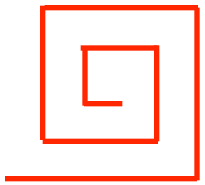
More Fruitful Recursion

CS111 Computer Programming

Department of Computer Science
Wellesley College

Fruitful Spiraling



Recall the definition for having a turtle draw a spiral and return to its original position and orientation:

```
def spiralBack(sideLen, angle, scaleFactor, minLength):
    """Draws a spiral based on the given parameters and
       brings the turtle back to its initial location and
       orientation."""
    if sideLen < minLength:
        pass
    else:
        fd(sideLen); lt(angle) # Put 2 stmts on 1 line with ;
        spiralBack(sideLen*scaleFactor, angle,
                   scaleFactor, minLength)
        rt(angle); bk(sideLen)
```

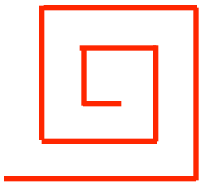
How can we modify this function to return

(1) the total length of lines in the spiral;

(2) the number of lines in the spiral;

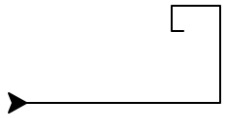
(3) both of the above numbers in a pair?

spiralLength

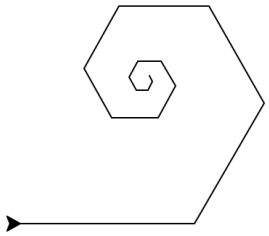


```
def spiralLength(sideLen, angle, scaleFactor, minLength)
    """Draws a spiral and returns the total length
       of the lines drawn."""
    if sideLen < minLength:
        return 0 # Length is 0 when no line drawn
    else:
        fd(sideLen); lt(angle)
        # Name the length returned by the recursive call
        subLen = spiralLength(sideLen*scaleFactor, angle,
                              scaleFactor, minLength)
        rt(angle); bk(sideLen)
        return sideLen + subLen # Length of all lines in spiral
```

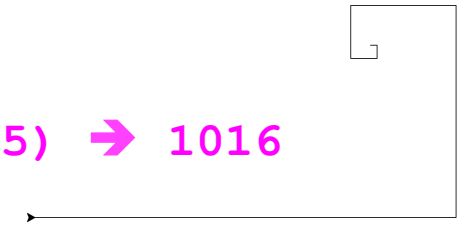
spiralLength(100, 90, 0.5, 5) ➔ 193.75



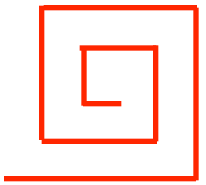
spiralLength(120, 60, 0.5, 5) ➔ 578.889376746700



spiralLength(512, 90, 0.5, 5) ➔ 1016

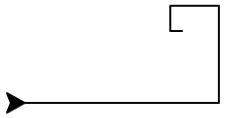


Exercise 1: spiralCount

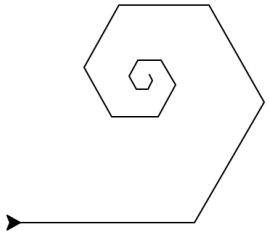



```
def spiralCount(sideLen, angle, scaleFactor, minLength)
    """Draws a spiral and returns the total number
       of lines drawn. """
    if sideLen < minLength:
        return ?? # What goes here?
    else:
        fd(sideLen); lt(angle)
        subCount = spiralCount(sideLen*scaleFactor, angle,
                               scaleFactor, minLength)
        rt(angle); bk(sideLen)
        return ?? # What goes here?
```

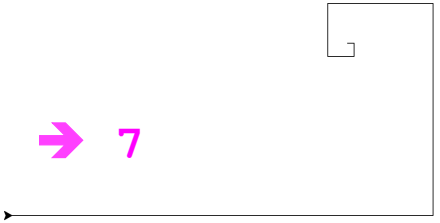
```
spiralCount(100, 90, 0.5, 5) ➔ 5
```



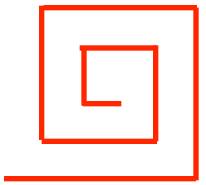
```
spiralTuple(120, 60, 0.5, 5) ➔ 15
```



```
spiralTuple(512, 90, 0.5, 5) ➔ 7
```

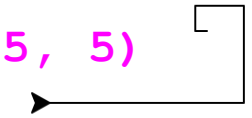


Exercise 2: `spiralTuple`

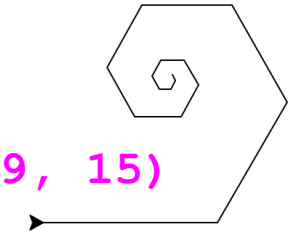



```
def spiralTuple(sideLen, angle, scaleFactor, minLength)
    """Draws a spiral and returns a pair of (1) the total length
       of the lines drawn and (2) the number of lines."""
    if sideLen < minLength:
        return ?? # What goes here?
    else:
        fd(sideLen); lt(angle)
        ??                      = spiralTuple(sideLen*scaleFactor, angle,
                                              scaleFactor, minLength)
        rt(angle); bk(sideLen)
        return ?? # What goes here?
```

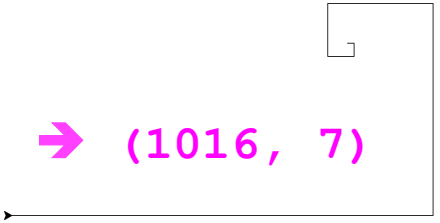
`spiralTuple(100, 90, 0.5, 5)` ➔ `(193.75, 5)`



`spiralTuple(120, 60, 0.5, 5)` ➔ `(578.8893767467009, 15)`



`spiralTuple(512, 90, 0.5, 5)` ➔ `(1016, 7)`



Exercise 3: Fruitful Trees

As with spirals, we can return counts of the drawings we make using fruitful recursion. Try this example below in the notebook and check the notebook solution for answers.

```
def branchCount(levels, trunkLen, angle, shrinkFactor):
    """Draw a 2-branch tree recursively and returns a
    count of the branches.
    levels: number of branches on any path
            from the root to a leaf
    trunkLen: length of the base trunk of the tree
    angle: angle from the trunk for each subtree
    shrinkFactor: shrinking factor for each subtree
    """
    # your code here
```

List of numbers from n down to 1

Define a function **`countDownList`** to return the list of numbers from n down to 1

```
countDownList(0) → [ ]
countDownList(5) → [5, 4, 3, 2, 1]
countDownList(8) → [8, 7, 6, 5, 4, 3, 2, 1]
```

Apply the wishful thinking strategy on n = 4:

- **`countDownList(4)`** should return **`[4, 3, 2, 1]`**
- By wishful thinking, assume **`countDownList(3)`** returns **`[3, 2, 1]`**
- How to combine **`4`** and **`[3, 2, 1]`** to yield **`[4, 3, 2, 1]`**?
  `[4] + [3, 2, 1]`
- Generalize: **`countDownList(n)`** = **`[n] + countDownList(n-1)`**

countDownList(n)

```
def countDownList(n):
    """Returns a list of numbers from n down to 1.
       For example, countDownList(5) returns
       [5,4,3,2,1].
    """
    if n <= 0:
        return []
    else:
        return [n] + countDownList(n-1)
```

To remember

When the glue operation in a recursive function involves lists, **the identity value** is the empty list.

Define countDownListPrintResults(n)

```
def countDownListPrintResults(n):
    """Returns a list of numbers from n down to 1
       and also prints each recursive result along
       the way."""
    if n <= 0:
        # add a print statement here
        result = []
    else:
        result = [n] + countDownListPrintResults(n-1)
        # add a print statement here
        return result
```

Exercise 4: Define `countUpList(n)`


```
def countUpList(n):
    """Returns a list of numbers from 1 up to n.
       For example, countUpList(5) returns
       [1,2,3,4,5]."""
    if n <= 0:
        return ?? # What goes here?
    else:
        return ?? # What goes here?
```

sublists

For a given list L (possibly containing duplicates), let's use the term *sublist of L* to refer to any list that keeps some elements of L and omits others in their same relative order. E.g., the sublists of `[5, 3, 8, 3]` are:

```
[5, 3, 8, 3] # Keep all elements
[3, 8, 3] # Omit 5
[5, 8, 3] # Omit 1st 3
[5, 3, 3] # Omit 8
[5, 3, 8] # Omit 2nd 3
[8, 3] # Omit 5 and 1st 3
[3, 3] # Omit 5 and 8
[3, 8] # Omit 5 and 2nd 3
[5, 3] # Omit 8 and 1st 3
[5, 3] # Omit 8 and 2nd 3
       # (note duplication)
[5, 8] # Omit both 3s
[5] # Keep only 5
[3] # Keep only 1st 3
[8] # Keep only 8
[3] # Keep only 2nd 3
    # (Note duplication)
[] # Omit all elements
```

`sublistSum` function

Given a list of numbers (possibly containing duplicates) and a target number, **`sublistSum`** returns a list of all sublists whose sum is the target number. For example:

```
sublistSum([2, 3, 5, 5, 11, 17], 23)
  → [[2, 5, 5, 11]] # Only sublist that sums to 23
# The fact that [2, 3, 5, 5, 11, 17] is sorted is irrelevant;
# it just makes it easy to keep track of the numbers.

sublistSum([2, 3, 5, 5, 11, 17], 30)
  → [[2, 11, 17], [3, 5, 5, 17]] # Two sublists sum to 30

sublistSum([2, 3, 5, 5, 11, 17], 24)
  → [[2, 5, 17], [2, 5, 17], [3, 5, 5, 11]]
  # One sublist uses the 1st 5, the other uses the 2nd 5

sublistSum([2, 3, 5, 5, 11, 17], 34)
  → [] # No sublists sum to 34
```

But how to think about **implementing** this function?

Recall Big Idea #3:

Big #3: Problem Solving Strategies

Example: Divide/Solve/Combine

Divide

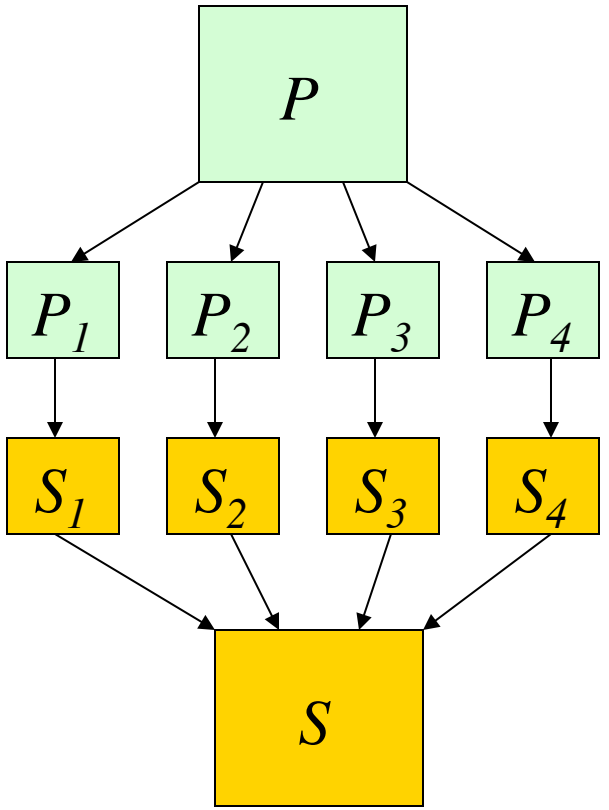
problem P into subproblems.

Solve

each of the subproblems.

Combine

the solutions to the subproblems into a solution S for P.

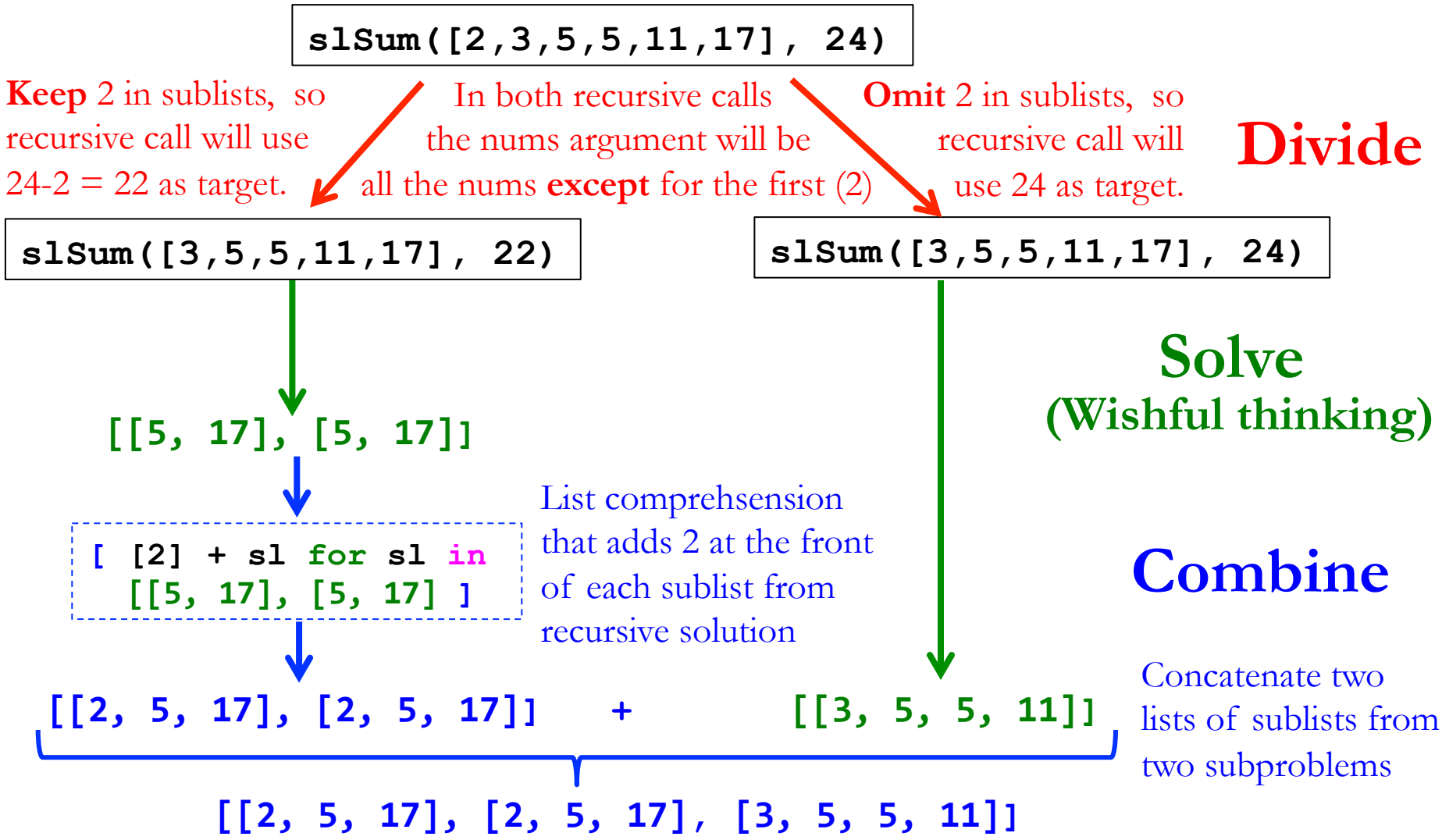


Other Strategies/Skills

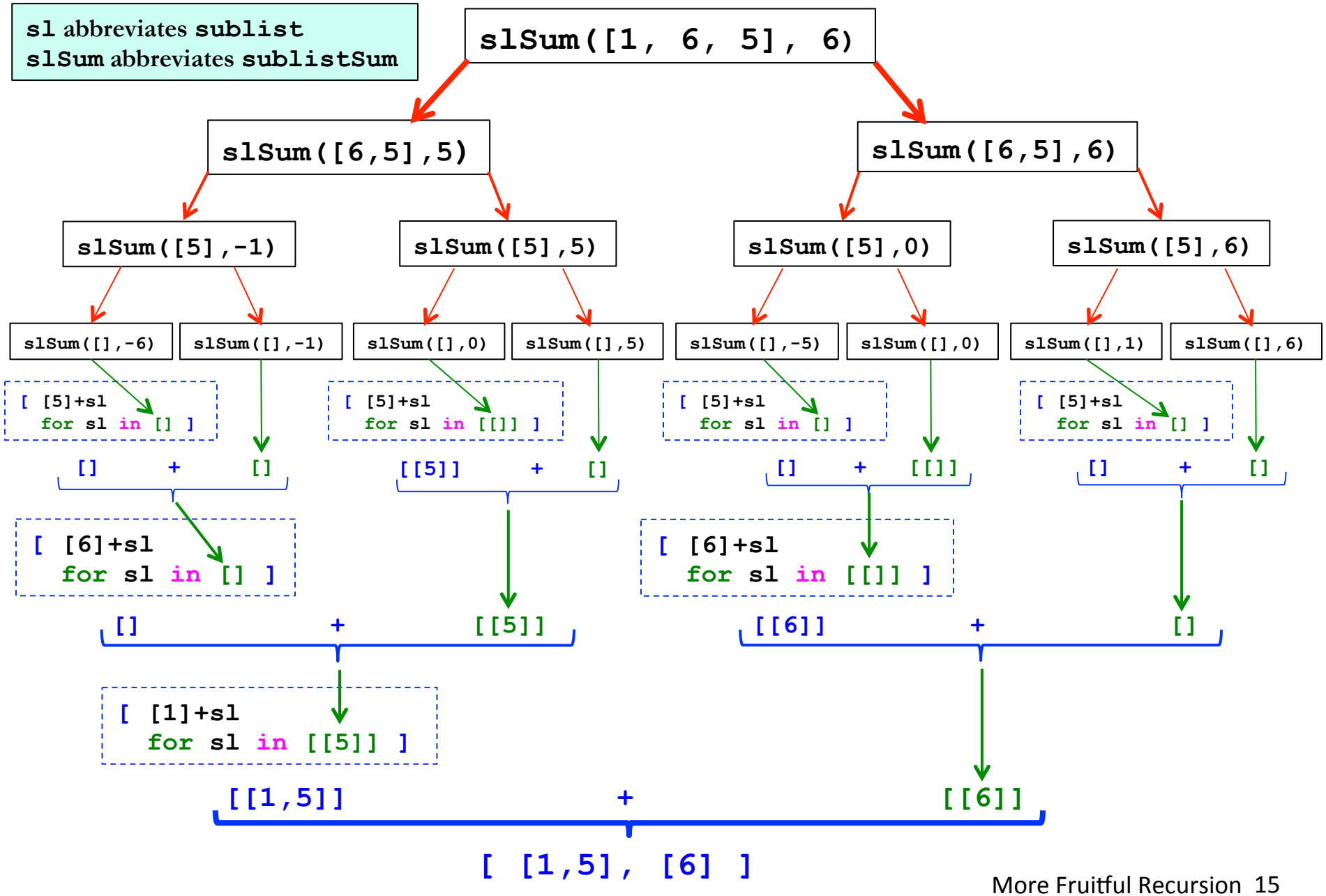
- Incremental/iterative development
- Testing & Debugging

`sublistSum` divide/solve/combine strategy: keep or omit 1st element

`sl` abbreviates `sublist`
`slSum` abbreviates `sublistSum`



sublistSum strategy: recursion tree example



`sublistSum` definition

```
def sublistSum(nums, target):
    if nums == []: # base case
        # Subtlety: there are *two* sub base cases:
        if target == 0:
            return [[]] # sum([]) == 0, so include [] in result list
        else:
            return [] # sum([]) cannot be nonzero,
                      # so don't include [] in result list
    else:
        fst = nums[0] # first number in list
        rst = nums[1:] # all but first numbers in list
        keepingFirst = [([fst] + sumList) # all sublists keeping fst
                        for sumList in sublistSum(rst, target-fst)
                                       # recursive call excludes fst
                       ]
        omittingFirst = sublistSum(rst, target) # all sublists omitting fst
        return keepingFirst + omittingFirst
```

list comprehension

Testing sublistSum

```
>>> for tgt in range(20,36):
    testSublistSum([2, 3, 5, 5, 11, 17], tgt)
```

```
sublistSum([2, 3, 5, 5, 11, 17], 20) => [[3, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 21) => [[2, 3, 5, 11], [2, 3, 5, 11], [5, 5, 11]]
sublistSum([2, 3, 5, 5, 11, 17], 22) => [[2, 3, 17], [5, 17], [5, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 23) => [[2, 5, 5, 11]]
sublistSum([2, 3, 5, 5, 11, 17], 24) => [[2, 5, 17], [2, 5, 17], [3, 5, 5, 11]]
sublistSum([2, 3, 5, 5, 11, 17], 25) => [[3, 5, 17], [3, 5, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 26) => [[2, 3, 5, 5, 11]]
sublistSum([2, 3, 5, 5, 11, 17], 27) => [[2, 3, 5, 17], [2, 3, 5, 17], [5, 5, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 28) => [[11, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 29) => [[2, 5, 5, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 30) => [[2, 11, 17], [3, 5, 5, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 31) => [[3, 11, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 32) => [[2, 3, 5, 5, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 33) => [[2, 3, 11, 17], [5, 11, 17], [5, 11, 17]]
sublistSum([2, 3, 5, 5, 11, 17], 34) => []
sublistSum([2, 3, 5, 5, 11, 17], 35) => [[2, 5, 11, 17], [2, 5, 11, 17]]
```

Alternative approach to `sublistSum`

Suppose we had a `sublists` function that returns all sublists of a list. (The order of sublists isn't specified; they can be in any order.) E.g.:

```
sublists([5, 3, 8, 3])
  → [ [5, 3, 8, 3], [5, 3, 8], [5, 3, 3], [5, 3],
      [5, 8, 3], [5, 8], [5, 3], [5],
      [3, 8, 3], [3, 8], [3, 3], [3],
      [8, 3], [8], [3], [] ]
```

1st half of results keeps first element at the front of every sublist in 2nd half

2nd half of results are sublists omitting first element

Then we could define `sublistSum` as:

```
def sublistSum(nums, target)
    """Alternative implementation of sublistSum
       using sublists"""
    return [ns for ns in sublists(nums)
            if sum(ns) == target]
```

Exercise 5: define `sublists`


```
def sublists(xs):
    '''Given a list of n values (which might contain duplicates),
       return a list of all possible 2^n sublists, where a sublist
       is the result of independently choosing to keep or not to
       keep particular value occurrences without changing their
       relative order. The order of sublists is not specified.
    '''
    if xs == []:
        return ?? # What goes here?
    else:
        fst = xs[0] # first element in list
        rst = xs[1:] # all but first element in list
        omittingFirst = ?? What goes here?
        keepingFirst = ?? What goes here?


        return ?? # What goes here?
```

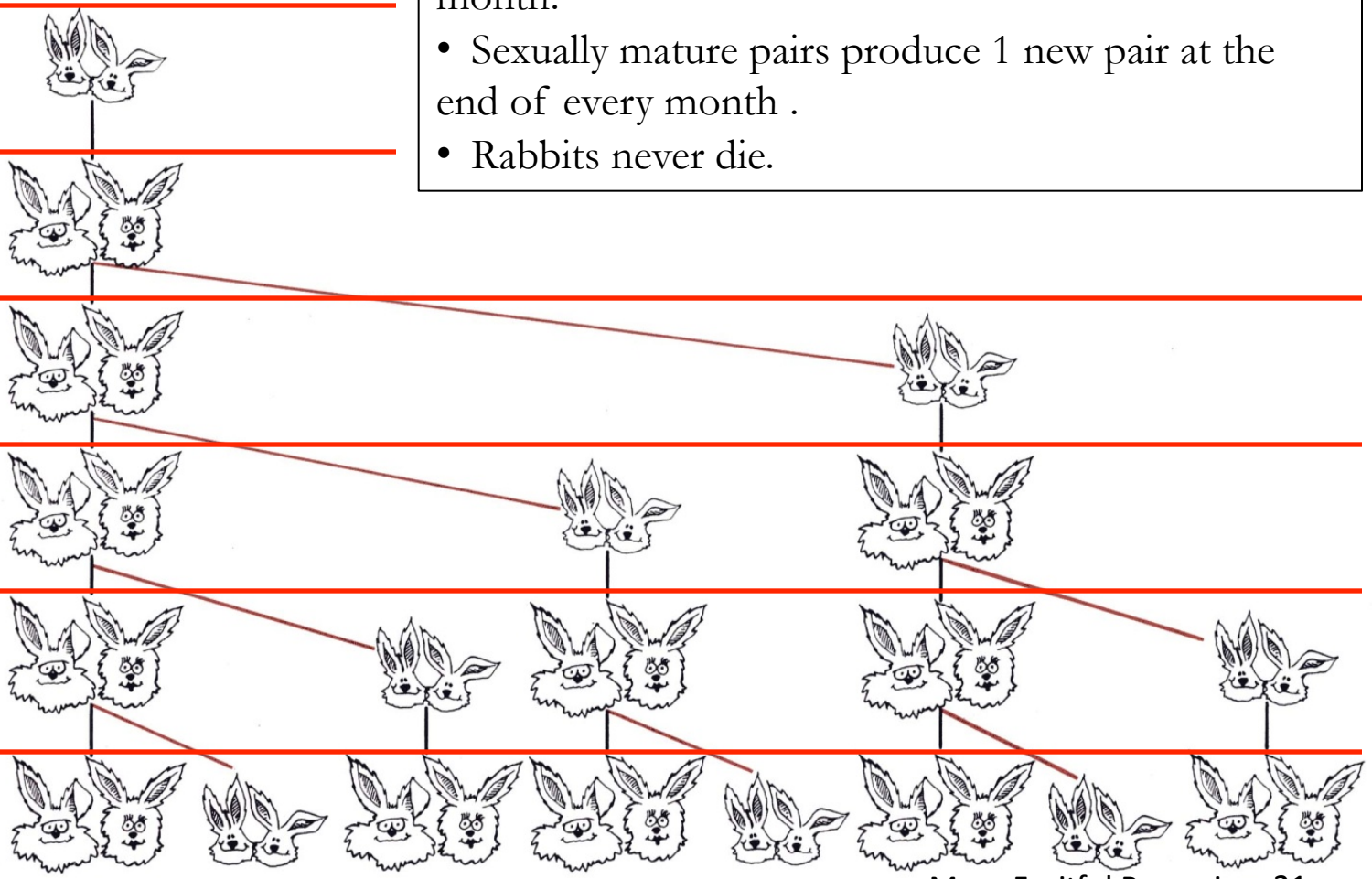
Extra: Fibonacci numbers

Leonardo Pisano Fibonacci counts Rabbits

Month	# Pairs
0	0
1	1
2	1
3	2
4	3
5	5
6	8

Assume:

- Start with one pair of newborn rabbits in month 1.
- Newborn rabbits become sexually mature after 1 month.
- Sexually mature pairs produce 1 new pair at the end of every month .
- Rabbits never die.

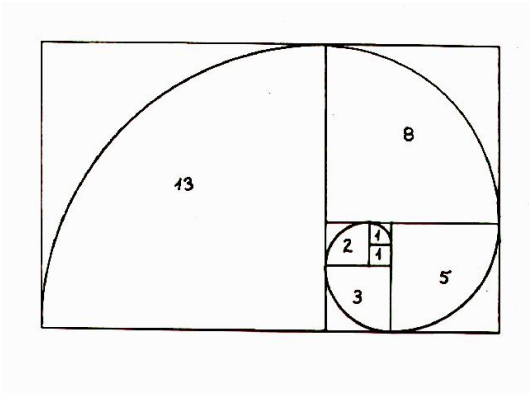


Exercise 6: Fibonacci Numbers `fib(n)`

The **n**th Fibonacci number fib(n) is the number of pairs of rabbits alive in the **n**th month.

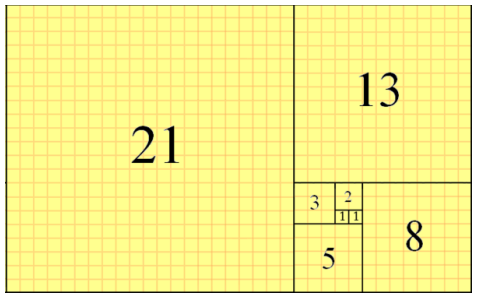
Formula:

fib(0) = 0 ; no pairs initially
fib(1) = 1 ; 1 pair introduced the first month
fib(n) = fib(n-1) ; pairs never die, so live to next month
+ fib(n-2) ; all sexually mature pairs produce
; a pair each month

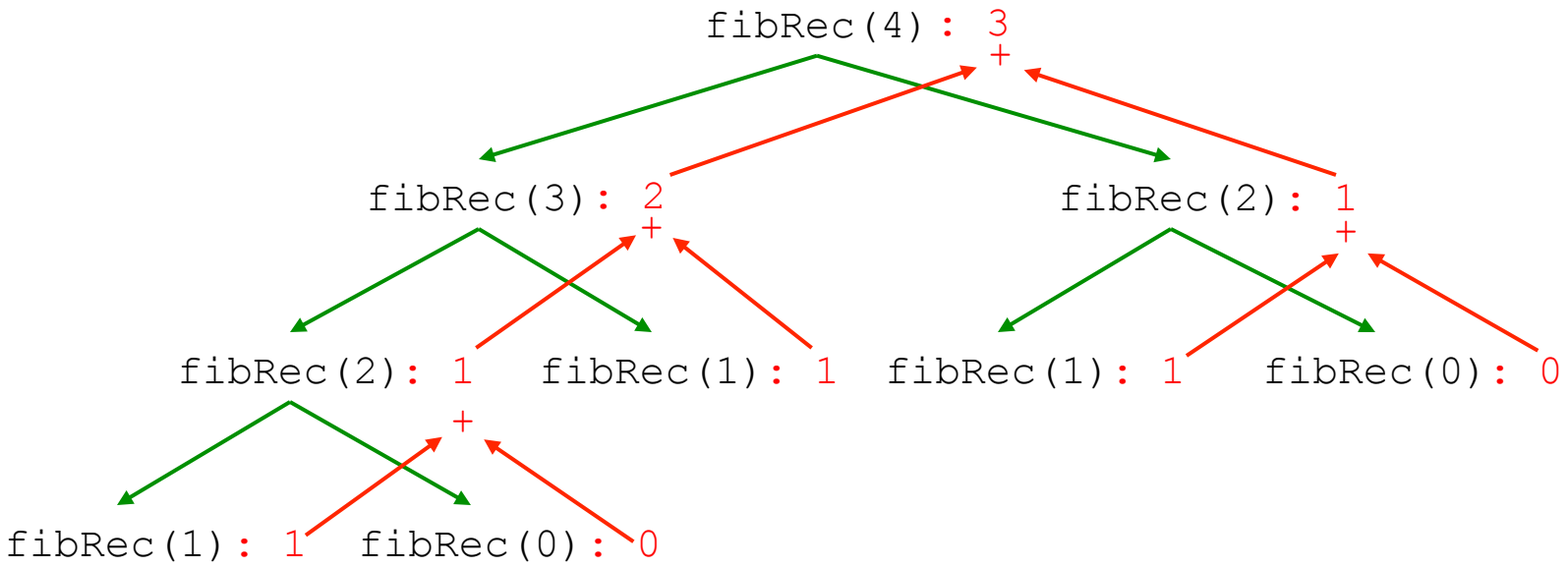


Now write the program:

```
def fibRec(n):
    '''Returns the nth Fibonacci number.'''
    if n <= 1:
        return n
    else:
        return fibRec(n-1) + fibRec(n-2)
```



Fibonacci: Efficiency



How long would it take to calculate `fibRec(100)`?

Is there a better way to calculate Fibonacci numbers?

Iteration leads to a more efficient `fib(n)`

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

Iteration table for calculating the 8th Fibonacci number:

i	fibi	fibi_next
0	0	1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13
7	13	21
8	21	34

Exercise 7: `fibLoop(n)`

Use iteration to calculate Fibonacci numbers more efficiently:

i	fibi	fibi_next
0	0	1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13
7	13	21
8	21	34

```
def fibLoop(n):
    '''Returns the nth Fibonacci number.'''
    fibi = 0
    fibi_next = 1
    for i in range(1, n+1):
        # flesh out this loop body


    return ?? # What goes here?
```