Lab 12: Final Review | CS 61A Spring 2020

λ inst.eecs.berkeley.edu/~cs61a/sp20/lab/lab12

Lab 12: Final Review
lab12.zip

Due by 11:59pm on Friday, April 24.

Starter Files

Download lab12.zip. Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok autograder.

Submission

By the end of this lab, you should have submitted the lab with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be graded. Check that you have successfully submitted your code on okpy.org.

Note

Note: We will be releasing lab solutions Wednesday morning (pre-deadline)! You still have to turn in the lab to get credit, however. (It's fine to copy/paste the staff solution once it's released, if you don't have time, but please try to read over the solutions if you do this.)

Review Topics

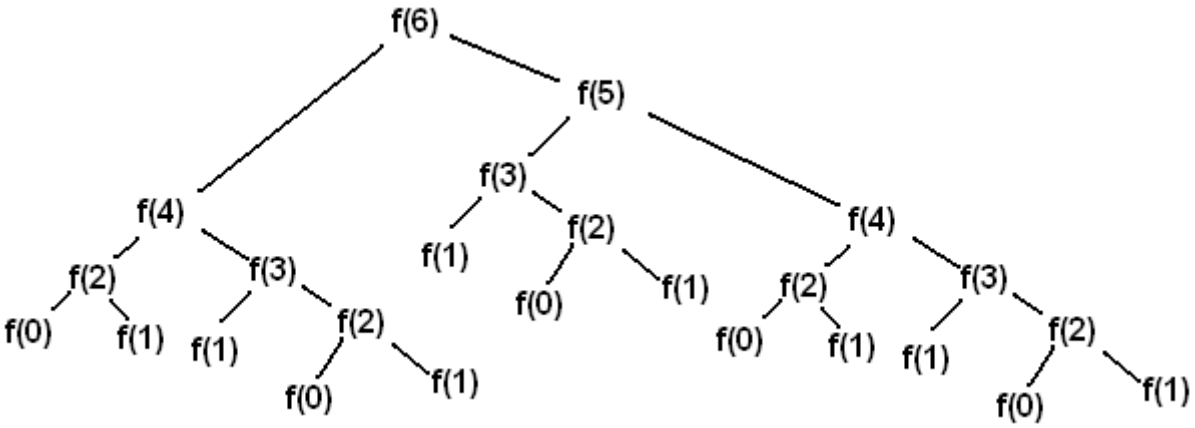
Tree Recursion

A tree recursive function is a recursive function that makes more than one call to itself, resulting in a tree-like series of calls.

A classic example of a tree recursion function is finding the nth Fibonacci number:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Calling `fib(6)` results in the following call structure (where `f` is `fib`):



Each `f(i)` node represents a recursive call to `fib`. Each recursive call makes another two recursive calls. `f(0)` and `f(1)` do not make any recursive calls because they are the base cases of the function. Because of these base cases, we are able to terminate the recursion and beginning accumulating the values.

Generally, tree recursion is effective when you want to explore multiple possibilities or choices at a single step. In these types of problems, you make a recursive call for each choice or for a group of choices. Here are some examples:

- Given a list of paid tasks and a limited amount of time, which tasks should you choose to maximize your pay? This is actually a variation of the Knapsack problem, which focuses on finding some optimal combination of different items.
- Suppose you are lost in a maze and see several different paths. How do you find your way out? This is an example of path finding, and is tree recursive because at every step, you could have multiple directions to choose from that could lead out of the maze.
- Your dryer costs $2 per cycle and accepts all types of coins. How many different combinations of coins can you create to run the dryer? This is similar to the partitions problem from the textbook.

Trees

Recall that a tree is a recursive abstract data type that has a `label` (the value stored in the root of the tree) and `branches` (a list of trees directly underneath the root).

We saw one way to implement the tree ADT -- using constructor and selector functions that treat trees as lists. Another, more formal, way to implement the tree ADT is with a class. Here is part of the class definition for `Tree`, which can be found in `lab07.py`:

```
class Tree:
    """
    >>> t = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
    >>> t.label
    3
    >>> t.branches[0].label
    2
    >>> t.branches[1].is_leaf()
    True
    """
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches
```

Even though this is a new implementation, everything we know about the tree ADT remains true. That means that solving problems involving trees as objects uses the same techniques that we developed when first studying the tree ADT (e.g. we can still use recursion on the branches!). The main difference, aside from syntax, is that tree objects are mutable.

Here is a summary of the differences between the tree ADT implemented using functions and lists vs. implemented using a class:

-	**Tree constructor and selector functions**	**Tree class**
Constructing a tree	To construct a tree given a `label` and a list of `branches`, we call `tree(label, branches)`	To construct a tree object given a `label` and a list of `branches`, we call `Tree(label, branches)` (which calls the `Tree.__init__` method)
Label and branches	To get the label or branches of a tree `t`, we call `label(t)` or `branches(t)` respectively	To get the label or branches of a tree `t`, we access the instance attributes `t.label` or `t.branches` respectively
Mutability	The tree ADT is immutable because we cannot assign values to call expressions	The `label` and `branches` attributes of a `Tree` instance can be reassigned, mutating the tree
Checking if a tree is a leaf	To check whether a tree `t` is a leaf, we call the convenience function `is_leaf(t)`	To check whether a tree `t` is a leaf, we call the bound method `t.is_leaf()`. This method can only be called on `Tree` objects.

Linked Lists

We've learned that a Python list is one way to store sequential values. Another type of list is a linked list. A Python list stores all of its elements in a single object, and each element can be accessed by using its index. A linked list, on the other hand, is a recursive object that only stores two things: its first value and a reference to the rest of the list, which is another linked list.

We can implement a class, `Link`, that represents a linked list object. Each instance of `Link` has two instance attributes, `first` and `rest`.

```
class Link:
    """A linked list.

    >>> s = Link(1)
    >>> s.first
    1
    >>> s.rest is Link.empty
    True
    >>> s = Link(2, Link(3, Link(4)))
    >>> s.first = 5
    >>> s.rest.first = 6
    >>> s.rest.rest = Link.empty
    >>> s                                    # Displays the contents of repr(s)
    Link(5, Link(6))
    >>> s.rest = Link(7, Link(Link(8, Link(9))))
    >>> s
    Link(5, Link(7, Link(Link(8, Link(9)))))
    >>> print(s)                             # Prints str(s)
    <5 7 <8 9>>
    """
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

A valid linked list can be one of the following:

1. An empty linked list (`Link.empty`)

2. A `Link` object containing the first value of the linked list and a reference to the rest of the linked list

What makes a linked list recursive is that the `rest` attribute of a single `Link` instance is another linked list! In the big picture, each `Link` instance stores a single value of the list. When multiple `Link`s are linked together through each instance's `rest` attribute, an entire sequence is formed.

> *Note*: This definition means that the `rest` attribute of any `Link` instance *must* be either `Link.empty` or another `Link` instance! This is enforced in `Link.__init__`, which raises an `AssertionError` if the value passed in for `rest` is neither of these things.

To check if a linked list is empty, compare it against the class attribute `Link.empty`. For example, the function below prints out whether or not the link it is handed is empty:

```
def test_empty(link):
    if link is Link.empty:
        print('This linked list is empty!')
    else:
        print('This linked list is not empty!')
```

OOP

Object-oriented programming (OOP) is a style of programming that allows you to think of code in terms of "objects." Here's an example of a `Car` class:

```
class Car(object):
    num_wheels = 4

    def __init__(self, color):
        self.wheels = Car.num_wheels
        self.color = color

    def drive(self):
        if self.wheels <= Car.num_wheels:
            return self.color + ' car cannot drive!'
        return self.color + ' car goes vroom!'

    def pop_tire(self):
        if self.wheels > 0:
            self.wheels -= 1
```

Here's some terminology:

- **class**: a blueprint for how to build a certain type of object. The `Car` class (shown above) describes the behavior and data that all `Car` objects have.

- **instance**: a particular occurrence of a class. In Python, we create instances of a class like this:

  ```
  >>> my_car = Car('red')
  ```

  `my_car` is an instance of the `Car` class.

- **attribute** or **field**: a variable that belongs to the class. Think of an attribute as a quality of the object: cars have *wheels* and *color*, so we have given our `Car` class `self.wheels` and `self.color` attributes. We can access attributes using **dot notation**:

  ```
  >>> my_car.color
  'red'
  >>> my_car.wheels
  4
  ```

- **method**: Methods are just like normal functions, except that they are tied to an instance or a class. Think of a method as a "verb" of the class: cars can *drive* and also *pop their tires*, so we have given our `Car` class the methods `drive` and `pop_tire`. We call methods using **dot notation**:

  ```
  >>> my_car = Car('red')
  >>> my_car.drive()
  'red car goes vroom!'
  ```

- **constructor**: As with data abstraction, constructors describe how to build an instance of the class. Most classes have a constructor. In Python, the constructor of the class defined as `__init__`. For example, here is the `Car` class's constructor:

  ```
  def __init__(self, color):
      self.wheels = Car.num_wheels
      self.color = color
  ```

  The constructor takes in one argument, `color`. As you can see, the constructor also creates the `self.wheels` and `self.color` attributes.

- `self`: in Python, `self` is the first parameter for many methods (in this class, we will only use methods whose first parameter is `self`). When a method is called, `self` is bound to an instance of the class. For example:

  ```
  >>> my_car = Car('red')
  >>> car.drive()
  ```

  Notice that the `drive` method takes in `self` as an argument, but it looks like we didn't pass one in! This is because the dot notation *implicitly* passes in `car` as `self` for us.

Required Questions

Trees

Q1: Prune Min

Write a function that prunes a `Tree t` mutatively. `t` and its branches always have zero or two branches. For the trees with two branches, reduce the number of branches from two to one by keeping the branch that has the smaller label value. Do nothing with trees with zero branches.

Prune the tree from the bottom up. The result should be a linear tree.

```
def prune_min(t):
    """Prune the tree mutatively from the bottom up.

    >>> t1 = Tree(6)
    >>> prune_min(t1)
    >>> t1
    Tree(6)
    >>> t2 = Tree(6, [Tree(3), Tree(4)])
    >>> prune_min(t2)
    >>> t2
    Tree(6, [Tree(3)])
    >>> t3 = Tree(6, [Tree(3, [Tree(1), Tree(2)]), Tree(5, [Tree(3), Tree(4)])])
    >>> prune_min(t3)
    >>> t3
    Tree(6, [Tree(3, [Tree(1)])])
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

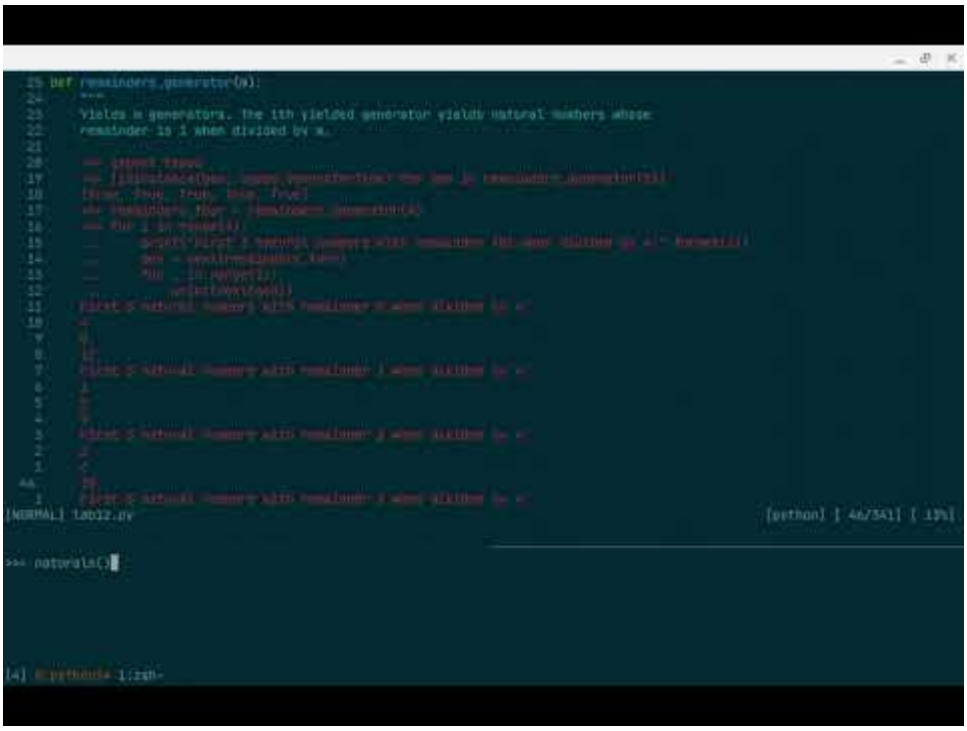
```
python3 ok -q prune_min
```

Q2: Remainder Generator

Like functions, generators can also be *higher-order*. For this problem, we will be writing `remainders_generator`, which yields a series of generator objects. `remainders_generator` takes in an integer `m`, and yields `m` different generators. The first generator is a generator of multiples of `m`, i.e. numbers where the remainder is 0. The second is a generator of natural numbers with remainder 1 when divided by `m`. The last generator yields natural numbers with remainder `m - 1` when divided by `m`.

> *Hint*: You can call the `naturals` function to create a generator of infinite natural numbers.

> *Hint*: Consider defining an inner generator function. Each yielded generator varies only in that the elements of each generator have a particular remainder when divided by `m`. What does that tell you about the argument(s) that the inner function should take in?



Watch Video At: https://youtu.be/l4jEb1o73Ek

```
def remainders_generator(m):
    """
    Yields m generators. The ith yielded generator yields natural numbers whose
    remainder is i when divided by m.

    >>> import types
    >>> [isinstance(gen, types.GeneratorType) for gen in remainders_generator(5)]
    [True, True, True, True, True]
    >>> remainders_four = remainders_generator(4)
    >>> for i in range(4):
    ...     print("First 3 natural numbers with remainder {0} when divided by
4:".format(i))
    ...     gen = next(remainders_four)
    ...     for _ in range(3):
    ...         print(next(gen))
    First 3 natural numbers with remainder 0 when divided by 4:
    4
    8
    12
    First 3 natural numbers with remainder 1 when divided by 4:
    1
    5
    9
    First 3 natural numbers with remainder 2 when divided by 4:
    2
    6
    10
    First 3 natural numbers with remainder 3 when divided by 4:
    3
    7
    11
    """
    "*** YOUR CODE HERE ***"
```

Note that if you have implemented this correctly, each of the generators yielded by `remainder_generator` will be *infinite* - you can keep calling `next` on them forever without running into a `StopIteration` exception.

Use Ok to test your code:

```
python3 ok -q remainders_generator
```

Folding Linked Lists

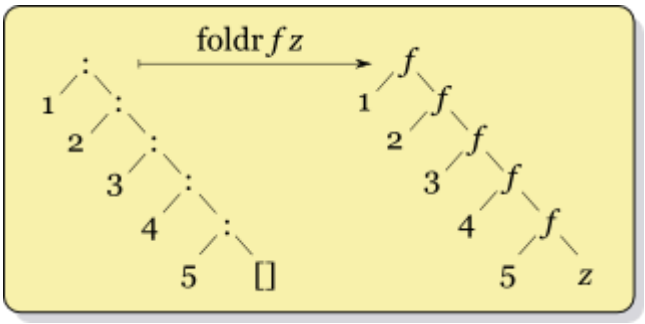
When we write recursive functions acting on `Link`s, we often find that they have the following form:

```
def func(link):
    if link is Link.empty:
        return <Base case>
    else:
        return <Expression involving func(link.rest)>
```

In the spirit of abstraction, we want to factor out this commonly seen pattern. It turns out that we can define an abstraction called `fold` that do this.

A linked list can be represented as a series of `Link` constructors, where `Link.rest` is either another linked list or the empty list.

We represent such a list in the diagram below:



In this diagram, the recursive list

```
Link(1, Link(2, Link(3, Link(4,Link(5)))))
```

is represented with `:` as the constructor and `[]` as the empty list.

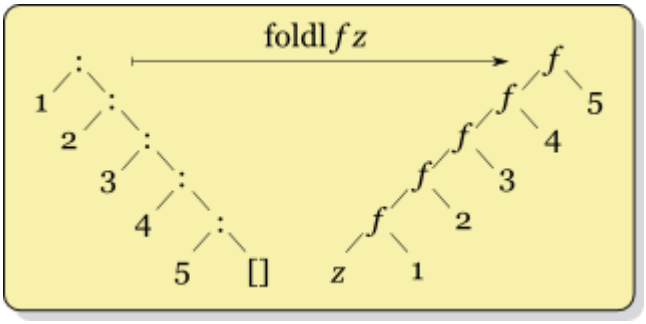
We define a function `foldr` that takes in a function `f` which takes two arguments, and a value `z`. `foldr` essentially replaces the `Link` constructor with `f`, and the empty list with `z`. It then evaluates the expression and returns the result. This is equivalent to:

```
f(1, f(2, f(3, f(4, f(5, z)))))
```

We call this operation a right fold.

Similarly we can define a left fold `foldl` that folds a list starting from the beginning, such that the function `f` will be applied this way:

```
f(f(f(f(f(z, 1), 2), 3), 4), 5)
```



Also notice that a left fold is equivalent to Python's `reduce` with a starting value.

Q3: Fold Right

Now write the right fold function.

```
def foldr(link, fn, z):
    """ Right fold
    >>> lst = Link(3, Link(2, Link(1)))
    >>> foldr(lst, sub, 0) # (3 - (2 - (1 - 0)))
    2
    >>> foldr(lst, add, 0) # (3 + (2 + (1 + 0)))
    6
    >>> foldr(lst, mul, 1) # (3 * (2 * (1 * 1)))
    6
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q foldr
```

Q4: Map With Fold

Write the `mapl` function, which takes in a Link `lst` and a function `fn`, and returns a new Link where every element is the function applied to every element of the original list. Use either `foldl` or `foldr`. Hint: it is much easier to write with one of them than the other!

```
def mapl(lst, fn):
    """ Maps FN on LST
    >>> lst = Link(3, Link(2, Link(1)))
    >>> mapl(lst, lambda x: x*x)
    Link(9, Link(4, Link(1)))
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q mapl
```

Scheme

Q5: Compose All

Implement `compose-all`, which takes a list of one-argument functions and returns a one-argument function that applies each function in that list in turn to its argument. For example, if `func` is the result of calling `compose-all` on a list of functions `(f g h)`, then `(func x)` should be equivalent to the result of calling `(h (g (f x)))`.

```
scm> (define (square x) (* x x))
square
scm> (define (add-one x) (+ x 1))
add-one
scm> (define (double x) (* x 2))
double
scm> (define composed (compose-all (list double square add-one)))
composed
scm> (composed 1)
5
scm> (composed 2)
17
```

```
(define (compose-all funcs)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q compose-all
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Objects

Q6: Checking account

We'd like to be able to cash checks, so let's add a `deposit_check` method to our `CheckingAccount` class. It will take a `Check` object as an argument, and check to see if the `payable_to` attribute matches the `CheckingAccount`'s holder. If so, it marks the `Check` as deposited, and adds the amount specified to the `CheckingAccount`'s total.

Write an appropriate `Check` class, and add the `deposit_check` method to the `CheckingAccount` class. Make sure not to copy and paste code! Use inheritance whenever possible.

See the doctests for examples of how this code should work.

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals.

    >>> check = Check("Steven", 42)  # 42 dollars, payable to Steven
    >>> steven_account = CheckingAccount("Steven")
    >>> eric_account = CheckingAccount("Eric")
    >>> eric_account.deposit_check(check)  # trying to steal steven's money
    The police have been notified.
    >>> eric_account.balance
    0
    >>> check.deposited
    False
    >>> steven_account.balance
    0
    >>> steven_account.deposit_check(check)
    42
    >>> check.deposited
    True
    >>> steven_account.deposit_check(check)  # can't cash check twice
    The police have been notified.
    """
    withdraw_fee = 1
    interest = 0.01

    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)

    "*** YOUR CODE HERE ***"

class Check(object):
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q CheckingAccount
```

Linked Lists

Q7: Fold Left

Write the left fold function by filling in the blanks.

```
def foldl(link, fn, z):
    """ Left fold
    >>> lst = Link(3, Link(2, Link(1)))
    >>> foldl(lst, sub, 0) # (((0 - 3) - 2) - 1)
    -6
    >>> foldl(lst, add, 0) # (((0 + 3) + 2) + 1)
    6
    >>> foldl(lst, mul, 1) # (((1 * 3) * 2) * 1)
    6
    """
    if link is Link.empty:
        return z
    "*** YOUR CODE HERE ***"
    return foldl(______, ______, ______)
```

Use Ok to test your code:

```
python3 ok -q foldl
```

Q8: Filter With Fold

Write the `filterl` function, using either `foldl` or `foldr`.

```
def filterl(lst, pred):
    """ Filters LST based on PRED
    >>> lst = Link(4, Link(3, Link(2, Link(1))))
    >>> filterl(lst, lambda x: x % 2 == 0)
    Link(4, Link(2))
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q filterl
```

Q9: Reverse With Fold

Notice that `mapl` and `filterl` are not recursive anymore! We used the implementation of `foldl` and `foldr` to implement the actual recursion: we only need to provide the recursive step and the base case to `fold`.

Use `foldl` to write `reverse`, which takes in a recursive list and reverses it. *Hint*: It only takes one line!

Extra for experience: Write a version of `reverse` that do not use the `Link` constructor. You do not have to use `foldl` or `foldr`.

```
def reverse(lst):
    """ Reverses LST with foldl
    >>> reverse(Link(3, Link(2, Link(1))))
    Link(1, Link(2, Link(3)))
    >>> reverse(Link(1))
    Link(1)
    >>> reversed = reverse(Link.empty)
    >>> reversed is Link.empty
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q reverse
```

Q10: Fold With Fold

Write `foldl` using `foldr`! You only need to fill in the `step` function.

```
identity = lambda x: x

def foldl2(link, fn, z):
    """ Write foldl using foldr
    >>> list = Link(3, Link(2, Link(1)))
    >>> foldl2(list, sub, 0) # (((0 - 3) - 2) - 1)
    -6
    >>> foldl2(list, add, 0) # (((0 + 3) + 2) + 1)
    6
    >>> foldl2(list, mul, 1) # (((1 * 3) * 2) * 1)
    6
    """
    def step(x, g):
        "*** YOUR CODE HERE ***"
    return foldr(link, step, identity)(z)
```

Use Ok to test your code:

```
python3 ok -q foldl2
```

Tree Recursion

Q11: Num Splits

Given a list of numbers `s` and a target difference `d`, how many different ways are there to split `s` into two subsets such that the sum of the first is within `d` of the sum of the second? The number of elements in each subset can differ.

You may assume that the elements in `s` are distinct and that `d` is always non-negative.

Note that the order of the elements within each subset does not matter, nor does the order of the subsets themselves. For example, given the list `[1, 2, 3]`, you should not count `[1, 2], [3]` and `[3], [1, 2]` as distinct splits.

> Hint: If the number you return is too large, you may be double-counting somewhere. If the result you return is off by some constant factor, it will likely be easiest to simply divide/subtract away that factor.

```
def num_splits(s, d):
    """Return the number of ways in which s can be partitioned into two
    sublists that have sums within d of each other.

    >>> num_splits([1, 5, 4], 0)  # splits to [1, 4] and [5]
    1
    >>> num_splits([6, 1, 3], 1)  # no split possible
    0
    >>> num_splits([-2, 1, 3], 2) # [-2, 3], [1] and [-2, 1, 3], []
    2
    >>> num_splits([1, 4, 6, 8, 2, 9, 5], 3)
    12
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q num_splits
```