Introduction to recursion

Recursion

- A **recursive function** is a function that solves a problem by making one or more calls to itself
- A recursive function consists of
  - **One or more base cases:** these are the portions of the problem for which an immediate solution is known
  - **One or more recursive calls**: to avoid infinite recursion these subproblems must be in some way smaller than the original problem

Recursion example

```
>>> countdown(3)
3
2
1
0
-1
-2
...
-976
-977
-978
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    countdown(3)
  File "/Users/me/ch10.py", line 3, in countdown
    countdown(n-1)
...
 File "/Users/me/ch10.py", line 3, in countdown
    countdown(n-1)
  File "/Users/me/ch10.py", line 2, in countdown
    print(n)
RuntimeError: maximum recursion depth exceeded
while calling a Python object
>>>
```

A recursive function is one that calls itself

What happens when we run `countdown(3)`?

```
def countdown(n):
        print(n)
        countdown(n-1)
```

ch10.py

The function calls itself repeatedly until the system resources are exhausted

- i.e., the limit on the size of the program stack is exceeded

In order for the function to terminate normally, there must be a stopping condition

Recursion

```
def countdown(n):
    'counts down to 0'
    if n <= 0:
        print('Blastoff!!!')
    else:
        print(n)
        countdown(n-1)
```

ch10.py

Suppose that we really want this behavior

```
>>> countdown(3)
3
2
1
Blastoff!!!
>>> countdown(1)
1
Blastoff!!!
>>> countdown(0)
Blastoff!!!
>>> countdown(-1)
Blastoff!!!
```

If $n \leq 0$

`'Blastoff!!!'` is printed

In order for the function to terminate normally, there must be a stopping condition

Recursion

A recursive function should consist of

1. One or more base cases which provide the stopping condition of the recursion
2. One or more recursive calls on input arguments that are "closer" to the base case

Base case

Recursive step

```
def countdown(n):
    'counts down to 0'
    if n <= 0:
        print('Blastoff!!!')
    else:
        print(n)
        countdown(n-1)
```

ch10.py

This will ensure that the recursive calls eventually get to the base case that will stop the execution

Program stack

Program stack

The execution of recursive function calls is supported by the program stack

- just like regular function calls

```
def countdown(n):
    'counts down to 0'
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

n = 1
n = 2
n = 3

Program stack

```
>>> countdown(3)
3
2
1
Blastoff!
```

```
n = 3
print(3)
countdown(2)
```

countdown(3)

```
n = 2
print(2)
countdown(1)
```

countdown(2)

```
n = 1
print(1)
countdown(0)
```

countdown(1)

```
n = 0
print('Blastoff!')
```

countdown(0)

Program stack

The execution of recursive function calls is supported by the program stack

- just like regular function calls

line = 7
n = 1
line = 7
n = 2
line = 7
n = 3

Program stack

```
1. def countdown2(n):
2.     'prints digits of n vertically'
3.     if n <= 0:
4.         print('Blastoff!')
5.     else:
6.         countdown2(n - 1)
7.         print(n)
```

```
>>> countdown2(3)
```

```
n = 3
countdown2(2)

print(3)
```
countdown2(3)

```
n = 2
countdown2(1)

print(2)
```
countdown2(2)

```
n = 1
countdown2(0)

print(1)
```
countdown2(1)

```
n = 0
print('Blastoff!')
```
countdown2(0)

Recursive thinking

Recursive thinking

```
def countdown(n):
    'counts down to 0'
    if n <= 0:
        print('Blastoff!!!')
    else:
        print(n)
        countdown(n-1)
```

ch10.py

A recursive function should consist of

1. One or more base cases which provide the stopping condition of the recursion
2. One or more recursive calls on input arguments that are "closer" to the base case

Problem with input n

To count down from n to 0 …

… we print n and then count down from n-1 to 0

Subproblem with input n-1

So, to develop a recursive solution to a problem, we need to:

1. Define one or more bases cases for which the problem is solved directly
2. Express the solution of the problem in terms of solutions to subproblems of the problem (i.e., easier instances of the problem that are closer to the bases cases)

Recursive thinking

We use **recursive thinking** to develop function `vertical()` that takes a non-negative integer and prints its digits vertically

```
>>> vertical(3124)
3
1
2
4
```

To print the digits of n vertically …

Original problem with input n

… print all but the last digit of n

and then print the last digit

Subproblem with input having one less digit than n

The last digit of n: `n%10`

The integer obtained by removing the last digit of n: `n//10`

First define the base case

- **The case when the problem is "easy"**
- When input n is a single-digit number

Next, we construct the recursive step

- When input n has two or more digits

```
def vertical(n):
    'prints digits of n vertically'
    if n < 10:
        print(n)
    else:
        vertical(n//10)
        print(n%10)
```

ch10.py

So, to develop a recursive solution to a problem, we need to:

1. Define one or more bases cases for which the problem is solved directly
2. Express the solution of the problem in terms of solutions to subproblems of the problem (i.e., easier instances of the problem that are closer to the bases cases)

Program stack

The execution of recursive function calls is supported by the program stack

- just like regular function calls

line = 7
n = 31
line = 7
n = 312
line = 7
n = 3124

Program stack

```
1. def vertical(n):
2.     'prints digits of n vertically'
3.     if n < 10:
4.         print(n)
5.     else:
6.         vertical(n//10)
7.         print(n%10)
```

```
>>> vertical(3124)
```

```
n = 3124
vertical(n//10)

print(n%10)
```

vertical(3124)

```
n = 312
vertical(n//10)

print(n%10)
```

vertical(312)

```
n = 31
vertical(n//10)

print(n%10)
```

vertical(31)

```
n = 3
print(n)
```

vertical(3)

Examples

Exercise

Implement recursive method `reverse()` that takes a nonnegative integer as input and prints its digits vertically, starting with the low-order digit.

```
>>> reverse(3124)
4
2
1
3
```

```
def reverse(n):
    'prints digits of n vertically starting with low-order digit'
    if n <10:            # base case: one digit number
        print(n)
    else:                # n has at least 2 digits
       print(n%10)       # prints last digit of n
       reverse(n//10)    # recursively print in reverse all but the last digit
```

```python
def reverse(n):
    'prints digits of n vertically starting with low-order digit'
    if n <10:              # base case: one digit number
        print(n)
    else:                  # n has at least 2 digits
        print(n%10)        # prints last digit of n
        reverse(n//10)     # recursively print in reverse all but the last digit
```

Exercise

Implement a function **sumDigits(num)** that takes a nonnegative integer as input and returns the sum of the digits

```
>>> sumDigits(3621)
12
>>> sumDigits(382015)
19
>>> sumDigits(2149087)
31
```

Subproblems:

Last digit: num%10

Remaining digits: num//10

Base:

num < 10

Exercise

Implement a function **vowels(s)** that takes a string as input and returns the number of vowels in the string.

```
>>> vowels('abed')
2
>>> vowels('Good bye')
3
>>> vowels('My name is')
3
```

Subproblems:

Last character: s[n-1:n]
Remaining character: s[:n-1]

Base:
n = len(s)
Empty string: n == 0

```python
def vowels(s):
    n = len(s)
    v = 'aeiou'
    counter = 0
    if n == 0:
        return 0
    elif s[n-1:n] in v: #last character in string
        #print(s[n-1:n], end = ' ' )
        counter = 1
    return vowels(s[:n-1])+ counter
```

A recursive number sequence pattern

So far, the problems we have considered could have easily been solved without recursion, i.e., using iteration

We consider a problem that are best solved recursively.

Consider function `pattern()` that takes a nonnegative integer as input and prints a corresponding number pattern

```
>>> pattern(0)
0
>>> pattern(1)
0 1 0
>>> pattern(2)
0 1 0 2 0 1 0
>>> pattern(3)
0 1 0 2 0 1 0 3 0 1 0 2 0 1 0
>>>
```

Base case: `n == 0`

Recursive step:

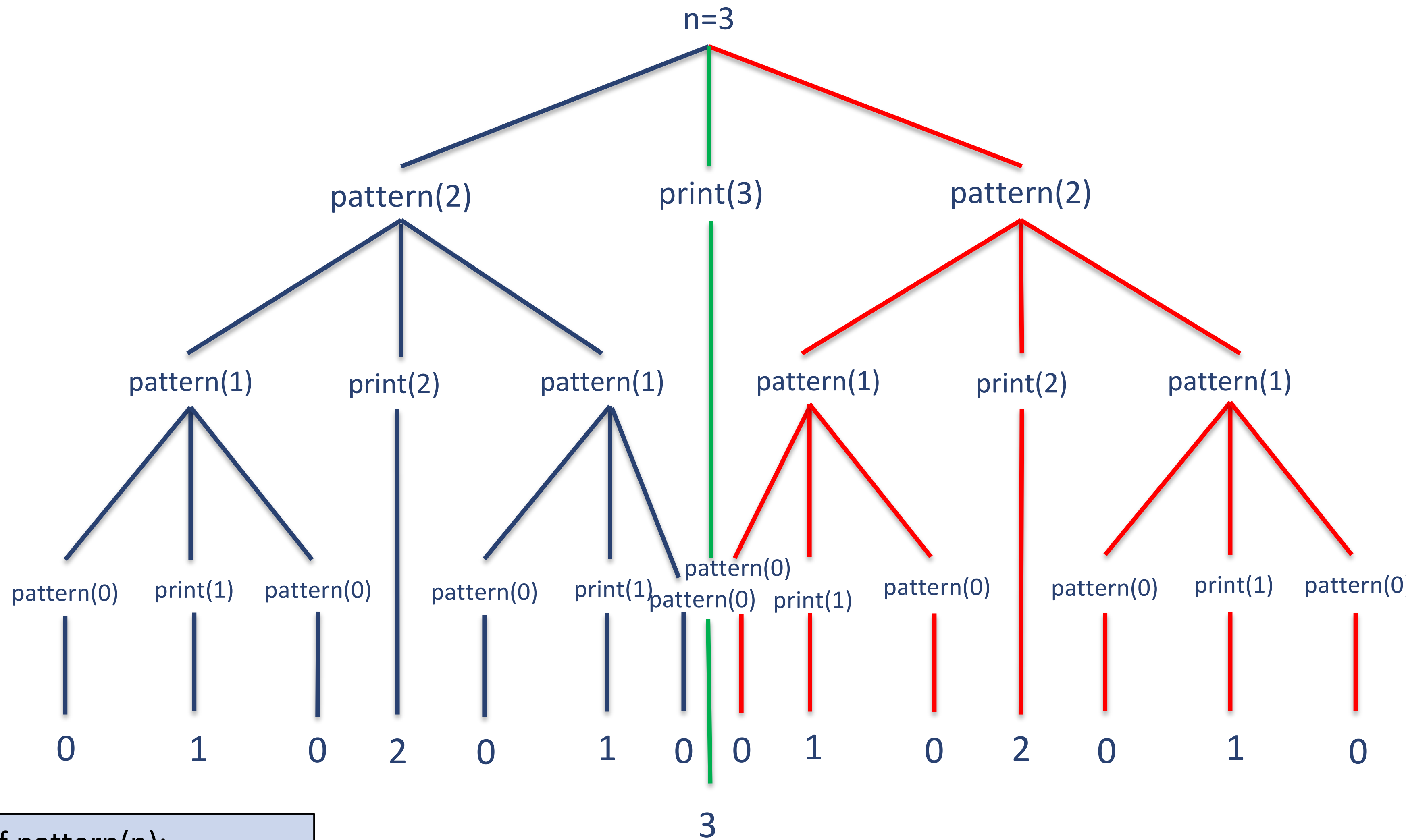
To implement `pattern(3)` …

… we need to execute `pattern(2)`,
then print `3`,
and then execute `pattern(2)` again

… we need to execute `pattern(n-1)`,
then print `n`,
and then execute `pattern(n-1)`

```
def pattern(n):
    'prints the n-th pattern'
    if n == 0:                # base case
        print(0, end=' ')
    else:                     # recursive step: n > 0

        # to do
```




```
def pattern(n):
    if n == 0:
        print(0, end =' ')
    else:
        pattern(n-1)
        print(n, end=' ')
        pattern(n-1)
```

RUN TIME ANALYSIS

Algorithm analysis

There are usually several approaches (i.e., algorithms) to solve a problem. Which one is the right one? the best?

Typically, the one that is fastest (for all or at least most real world inputs)

How can we tell which algorithm is the fastest?

- theoretical analysis
- experimental analysis

```
import time
def timing(func, n):
    'runs func on input n'

    start = time.time()        # take start time
    func(n)                # run func on n
    end = time.time()          # take end time

    return end - start         # return execution time
```

```
>>> timing(fib, 30)
1.1920928955078125e-05
>>> timing(rfib, 30)
0.7442440986633301
```

0.00000119... seconds

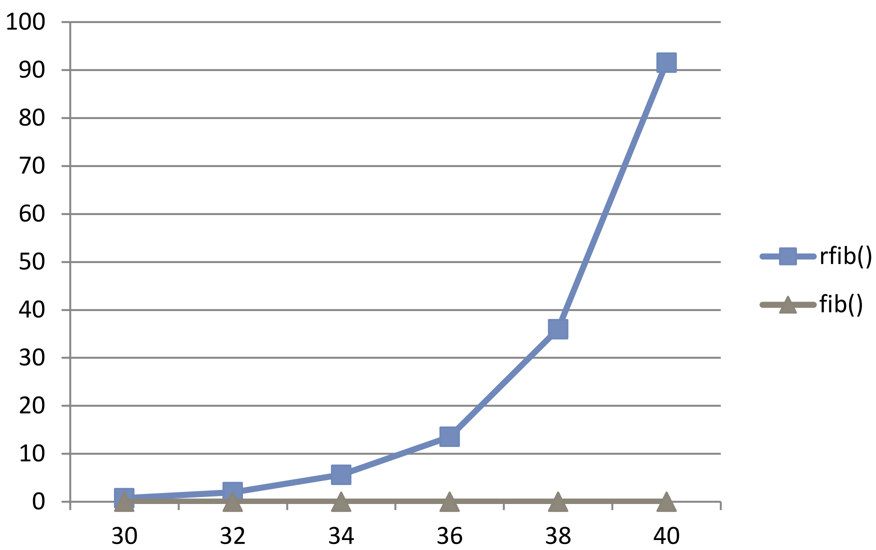
0.744... seconds

using a range of input values

- e.g., 30, 32, 34, ..., 40

Algorithm analysis

```
def timingAnalysis(func, start, stop, inc, runs):
    '''prints average run-times of function func on inputs of
       size start, start+inc, start+2*inc, ..., up to stop'''
    for n in range(start, stop, inc):  # for every input size n
        acc=0.0                        # initialize accumulator
        for i in range(runs):       # repeat runs times:
            acc += timing(func, n)      # run func on input size n
                                        # and accumulate run-times
        # print average run times for input size n
        formatStr = 'Run-time of {}({}) is {:.7f} seconds.'
        print(formatStr.format(func.__name__,n,acc/runs))
```




```
>>> timingAnalysis(rfib, 30, 41, 2, 5)
Run-time of rfib(30) is 0.7410099 seconds.
Run-time of rfib(32) is 1.9761698 seconds.
Run-time of rfib(34) is 5.6219893 seconds.
Run-time of rfib(36) is 13.5359141 seconds.
Run-time of rfib(38) is 35.9763714 seconds.
Run-time of rfib(40) is 91.5498876 seconds.
>>> timingAnalysis(fib, 30, 41, 2, 5)
Run-time of fib(30) is 0.0000062 seconds.
Run-time of fib(32) is 0.0000072 seconds.
Run-time of fib(34) is 0.0000074 seconds.
Run-time of fib(36) is 0.0000074 seconds.
Run-time of fib(38) is 0.0000082 seconds.
Run-time of fib(40) is 0.0000084 seconds.
```

Searching a list

Consider list method `index()` and list operator `in`

```
>>> lst = random.sample(range(1,100), 17)
>>> lst
[9, 55, 96, 90, 3, 85, 97, 4, 69, 95, 39, 75, 18, 2, 40, 71, 77]
>>> 45 in lst
False
>>> 75 in lst
True
>>> lst.index(75)
11
>>>
```

How do they work? How fast are they? Why should we care?

- the list elements are visited from left to right and compared to the target; this search algorithm is called sequential search
- In general, the running time of sequential search is a linear function of the list size
- If the list is huge, the running time of sequential search may take time; there are faster algorithms if the list is sorted

Searching a sorted list

How can search be done faster if the list is sorted?

Suppose we search for 75

3 comparisons instead of 12

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	9	18	39	40	55	69	71	75	77	85	90	95	96	97

Suppose we search for 45

5 comparisons instead of 17

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	9	18	39	40	55	69	71	75	77	85	90	95	96	97

Algorithm idea: Compare the target with the middle element of a list

- Either we get a hit
- Or the search is reduced to a sublist that is less than half the size of the list

Let's use recursion to describe this algorithm

Searching a list

Suppose we search for target 75

								mid		mid		mid				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	9	18	39	40	55	69	71	75	77	85	90	95	96	97

```
def search(lst, target, i, j):
    '''attempts to find target in sorted sublist lst[i:j];
       index of target is returned if found, -1 otherwise'''
```

The recursive calls will be on sublists of the original list

Algorithm:

1. Let `mid` be the middle index of list `lst`
2. Compare `target` with `lst[mid]`
   - If `target > lst[mid]` continue search of `target` in sublist `lst[mid+1:]`
   - If `target < lst[mid]` continue search of `target` in sublist `lst[?:?]`
   - If `target == lst[mid]` return `mid`

Binary search

```
def search(lst, target, i, j):
    '''attempts to find target in sorted sublist lst[i:j];
       index of target is returned if found, -1 otherwise'''
    if i == j:                           # base case: empty list
        return -1                        # target cannot be in list

    mid = (i+j)//2                       # index of median of l[i:j]

    if lst[mid] == target:               # target is the median
        return mid
    if target < lst[mid]:                # search left of median
        return search(lst, target, i, mid)
    else:                                # search right of median
        return search(lst, target, mid+1, j)
```

Algorithm:

1. Let `mid` be the middle index of list `lst`
2. Compare `target` with `lst[mid]`
   - If `target > lst[mid]` continue search of `target` in sublist `lst[mid+1:]`
   - If `target < lst[mid]` continue search of `target` in sublist `lst[?:?]`
   - If `target == lst[mid]` return `mid`

Comparing sequential and binary search

Let’s compare the running times of both algorithms on a random array

```
def binary(lst):
    'chooses item in list lst at random and runs search() on it'
    target = random.choice(lst)
    return search(lst, target, 0, len(lst))



def linear(lst):
    'choose item in list lst at random and runs index() on it'
    target = random.choice(lst)
    return lst.index(target)
```

But we need to abstract our experiment framework first

Comparing sequential and binary search

```
import time
def timing(func, n):
    'runs func on input returned by buildInput'
    funcInput = buildInput(n)  # obtain input for func

    start = time.time()        # take start time
    func(funcInput)            # run func on funcInput
    end = time.time()          # take end time

    return end - start         # return execution time


# buildInput for comparing Linear and Binary search
def buildInput(n):
    'returns a random sample of n numbers in range [0, 2n)'
    lst = random.sample(range(2*n), n)
    lst.sort()
    return lst
```

function `timing()` used for the factorial problem

generalized function `timing()` for arbitrary problems

But we need to abstract our experiment framework first

Comparing sequential and binary search

```
>>> timingAnalysis(linear, 200000, 1000000, 200000, 20)
Run time of linear(200000) is 0.0046095
Run time of linear(400000) is 0.0091411
Run time of linear(600000) is 0.0145864
Run time of linear(800000) is 0.0184283
>>> timingAnalysis(binary, 200000, 1000000, 200000, 20)
Run time of binary(200000) is 0.0000681
Run time of binary(400000) is 0.0000762
Run time of binary(600000) is 0.0000943
Run time of binary(800000) is 0.0000933
```

Exercise

Consider 3 functions that return `True` if every item in the input list is unique and `False` otherwise

Compare the running times of the 3 functions on 10 lists of size 2000, 4000, 6000, and 8000 obtained from the below function `buildInput()`

```
def dup1(lst):
    for item in lst:
        if lst.count(item) > 1:
            return True
    return False

def dup2(lst):
    lst.sort()
    for index in range(1, len(lst)):
        if lst[index] == lst[index-1]:
            return True
    return False

def dup3(lst):
    s = set()
    for item in lst:
        if item in s:
            return False
        else:
            s.add(item)
    return True
```

```
import random
def buildInput(n):
    'returns a list of n random integers in range [0, n**2)'
    res = [] for i in range(n):
        res.append(random.choice(range(n**2)))
    return res
```