Topic 5
Functions and recursion

CS 1MD3 • Introduction to Programming
Winter 2018

Dr. Douglas Stebila

McMaster University

Functions

- A way of grouping together a sequence of operations under a common name so we can refer to it multiple times later
- Similar to mathematical functions: cos, exp, log, etc.

Syntax of functions

def used to define a function

Same rules for naming functions as for variables

0 or more variable names, which will be assigned to whatever values the function caller provides

```
def  name_of_function(list_of_inputs):
    body_of_function
    return value_to_return
```

The output to give back to the function caller

Body and return statement have to be indented

Function parameters

```
def change(target, coins):
    # some code here
    return d

t = 185
c = [5, 10, 25, 100, 200]
change(t, c)
```

Formal parameters

Actual parameters

- When a function is defined, the **formal parameters** are included in the function definition
- When the function is invoked (called), the **actual parameters** are provided
  - During invocation, the formal parameters are **bound** to the actual parameters
  - Python uses **pass by assignment** in which each formal parameter is assigned (using =) to the actual parameter

Default parameters

```
def sort(values, ascending = True):
    # your code here
    return whatever

sort(L)
sort(L, True)
sort(L, False)
sort(L, ascending = False)
```

- When defining a function, can provide default values for function parameters
  - Need to have parameters with default values after all parameters without default values
- When calling a function, can omit optional parameters which will then be assigned the default value
  - Careful you don't get confused when there are multiple optional values

SCOPING

Scoping variables

- **Scope:** where in your code a variable is available
  - **Global variables:** available to the main program and to all functions
  - **Local variables:** only available to the function they're defined in

Example: Computing factorial
No function

Scope of n

Scope of x

```
n = 7
x = 1
for i in range(2,
n+1):
    x *= i

print(x)
```

- All of the code is at the "top level" of our program
- Once this code is run, we can't call it again with a different input
  - No reuse of code
  - No modularity
- All variables are global

Example: Computing factorial Function

Scope of n

Scope of x

Scope of i

```
def factorial(n):
    x = 1
    for i in range(2, n+1):
        x *= i
    return x

print(factorial(6))
print(factorial(4))
```

- The definition of factorial and the print statements are "top level" code, but the red lines are local to the function
- We can use the factorial function many times, with different inputs
  - Enables code reuse
  - "Modular design"

Example: Computing factorial Function

Scope of n

Scope of x

Scope of i

```
def factorial(n):
    x = 1
    for i in range(2, n+1):
        x *= i
    return x

print(factorial(6))
print(factorial(4))

print(x)
print(n)
```

- x and n are **local variables** within the function factorial
- Can't be used outside the function

Example: Computing factorial
Function with a global variable

Scope of n

Scope of res

Scope of i

```
def factorial():
    res = 1
    for i in range(2, n+1):
        res *= i
    return res

n = 6
print(factorial())
```

- n is a **global variable**
- Even though n isn't defined when we define the function factorial, n is defined by the time we first run the function factorial
- Functions can read global variables outside the function

Example: Computing factorial Function with a global variable

Scope of n

Scope of res

Scope of i

```
def factorial(n):
    for i in range(2, n+1):
        res *= i

res = 1
factorial(7)
print(res)
```

- res is a **global variable**
- Even though n isn't defined when we define the function factorial, n is defined by the time we first run the function factorial
- But we get an error because functions cannot modify global variables (without permission)

Example: Computing factorial
Function with a global variable

Scope of n

Scope of res

Scope of i

```
def factorial(n):
    global res
    for i in range(2, n+1):
        res *= i

res = 1
factorial(7)
print(res)
```

- We can use the **global** keyword to allow a function to modify global variables

Example: Computing factorial Function with a global variable

Scope of n

Scope of i

Scope of res

```
def factorial(n):
    global res
    for i in
     range(2, n+1):
        res *= i

res = 1
factorial(7)
print(res)

factorial(4)
print(res)
```

- Have to be careful of **side-effects** when using global variables
- Generally you should try to avoid using global variables
- Functions are best when they can be run and tested in isolation, "communicating" with other code only via their explicit inputs and outputs (return)

SOFTWARE DESIGN

Software design principles

- **Modularity / decomposition**: a program is broken into parts (functions) that are
  - reasonably self-contained
  - achieve a clear purpose, and
  - can be reused
- **Abstraction:** a component (function) of a program can be used without knowing how it achieves its goal

Functions in large programs

- Large programs will have many functions
- Each function should be relatively small and be designed to do one thing only
- Example: Twitter
  - 1 function to store a user's new tweet in the database
  - 1 function to record a user liking a tweet
  - 1 function to delete a user's tweet from the database
  - ...

Documenting functions

- We can provide a **docstring** when we define a function that describes how the function should be used
- Two components:
  - **Assumptions / requirements / preconditions:** what conditions the **input** must satisfy for the function to work correctly
  - **Guarantees / postconditions:** what conditions the **output** will satisfy (as long as the assumptions were satisfied)

Docstring example

```
def makingChange(target, coins):
    """Assumes target is an int and that
       coins is a sequence of ints in
       decreasing order.
       Returns a dictionary with a key for each
       entry in coins and a corresponding int for
       the number of coins of that denomination."""
    change = {}
    for c in coins:
        change[c] = target // c
        target %= c
    return change

makingChange(185, (200, 100, 25, 10, 5))
```

Getting docstring help

```
help(makingChange)

Help on function makingChange in module
__main__:


makingChange(target, coins)
   Assumes target is an int and that coins is
a sequence of ints in decreasing order.
   Returns a dictionary with a key for
each entry in coins and a corresponding int
for the number of coins of that denomination.
```

RECURSION

Factorial recursively

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n - 1)! & \text{if } n > 0 \end{cases}$$

Base case

Iterative case (recursion)

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

factorial(5)
```

Base case

Iterative case (recursion)

Factorial recursively

```
def factorial(n):
    print("entered factorial with n = " + str(n))
    if n == 0:
        print("about to return from base case")
        return 1
    else:
        f = factorial(n-1)
        print("about to return from iterative case in n = " + str(n))
        return n * f

factorial(5)

entered factorial with n = 5
entered factorial with n = 4
entered factorial with n = 3
entered factorial with n = 2
entered factorial with n = 1
entered factorial with n = 0
about to return from base case
about to return from iterative case in n = 1
about to return from iterative case in n = 2
about to return from iterative case in n = 3
about to return from iterative case in n = 4
about to return from iterative case in n = 5
120
```

Recursion

- Every time we recurse the interpreter creates a new **scope** for the variables in the function

Problem solving with recursion

- Recursion is a natural technique for solving problems with an inductive structure

- But depending on the structure of the problem, recursion can be inefficient

Fibonacci sequence, recursively

$$\mathrm{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{if } n > 1 \end{cases}$$

Base case

Iterative case

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

fib(4)
```

Counting amount of recursion in Fibonacci

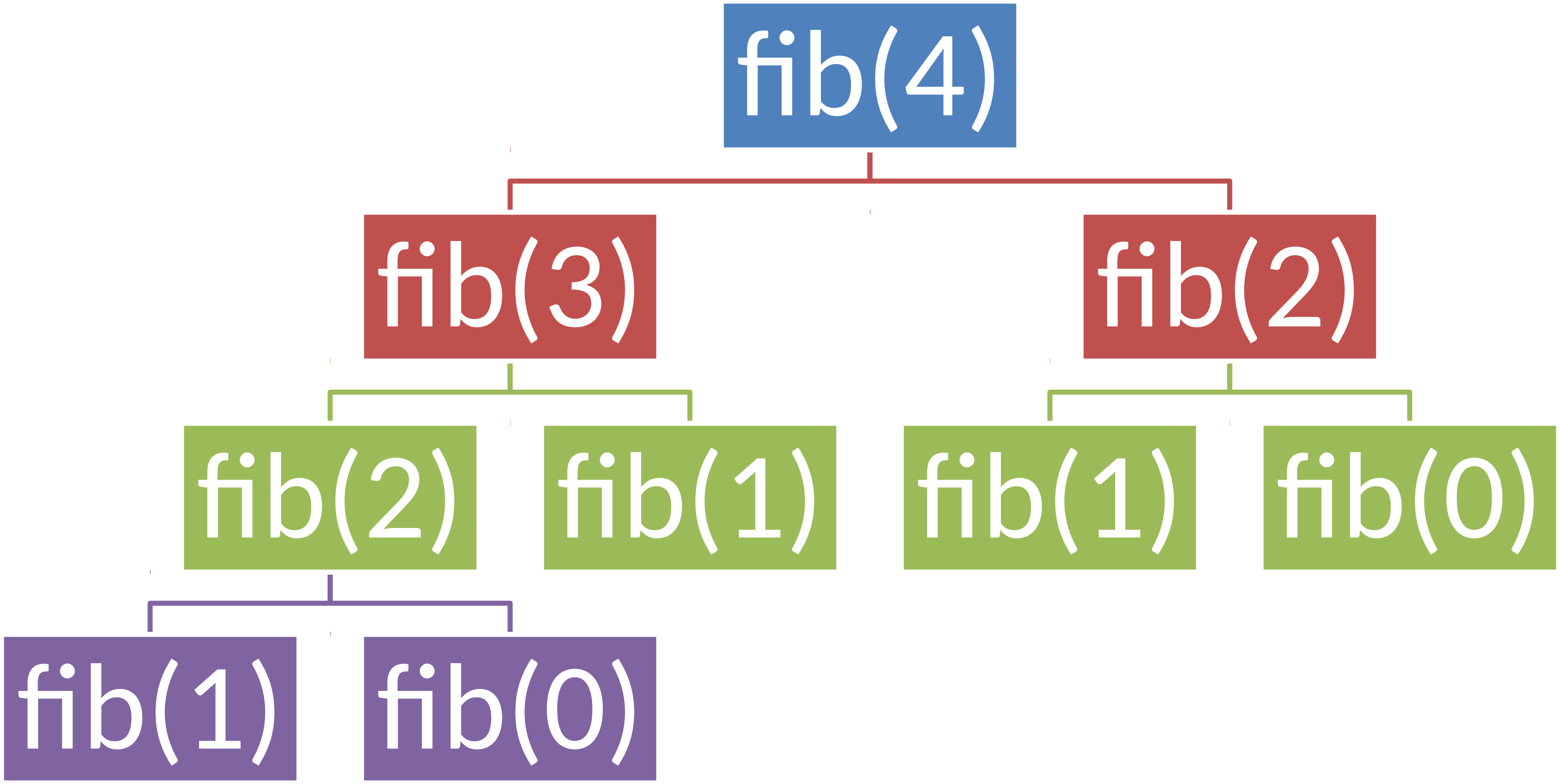
```
def fib(n):
    global counter
    counter += 1
    if (n == 0) or (n == 1):
        return 1
    else:
        return fib(n-1) + fib(n-2)

counter = 0
fib(4)
print(counter)
```

Question: How many times does fib get called? In other words, what is the value of counter at the end?

- A: 4 B: 5 C: 7 D: 9 E: Don't know

Counting amount of recursion in Fibonacci



Counting amount of recursion in Fibonacci

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

counter = 0
fib(4)
print(counter)
```

A purely recursive solution to computing the Fibonacci sequence is inefficient because it has to recompute the same values many times.

A more advanced technique called **dynamic programming** remembers intermediate solutions to save on recursion.

fib(n)	counter
fib(4)	9
fib(6)	25
fib(8)	67
fib(10)	177

Infinite recursion

- Just like infinite loops, have to be careful for infinite recursion
- Can occur when a base case is missing
  - Or accidentally gets bypassed
- Need to be confident our recursion terminates

```
def twofib(n):
    if (n == 0) or (n == 1):
        return 1
    else:
        return twofib(n-1) + \
        twofib(n-3)

twofib(4)
```

$$\text{twofib}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ \text{twofib}(n-1) + \text{twofib}(n-3) & \text{if } n > 1 \end{cases}$$

Measuring efficiency

Theoretical

- **Runtime complexity**: what is the approximate number of basic operations the algorithm will perform for a certain set of inputs?
- **Memory complexity**: what is the approximate number of entries that the algorithm will read/write for a certain set of inputs?

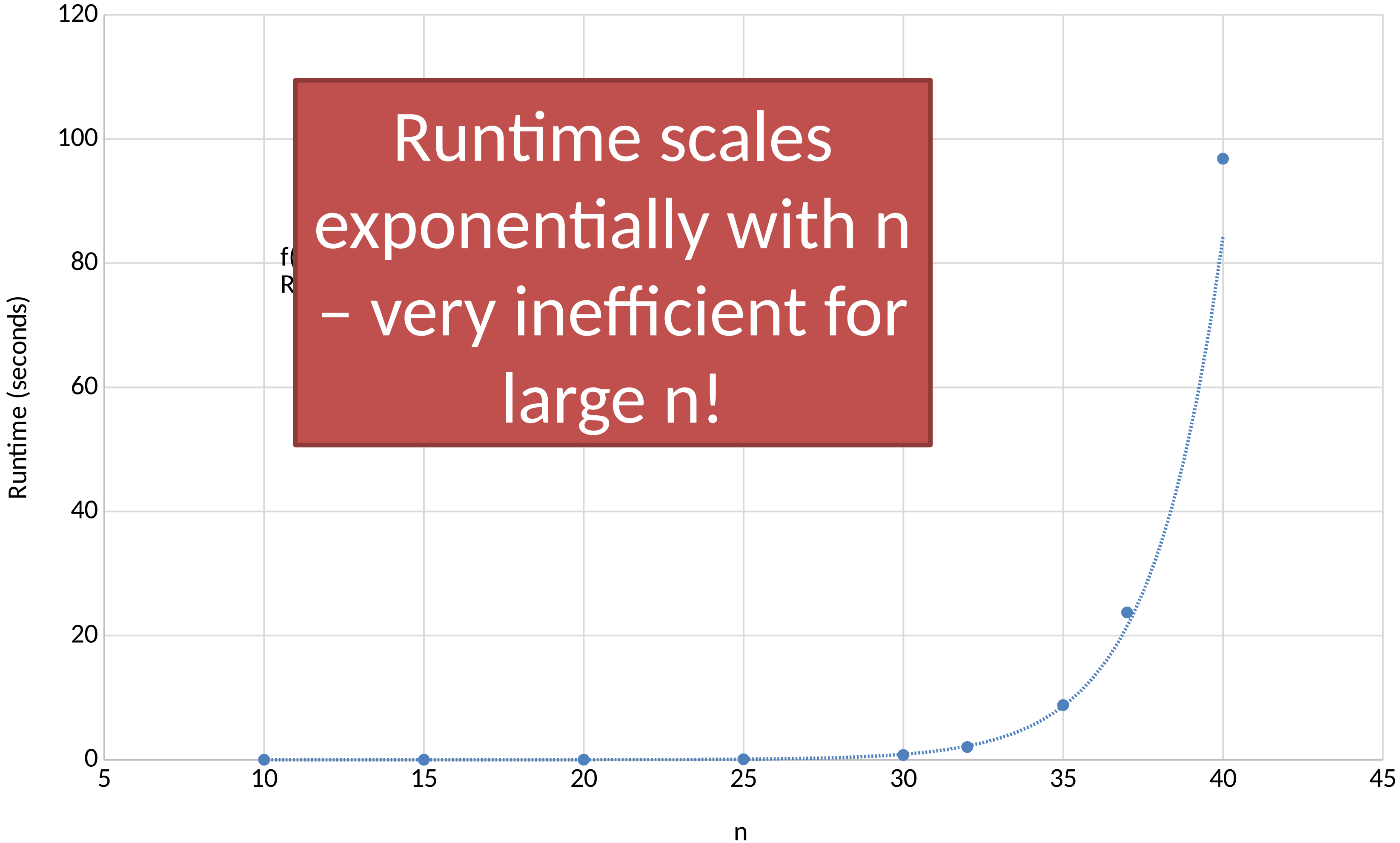
Practical

- **Runtime**: how many seconds does this implementation run for on a particular computer with a particular input?
- **Memory**: how many megabytes of RAM does this implementation use on a particular computer with a particular input?

Practical runtime of recursive Fibonacci fib(n)

n	Runtime (seconds)
10	0.000117
15	0.000918
20	0.008939
25	0.075447
30	0.767061
35	8.806806
40	96.826460

Practical runtime of recursive Fibonacci fib(n)



Memoization

- Idea: during recursion, save intermediate results, and use them if available to avoid recursion

Fibonacci with memoization

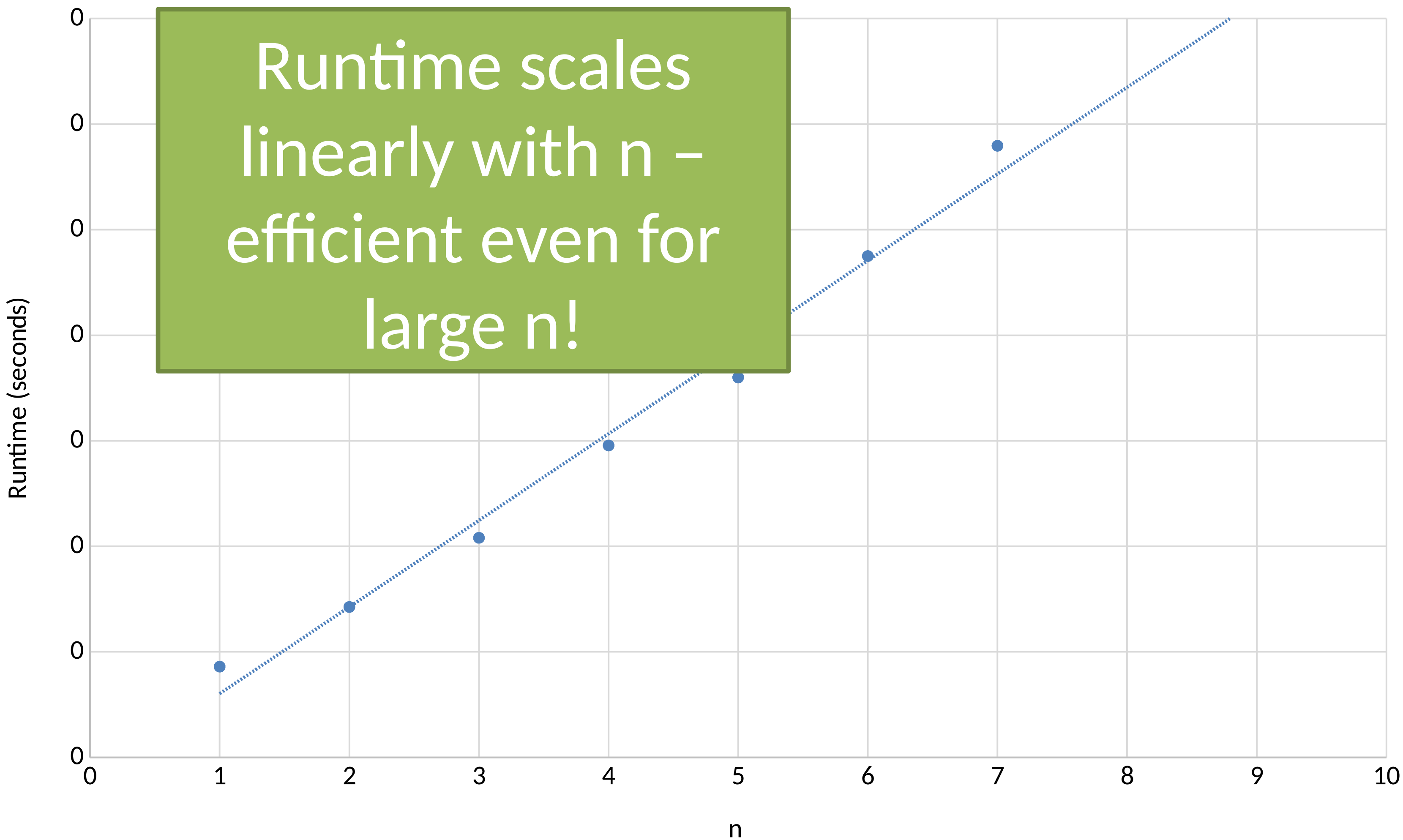
```
def fib_fast(n, memo):
    global counter
    counter += 1
    if (n == 0) or (n == 1):
        return 1
    else:
        if n-1 not in memo:
            memo[n-1] = fib_fast(n-1, memo)
        if n-2 not in memo:
            memo[n-2] = fib_fast(n-2, memo)
        return memo[n-1] + memo[n-2]

counter = 0
fib_fast(4, {})
print(counter)
```

Practical runtime of recursive Fibonacci with memoization fib_fast(n)

n	Runtime (seconds)
100	0.000172
200	0.000285
300	0.000416
400	0.000591
500	0.000720
600	0.000950
700	0.001159

Practical runtime of recursive Fibonacci with memoization fib_fast(n)



SEARCH

Search

- Suppose we have a sequence of length n and we want to know if a particular element is in the sequence
- How many entries of the sequence do we have to check to find it or know that it is not in the list?
  - If the sequence is not organized in any way, then in the worst case we have to check every one of the n entries
  - Maybe we can do better if the list is sorted.

Unstructured search

- Idea: Go through the entries in the sequence one at a time, checking if each one is the target element
- Inputs: a sequence and a target element
- Output: True or False

Unstructured search

```
def search1(L, e):
  for i in range(0, len(L)):
    if L[i] == e:
      return True
  return False
```

Unstructured search

- Unstructured search is sometimes called **linear search** because
  - It goes through the sequence in a linear fashion
  - Its runtime is a linear function of the list size

Searching a sorted list

- If the list is sorted, then we can make use of this organization to make the job easier
- Idea:
  - Assume the list is sorted in increasing order
  - Look at the entry in the middle of the list
    - If the middle entry is equal to the target entry, then we found it!
    - If the middle entry is bigger than the target entry, then we only need to look in the first half of the list
      - Recurse on the left half of the list
    - If the middle entry is smaller than the target entry, then we only need to look in the second half of the list

Example: searching a sorted list for 33

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4	6	8	9	14	16	17	22	23	29	32	43	44	51	63
									23	29	32	43	44	51	63
									23	29	32				
											32				

- Current list: [0:15]
  Middle entry: 22; smaller than target
  Recurse on right half
- Current list: [9:15]
  Middle entry: 43; larger than target
  Recurse on left half
- Current list [9:11]
  Middle entry: 29; smaller than target
  Recurse on right half
- Current list: [11:11]
  Only entry: not equal to target

Binary search using recursion

```
def search2(L, e):
    """Assume L is in increasing order"""
    if len(L) == 0:
        return False
    if len(L) == 1:
        return (L[0] == e)
    mid = len(L) // 2
    if L[mid] == e:
        return True
    elif L[mid] < e:
        return search2(L[mid+1:], e)
    else:
        return search2(L[:mid], e)
```

Binary search using recursion

- Binary search using recursion is much faster because it only has to look at a small number of items in the list
- If the list has length n, then binary search using recursion looks at approximately $\log_2(n)$ elements of the list

- But the version on the previous slide is still somewhat inefficient since it creates a new list every time by slicing
- Next slide shows binary search using recursion by keeping track of start and end points of the sub-list, rather than making a new list every time

Binary search using recursion

```
def search3(L, e):
    """Assume L is a list sorted in increasing order"""

    def search3Helper(L, e, low, high):
        if high == low:
            return (L[low] == e)
        mid = low + ((high - low) // 2)
        if L[mid] == e:
            return True
        elif L[mid] < e:
            return search3Helper(L, e, mid+1, high)
        else:
            return search3Helper(L, e, low, mid)

    return search3Helper(L, e, 0, len(L))
```

Notice we defined one function inside another. search3Helper is a local function, only available within the scope of search3 but not globally available.

FUNCTIONS AS OBJECTS

Functions as objects

- In Python, functions as **first-class objects**
- We can use functions like objects of other types
- In particular, we can pass functions as arguments to other functions

- This is why we say Python can behave like a functional programming language
  - Python is multi-paradigm: procedural, functional, …

Passing functions to functions

```
def map (L, f):
    """Assumes L is a list and f is a function.
    Modifies L by replacing each entry of L with
    f applied to that entry."""

    for i in range(0, len(L)):
        L[i] = f(L[i])
```

```
import math
L = [1,2,3]
map (L, math.cos)
print(L)
L = [1,2,3]
map (L, fib)
print(L)
```

Anonymous functions

- We can also make "anonymous" functions right when we need them
- Notation:
  `lambda <variable names>: <expression>`
- Examples:
  `lambda x, y: x** y`
  `lambda x: x ** 2`

```
L = [1,2,3]
applyToEach(L, lambda x: x**2)
print(L)
```