * x) # 1^3 + 2^3 + 3^3 + 4^3 + 5^3
Expecting:
    225
ok
Trying:
    summation(9, lambda x: x + 1) # 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
Expecting:
    54
ok
Trying:
    summation(5, lambda x: 2**x) # 2^1 + 2^2 + 2^3 + 2^4 + 2^5
Expecting:
    62
ok
1 item had no tests:
    Question3
1 item passed all tests:
    3 tests in Question3.summation
3 tests in 2 items.
3 passed.
Test passed.
C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>
```

Q4: Is Prime

Write a function `is_prime` that takes a single argument `n` and returns `True` if `n` is a prime number and `False` otherwise. Assume `n > 1`. We implemented this iteratively before, now time to do it recursively!

Hint: You will need a helper function! Remember helper functions are useful if you need to keep track of more variables than the given parameters, or if you need to change the value of the input

```
Question4.py
1 def is_prime(n):
2     """Returns True if n is a prime number and False otherwise.
3     """
4     >>> is_prime(2)
5     True
6     >>> is_prime(16)
7     False
8     >>> is_prime(521)
9     True
10    """
11    def checker(n, d):
12        if d==1:
13            return True;
14        elif (n%d==0):
15            return False
16        else:
17            return checker(n,d-1)
18
19    if n<=1:
20        return False
21    else:
22        return checker(n,n-1)

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>python -m doctest Question4.py -v
Trying:
    is_prime(2)
Expecting:
    True
ok
Trying:
    is_prime(16)
Expecting:
    False
ok
Trying:
    is_prime(521)
Expecting:
    True
ok
1 item had no tests:
    Question4
1 item passed all tests:
    3 tests in Question4.is_prime
3 tests in 2 items.
3 passed.
Test passed.

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>
```

Q5: GCD

The greatest common divisor of two positive integers `a` and `b` is the largest integer which evenly divides both numbers (with no remainder). Euclid, a Greek mathematician in 300 B.C., realized that the greatest common divisor of `a` and `b` is one of the following:

- the smaller value if it evenly divides the larger value, or
- the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value

In other words, if `a` is greater than `b` and `a` is not divisible by `b`, then

$$\text{gcd}(a, b) = \text{gcd}(b, a \% b)$$

Write the `gcd` function recursively using Euclid's algorithm.

```
Question5.py
1 def gcd(a, b):
2     """ Returns the greatest common divisor of a and b.
3     Should be implemented using recursion.
4     """
5     >>> gcd(34, 19)
6     1
7     >>> gcd(39, 91)
8     13
9     >>> gcd(20, 30)
10    10
11    >>> gcd(40, 40)
12    40
13    """
14    if(b==0):
15        return a
16    else:
17        return gcd(b, a % b)

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>python -m doctest Question5.py -v
Trying:
    gcd(34, 19)
Expecting:
    1
ok
Trying:
    gcd(39, 91)
Expecting:
    13
ok
Trying:
    gcd(20, 30)
Expecting:
    10
ok
Trying:
    gcd(40, 40)
Expecting:
    40
ok
1 item had no tests:
    Question5
1 item passed all tests:
    4 tests in Question5.gcd
4 tests in 2 items.
4 passed.
Test passed.

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>
```

Q6: Count Stairs

You want to go up a flight of stairs that has n steps. You can either take 1 or 2 steps each time. How many different ways can you go up this flight of stairs? Write a function `count_stair_ways` that solves this problem. Assume n is positive.

Before we start, what's the base case for this question? What is the simplest input?

What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

Use those two recursive calls to write the recursive case:

```
Question6.py
1 def count_stair_ways(n):
2     """
3     >>> count_stair_ways(1)
4     1
5     >>> count_stair_ways(2)
6     2
7     >>> count_stair_ways(3)
8     3
9     >>> count_stair_ways(4)
10    5
11    >>> count_stair_ways(5)
12    8
13    """
14    if n < 0:
15        return 0
16    elif (n == 0):
17        return 1
18    else:
19        return count_stair_ways(n-1) + count_stair_ways(n-2)

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>python -m doctest Question6.py -v
Trying:
    count_stair_ways(1)
Expecting:
    1
ok
Trying:
    count_stair_ways(2)
Expecting:
    2
ok
Trying:
    count_stair_ways(3)
Expecting:
    3
ok
Trying:
    count_stair_ways(4)
Expecting:
    5
ok
Trying:
    count_stair_ways(5)
Expecting:
    8
ok
1 item had no tests:
    Question6
1 item passed all tests:
    5 tests in Question6.count_stair_ways
5 tests in 2 items.
5 passed.
Test passed.
```

Q7: Count Stairs with k steps

Consider a special version of the `count_stairways` problem, where instead of taking 1 or 2 steps, we are able to take up to and including k steps at a time.

Write a function `count_k` that figures out the number of paths for this scenario. Assume n and k are positive.

Tip: You may need to use a while loop in your solutions.

```
Question7.py
1 def count_k(n, k):
2     """
3     >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
4     4
5     >>> count_k(4, 4)
6     8
7     >>> count_k(10, 3)
8     274
9     >>> count_k(300, 1) # Only one step at a time
10    1
11    """
12    def ranger(stairs, stepSize):
13        if (stairs < 0 or stepSize > k):
14            return 0
15        elif (stairs == 0):
16            return 1
17        else:
18            return ranger(stairs - stepSize, 1) + ranger(stairs, stepSize + 1)
19    return ranger(n, 1)

C:\Users\Ben\OneDrive\Desktop\CIS Repos\CIS61A\Lab4\Code>python -m doctest Question7.py -v
Trying:
    count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
Expecting:
    4
ok
Trying:
    count_k(4, 4)
Expecting:
    8
ok
Trying:
    count_k(10, 3)
Expecting:
    274
ok
Trying:
    count_k(300, 1) # Only one step at a time
Expecting:
    1
ok
1 item had no tests:
    Question7
1 item passed all tests:
    4 tests in Question7.count_k
4 tests in 2 items.
4 passed.
Test passed.
```