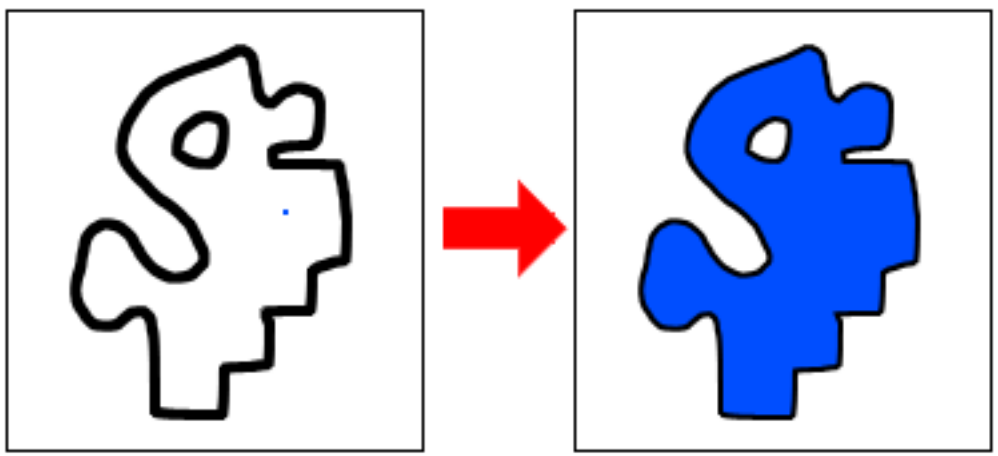
15-112
Fundamentals of Programming

Week 10 - Lecture 2:
Wrapping up recursion. Functions redux.



March 24, 2016

Wrapping up recursion

Back to sorting

Selection Sort

Find min, put it in the first index.

Repeat on the remaining elements.

```
def selectionsort(L):
    if (len(L) < 2):
        return L
    else:
        i = L.index(min(L))
        return [L[i]] + selectionsort(L[:i] + L[i+1:])
```

Back to sorting

Merge Sort

```
def merge(a, b):
    # We have already seen this.

def mergesort(L):
    if (len(L) <= 1):
        return L
    leftHalf = L[0 : len(L)//2]
    rightHalf = L[len(L)//2 : len(L)]
    return merge(mergesort(leftHalf), mergesort(rightHalf))
```

This strategy has a name: Divide and Conquer

Can we do merge recursively also?

Back to sorting

Merge Sort

```
def recursiveMerge(a, b):
    # beautiful, but not as efficient as iterative merge
    if ((len(a) == 0) or (len(b) == 0)):
        return a+b
    else:
        if (a[0] < b[0]):
            return [a[0]] + recursiveMerge(a[1:], b)
        else:
            return [b[0]] + recursiveMerge(a, b[1:])
```

Back to sorting

Insertion Sort

Insertion Sort = Merge Sort where

$$\text{leftHalf} = L[0]$$

$$\text{rightHalf} = L[1 : \text{len}(L)]$$

```
def insertionsort(L):
    if (len(L) <= 1):
        return L
    else:
        first = L[0]
        rest = insertionsort(L[1:])
        lo = [x for x in rest if x < first]
        hi = [x for x in rest if x >= first]
        return lo + [first] + hi
```

"leftHalf" already sorted

recursively sort "rightHalf"

"merge" the "halves"

Back to sorting

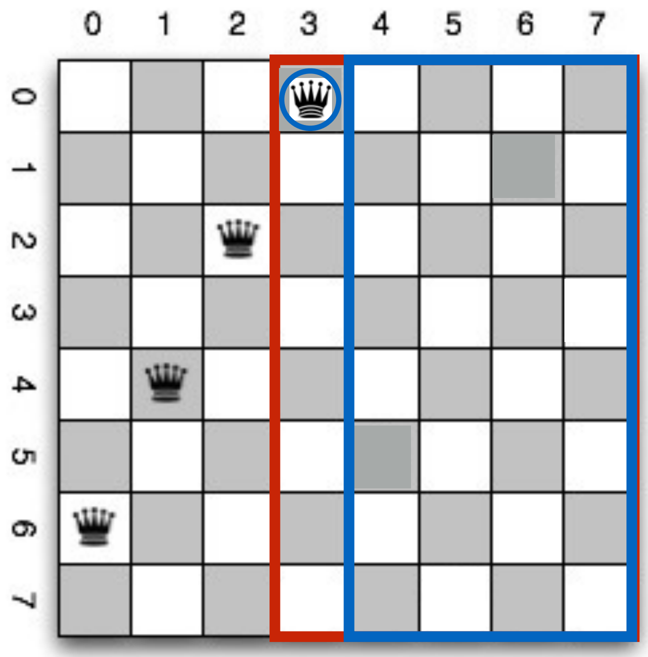
Quick Sort

Quick Sort = Insertion Sort where the recursive sorting is done at the end.

```
def quicksort(L):
    if (len(L) <= 1):
        return L
    else:
        first = L[0]  # this element is called a pivot
        rest = L[1:]
        lo = [x for x in rest if x < first]
        hi = [x for x in rest if x >= first]
        return quicksort(lo) + [first] + quicksort(hi)
```

Quite efficient in practice.

nQueens Problem



[0,?,?,?,?]

```
def solve(n, m, constraints):
    if(m == 0):
        return []
    for row in range(n):
        if (isLegal(row, constraints)):
            newConstraints = constraints + [row]
            result = solve(n, m-1, newConstraints)
            if (result != False):
                return [row] + result
    return False
```

n = 8

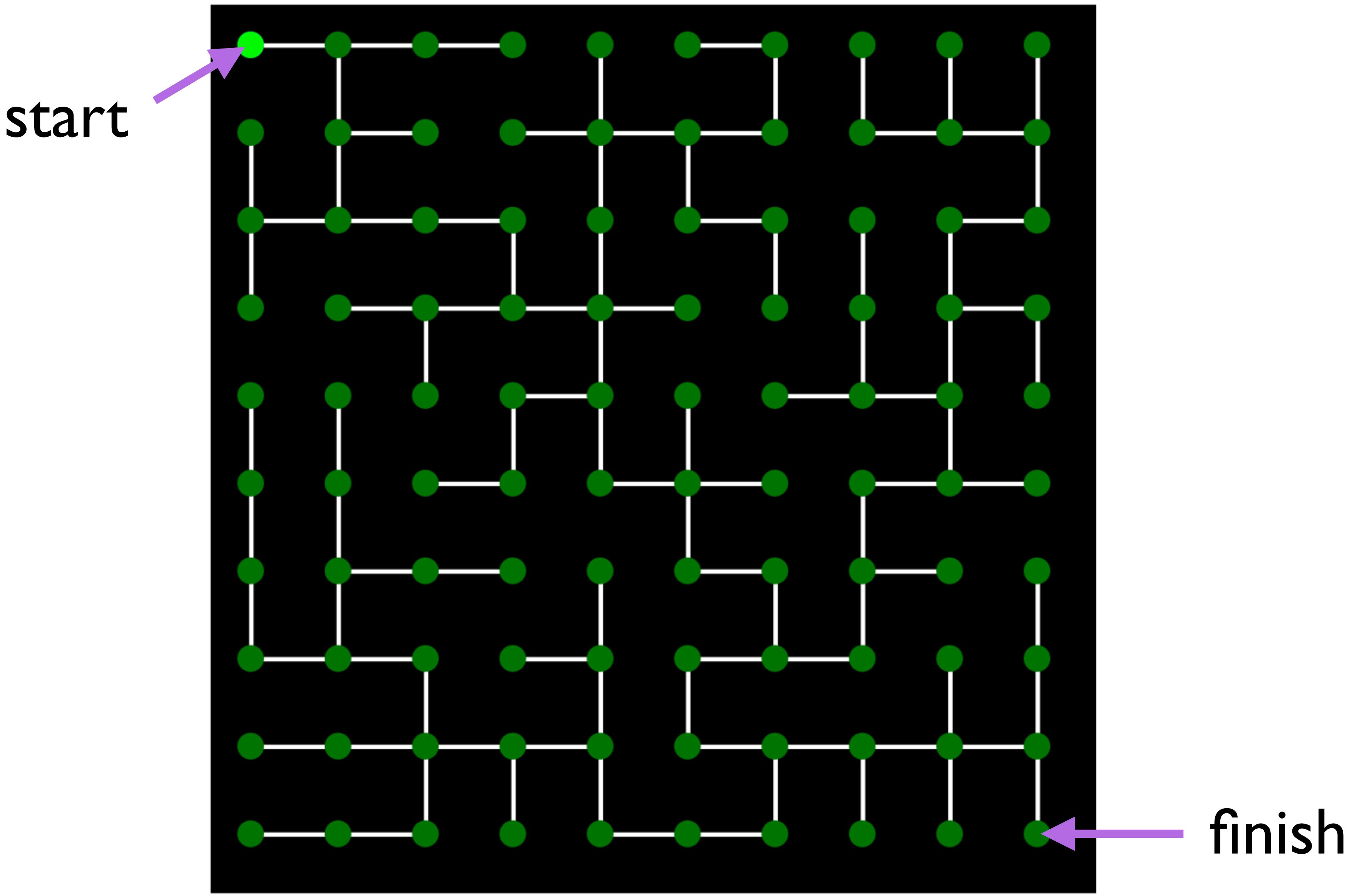
m = 5

constraints = [6,4,2]

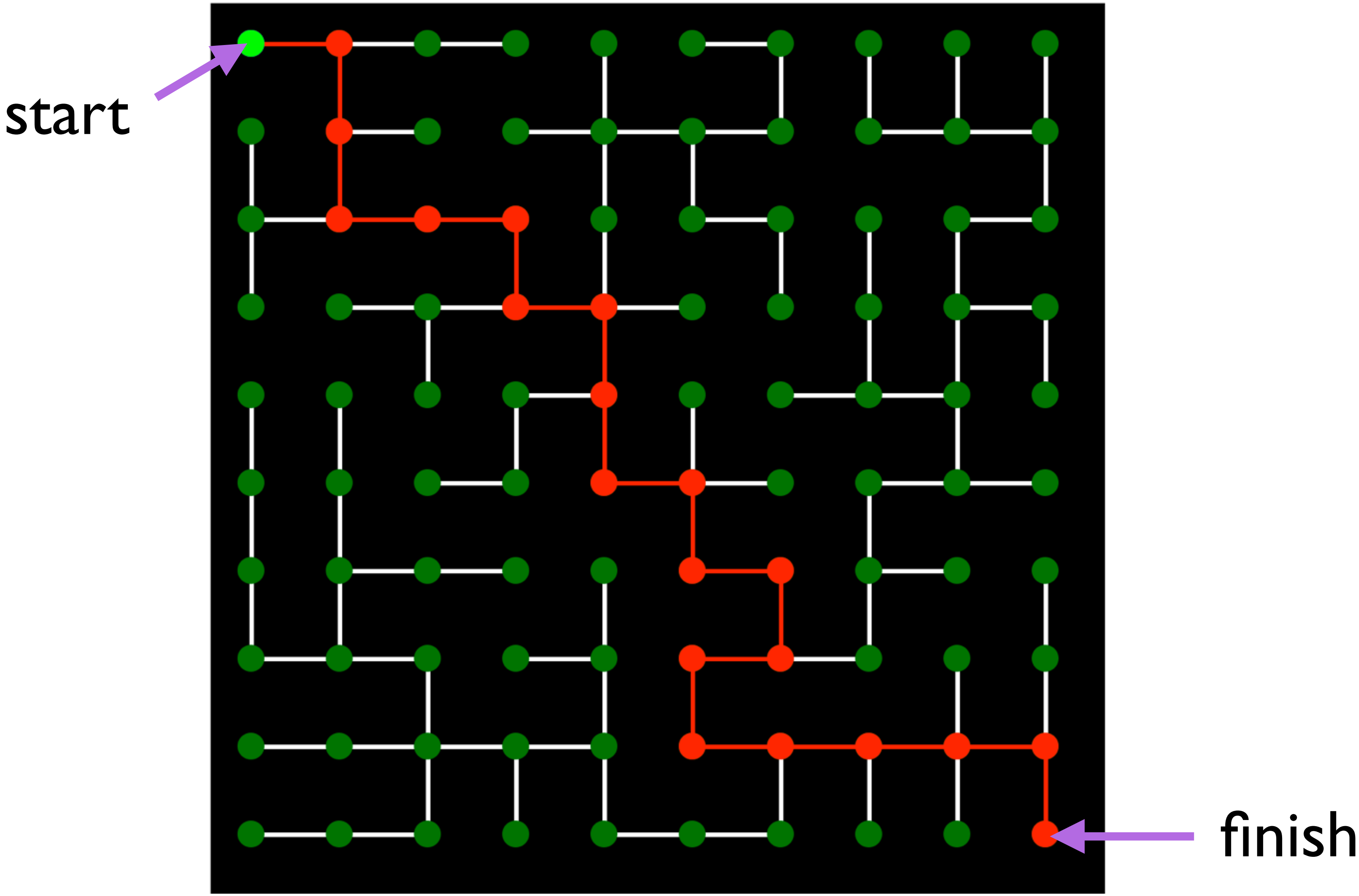
Call solve(8, 8, [ ])

to get solution for n = 8

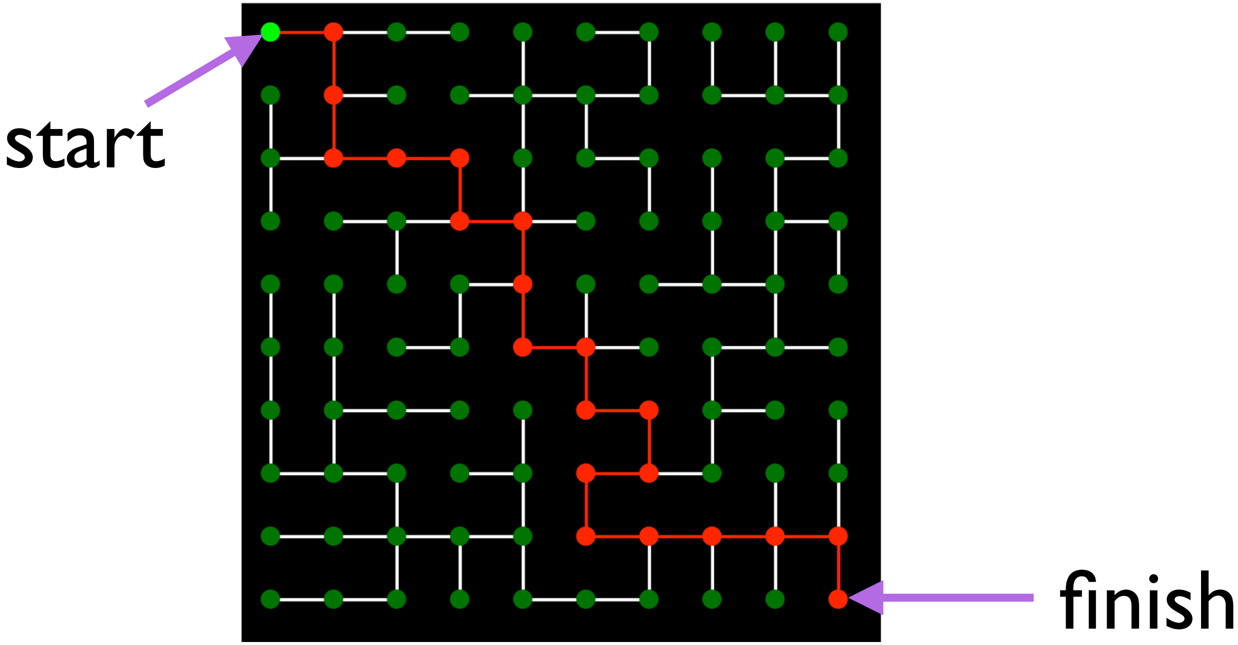
Solving a maze puzzle



Solving a maze puzzle



Solving a maze puzzle



def isSolvable(maze, (rowStart, colStart), (rowEnd, colEnd)):

—> True or False

Main Idea:

if isSolvable(maze, (rowStart, colStart), (rowEnd, colEnd)),

then for some neighbor (rowN, colN) of (rowStart, colStart),

isSolvable(maze, (rowN, colN), (rowEnd, colEnd))

Solving a maze puzzle

```
def isSolvable(maze, (rowStart, colStart), (rowEnd, colEnd)):
  if ((rowStart, colStart) == (rowEnd, colEnd)):
    return True
                   U      D      R      L
  for dir in [(-1,0), (1,0), (0,1), (0,-1)]:
    newCell = (rowStart, colStart) + dir
    if (isLegal(maze, (rowStart, colStart), dir) and
        isSolvable(maze, (newCell[0], newCell[1]), (rowEnd, colEnd))):
      return True
  return False
```

Where is the bug?

Solving a maze puzzle

```
visited = set()
def isSolvable(maze, (rowStart, colStart), (rowEnd, colEnd)):
    if ((rowStart, colStart) in visited):
        return False
    visited.add((rowStart, colStart))
    if ((rowStart, colStart) == (rowEnd, colEnd)):
        return True
    for dir in [(-1,0), (1,0), (0,1), (0,-1)]:
        newCell = (rowStart, colStart) + dir
        if (isLegal(maze, (rowStart, colStart), dir) and
            isSolvable(maze, (newCell[0], newCell[1]), (rowEnd, colEnd))):
            return True
    return False
```

Solving a maze puzzle

```
visited = set()
def isSolvable(maze, (rowStart, colStart), (rowEnd, colEnd)):
  if ((rowStart, colStart) in visited):
    return False
  visited.add((rowStart, colStart))
  if ((rowStart, colStart) == (rowEnd, colEnd)):
    return True
  for dir in [(-1,0), (1,0), (0,1), (0,-1)]:
    newCell = (rowStart, colStart) + dir
    if (isLegal(maze, (rowStart, colStart), dir) and
       isSolvable(maze, (newCell[0], newCell[1]), (rowEnd, colEnd))):
      return True
  visited.remove((rowStart, colStart))
  return False
```

if you want visited to be the cells in the solution.

Solving a maze puzzle

```
visited = set()        solution = set()
def isSolvable(maze, (rowStart, colStart), (rowEnd, colEnd)):
    if ((rowStart, colStart) in visited):
        return False
    visited.add((rowStart, colStart))
    solution.add((rowStart, colStart))
    if ((rowStart, colStart) == (rowEnd, colEnd)):
        return True
    for dir in [(-1,0), (1,0), (0,1), (0,-1)]:
        newCell = (rowStart, colStart) + dir
        if (isLegal(maze, (rowStart, colStart), dir) and
             isSolvable(maze, (newCell[0], newCell[1]), (rowEnd, colEnd))):
            return True
    solution.remove((rowStart, colStart))
    return False
```

more efficient

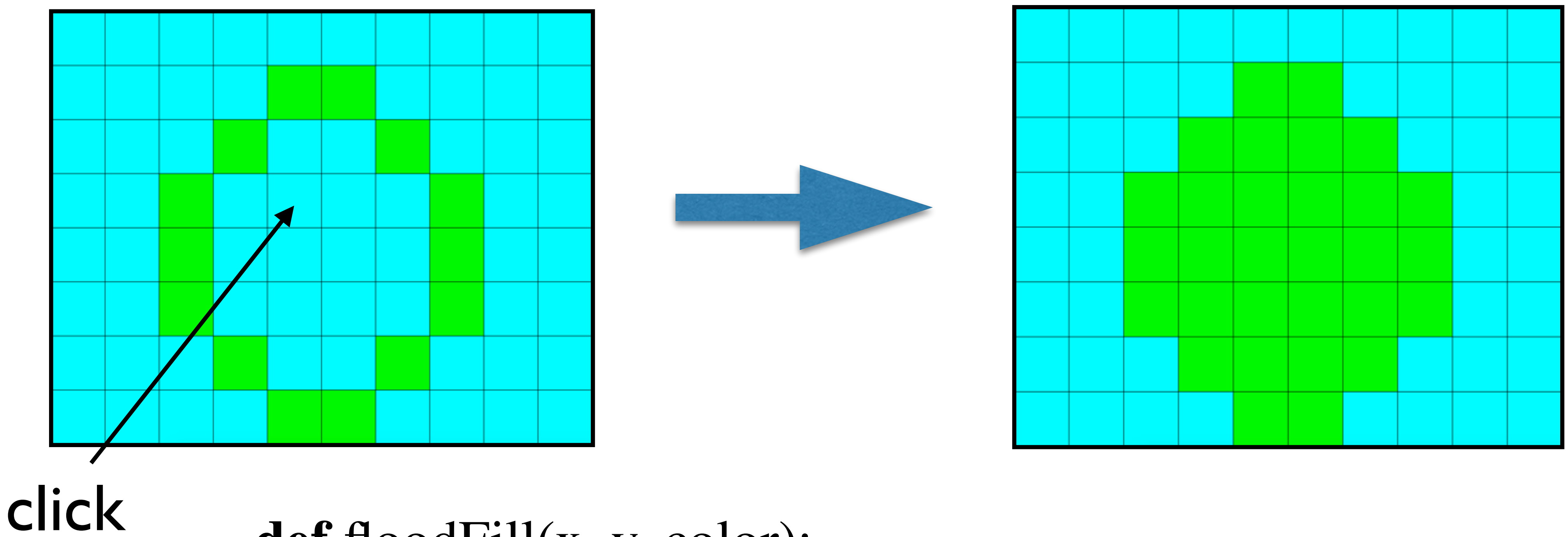
Solving a maze puzzle

```
def solve(maze, (rowStart, colStart), (rowEnd, colEnd), path=[]):
    # path corresponds to the cells already in your solution.
    # these cells are unusable.
    if ((rowStart, colStart) == (rowEnd, colEnd)):
        return [(rowStart, colStart)]

    for dir in [(-1,0), (1,0), (0,1), (0,-1)]:
        newCell = (rowStart, colStart) + dir
        if (isLegal(maze, (rowStart, colStart), dir) and (newCell not in path)):
            newPath = path + [(rowStart, colStart)]
            result = solve(maze, (newCell[0], newCell[1]),
                                  (rowEnd, colEnd), newPath)
            if (result != False): return [newCell] + result
    return False
```

no globals
or nonlocals

Flood fill

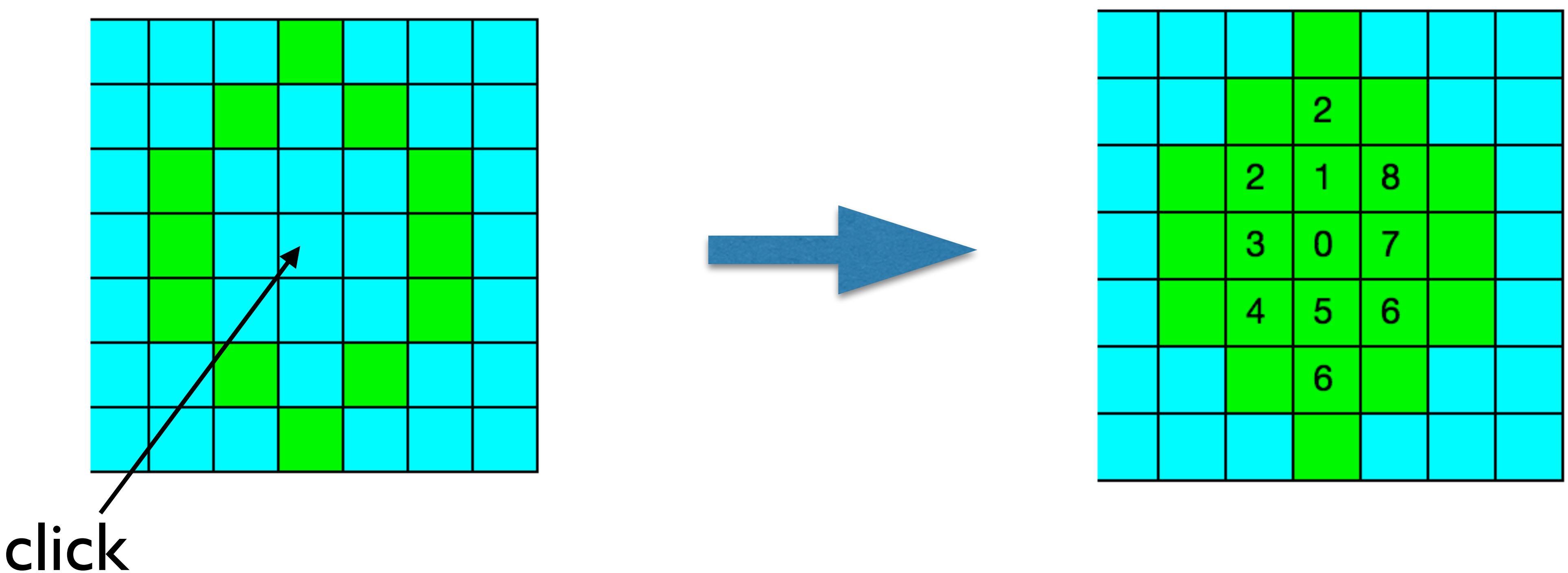



```
def floodFill(x, y, color):
    if ((not inImage(x,y)) or (getColor(img, x, y) == color)):
        return
    img.put(color, to=(x, y))
    floodFill(x-1, y, color)      U
    floodFill(x+1, y, color)      D
    floodFill(x, y-1, color)      L
    floodFill(x, y+1, color)      R
```

Flood fill

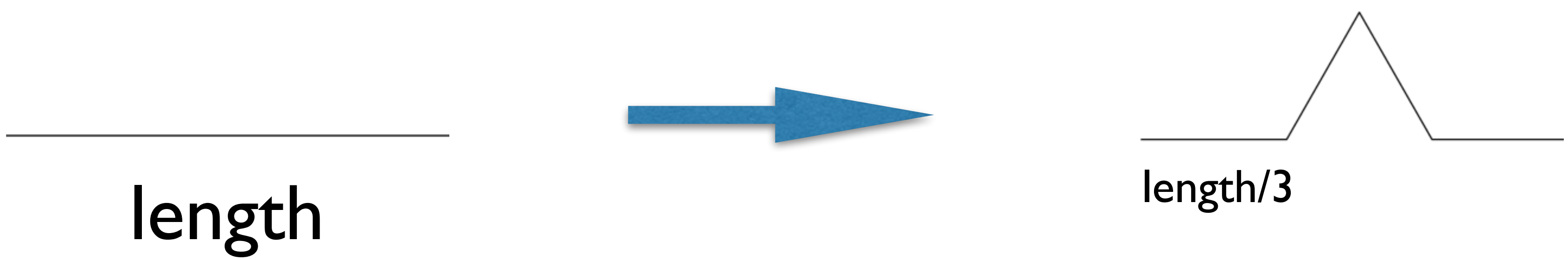
U floodFill(row-1, col, color, depth+1)
D floodFill(row+1, col, color, depth+1)
L floodFill(row, col-1, color, depth+1)
R floodFill(row, col+1, color, depth+1)

If we were to print the depth in the cell:



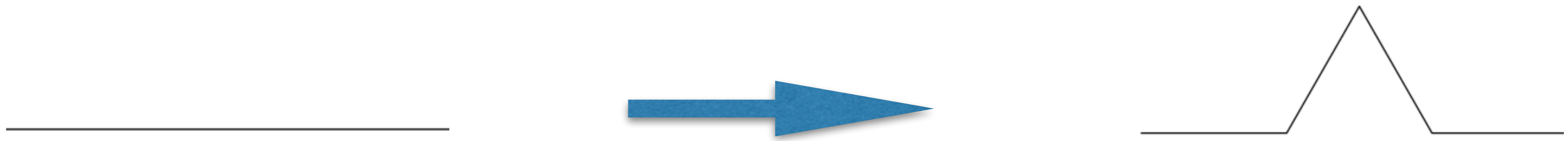
Fractals

A change rule:



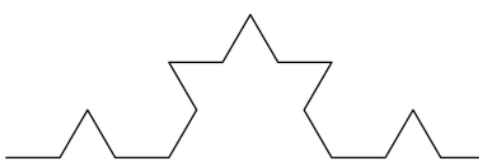
Fractals: kochSnowflake

n = 1

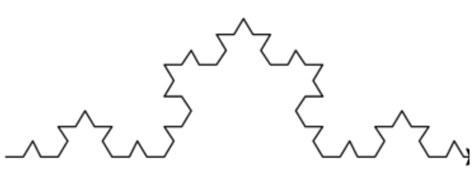


n = 2

n = 3

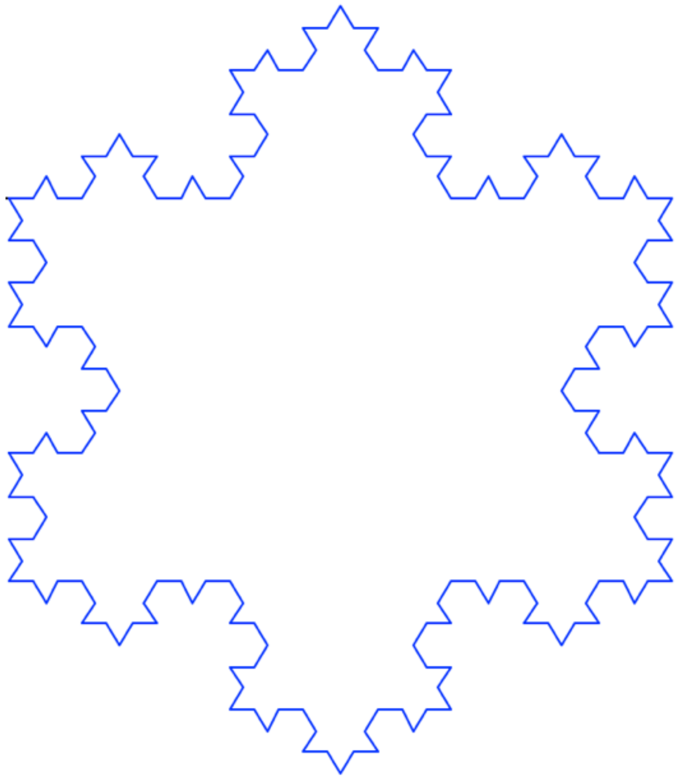


n = 4



```
def kochSide(length, n):
    if (n == 1):
        turtle.forward(length)
    else:
        kochSide(length/3, n-1)
        turtle.left(60)
        kochSide(length/3, n-1)
        turtle.right(120)
        kochSide(length/3, n-1)
        turtle.left(60)
        kochSide(length/3, n-1)
```

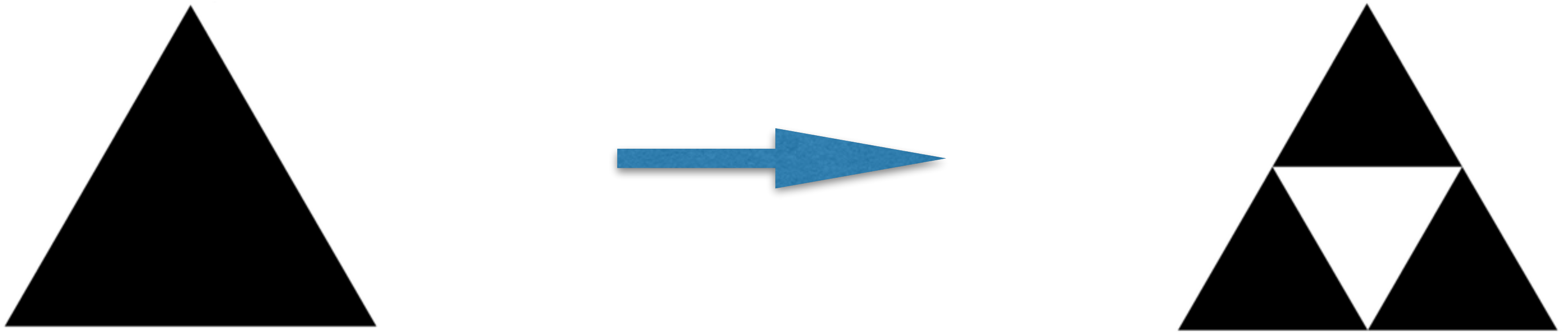
Fractals: kochSnowflake



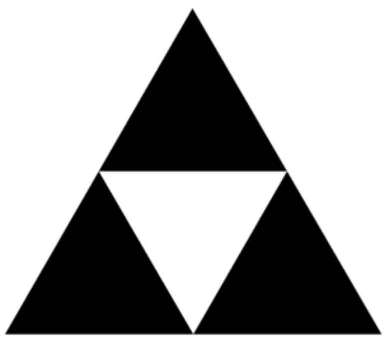
```
def kochSnowflake(length, n):
    # just call kochSide 3 times
    for step in range(3):
        kochSide(length, n)
        turtle.right(120)
```

Fractals: Sierpinski Triangle

level 0



level 1



```
def drawST(x, y, size, level):
    # (x, y) is the bottom-left corner of the triangle
    if (level == 0):
        canvas.create_polygon((x, y),
                              (x+size, y),
                              (x+size/2, y-size*(3**0.5)/2),
                              fill="black" )
    else:
        drawST(x, y, size/2, level-1)
        drawST(x+size/2, y, size/2, level-1)
        drawST(x+size/4, y-size*(3**0.5)/4, size/2, level-1)
```

level 2

Functions redux

Functions are first class objects

Functions are first-class citizens:

Can use them like you use any other object.
(in Python, pretty much everything is an object)

- Can pass functions as arguments to other functions
- Functions can be return values for other functions
- Functions can be assigned to other variables, or can be stored in data structures (e.g. lists)

Functions are first class objects

```
# Assume selectionSort, bubbleSort, mereSort are defined

def testSort(sortFn, n):
    a = [random.randint(0, 2**31) for i in range(n)]
    start = time.time()
    sortFn(a)
    end = time.time()
    return (end - start)


sortFunctions = [selectionSort, bubbleSort, mergeSort]
n = 2**12
for sortFn in sortFunctions:
    testSort(sortFn, n)
```

Default argument values

def myPrint(x, times=1):
    for i **in** range(times):
        print (x)

myPrint(“Hello”)   Hello

myPrint(“Hi”, 5)   Hi
Hi
Hi
Hi
Hi

Default argument values

Need to be careful with default argument values.

```
def f(x, L=[ ]):
    L.append(x)
    print(L)

f(1)    # expect: [1]    reality: [1]
f(2)    # expect: [2]    reality: [1, 2]
```

Default argument is evaluated <u>**once**</u> when function is defined.

Default argument values

Need to be careful with default argument values.

```
def f(x, L=[ ]):
    L.append(x)
    print(L)
```

defaultL —> [ ]

if given as input, use that.
if not given as input, use an alias of defaultL.

```
f(1)    # expect: [1]    reality: [1]
f(2)    # expect: [2]    reality: [1, 2]
```

Default argument is evaluated **<u>once</u>** when function is defined.

Default argument values

Need to be careful with default argument values.

A way to fix this:

```
def f(x, L=None):
    if(L == None): L = [ ]
    L.append(x)
    print(L)

f(1)    # expect: [1]   reality: [1]
f(2)    # expect: [2]   reality: [2]
```

Keyword arguments

```
def f(x, y, z):
    print(x, y, z)
```

```
f(1, 2, 3)
```

```
f(1, z=3, y=2)
```

keyword arguments

```
canvas.create_rectangle(0, 0, 50, 50,
                        fill="green", outline="red", width=3)
```

keyword arguments

Variable-length argument list

```
def longestWord(*args):        * "packs" arguments into one tuple
    if (len(args) == 0): return None
    result = args[0]
    for word in args:
        if (len(word) > len(result)):
            result = word
    return result

print(longestWord("this", "is", "really", "nice"))
```

The * makes args = ("this", "is", "really", "nice")

Variable-length argument list

```
def longestWord(*args):
    if (len(args) == 0): return None
    result = args[0]
    for word in args:
        if (len(word) > len(result)):
            result = word
    return result
```

```
print(longestWord("this", "is", "really", "nice"))
```

```
words = ("this", "is", "really", "nice")
print(longestWord(words))
```

Not what you want: args = (("this", "is", "really", "nice"),)

Variable-length argument list

```
def longestWord(*args):
    if (len(args) == 0): return None
    result = args[0]
    for word in args:
        if (len(word) > len(result)):
            result = word
    return result

print(longestWord("this", "is", "really", "nice"))

words = ("this", "is", "really", "nice")
print(longestWord(words[0], words[1], words[2], words[3]))
```

Variable-length argument list

```
def longestWord(*args):
    if (len(args) == 0): return None
    result = args[0]
    for word in args:
        if (len(word) > len(result)):
            result = word
    return result

print(longestWord(“this”, “is”, “really”, “nice”))

words = (“this”, “is”, “really”, “nice”)
print(longestWord(*words))
```

* “unpacks” the tuple

Variable-length keyword argument list

Same idea works for keyword arguments
Now use **

```
def f(x, **kwargs):
    return (x, kwargs)
```

** "packs" keyword arguments into a dictionary

```
print(f(1, y=2, z=3))
```

1 {'y': 2, 'z': 3}

```
print(f(1, y=2, z=3, a=4, b=5))
```

1 {'z': 3, 'a': 4, 'b': 5, 'y': 2}

```
d = {'a': 4, 'b': 5, 'c': 6}
print(f(1, **d))
```

1 {'a': 4, 'b': 5, 'c': 6}

** "unpacks" the dictionary

Anonymous functions

<u>Motivational example:</u>

```
class Book(object):
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year

def getYear(book):
    return book.year

b = Book("Hamlet", "Shakespeare", 1603)
```

Creates an object of type Book

```
print (getYear(Book("It", "Stephen King", 1986)))
```

Didn't assign it to a variable first

Anonymous functions

Motivational example:

```
class Book(object):
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year

def getYear(book):
    return book.year

b = Book("Hamlet", "Shakespeare", 1603)
```

Creates an object of type Book

```
library = [ ]
library.append(Book("It", "Stephen King", 1986))
```

Anonymous functions

Can - sort of - create and use functions similarly with lambda expressions.

```
f = lambda x,y: x+y
```

Creates an object of type function

Same as:

```
def f(x,y):
    return x+y
```

Anonymous functions

Can - sort of - create and use functions similarly with lambda expressions.

```
f = lambda x,y: x+y
```

inputs

Creates an object of type function

Anonymous functions

Can - sort of - create and use functions similarly with lambda expressions.

```
f = lambda x,y: x+y
```

an expression (the value is returned)

Creates an object of type function

```
f = lambda x,y: print(x+y)  # Crashes
```

Anonymous functions

Can - sort of - create and use functions similarly with lambda expressions.

f = lambda x,y: x+y   an expression (the value is returned)

Creates an object of type function

```
someFunctions = [ ]
someFunctions.append(lambda x,y: x+y)
```

From the run function in animation framework:

```
root.bind("<Button-1>", lambda event: mousePressedWrapper(event, canvas, data))
```

Nested functions

Can be used to avoid "polluting" the global space.

```
def f(a):
    def evens(a):
        return [value for value in a if (value % 2) == 0]
    return list(reversed(evens(a)))


print(f(range(10)))
print(evens(range(10)))  # Crashes
```

Nested functions

Can be used to change function signature.

```
def nQueens(n):
    def solve(n, m, constraints):
        ...

    return solve(n, n, [])
```

Nested functions

Can be used to change function signature.

Suppose you have a math function $f(x, y)$

For each fixed y, $f(x, y)$ defines a function in one variable: $f_y(x)$

Example: $f(x, y) = x + y$

$$f_0(x) = x$$

$$f_1(x) = x + 1$$

$$f_2(x) = x + 2$$

$$\ldots$$

f(x,y) is like a collection of functions in one variable.

How can we generate these functions in Python?

Nested functions

How to do this in Python:

```
def f(y):
    def g(x):
        return x + y
    return g
```

y is called a "non-local" variable.

→ For each y, this returns a different function

```
f_1 = f(1)
print(f1(5))
f_3 = f(3)
print(f_3(5))
```

Returned value:
g packaged together with a y value

Closure: a function bound together with a value