Exam 2 Review

15-110 – Monday 11/04

Announcements

- **Exam2 on Wednesday!**
  - Bring something to write with, your andrewID card, and a large piece of paper with your name and andrewID for us to collect
  - Arrive **early if possible** – we're checking IDs + collecting andrewID papers at the door
- Hw5 includes **Code Review #2**! Same rules as Code Review #1.
  - Timeslot signups will be released Thursday

General Exam Taking Tips:

- Skim all the questions before answering questions
    - You do not have to answer the questions in order
- Do not leave anything blank
- For code writing:
    - You can always at least write the function definition!
    - Make sure you understand the example
    - Do I need to return something?
        - Do I need to build up a result?
        - What is the type of the thing I need to return?
- For code reading:
    - Don’t do it all in your head
    - Write down variables, function calls, and track variable updates after each line of code
- Review the notes sheet before the exam so you know what is there

Review Topics

- Recursion
- Lists: Code Reading
- Graphs: Code Writing
- Big-O Calculation
- Heuristics
- Hashing reminders

Recursion

Recursion: Big Idea

The core idea of recursion is that we can solve problems by **delegating** most of the work instead of solving it immediately.

This works because we make the input to the problem **slightly smaller** every time the function is called. That means it will eventually hit a **base case**, where the answer is known right away.

Once the base case returns a value, all the recursive calls can start returning their own values up the **chain of function calls** until they reach the initial call, which returns the final result.

Writing Recursive Code: reverseList

When working with recursive code, it often helps to think **abstractly** about how to solve the problem with delegation before jumping into coding.

For example: what if we wanted to reverse a list using recursion? What is a **base case** that we can solve immediately?

In the recursive case, how do we make the problem smaller? What can we expect the recursive result to be **if the function works correctly**? Use that assumption to create the final result!

```
def reverseList(lst):
    if len(lst) == 0:
        return []
    else:
        smaller = reverseList(lst[1:])
        return smaller + [ lst[0] ]
```

Activity: Recursion Code Reading

It's important to understand **how recursive calls work** behind the scenes!

You do: trace by hand what the shown function call will output. Note the print statements!

```
def reverseList(lst):
    print("Call:", lst)
    if len(lst) == 0:
        print("Return:", [])
        return []
    else:
        smaller = reverseList(lst[1:])
        result = smaller + [ lst[0] ]
        print("Return:", result)
        return result

reverseList([3, 6, 9])
```

Recursion with Multiple Calls

Recursion is most powerful when we make **multiple recursive calls** in the recursive case. This allows us to solve problems we can't solve without recursion.

Example: fibonacci

The **Fibonacci sequence** is a sequence in which each number is the sum of the two preceding ones.

$F(0) = 0$

$F(1) = 1$

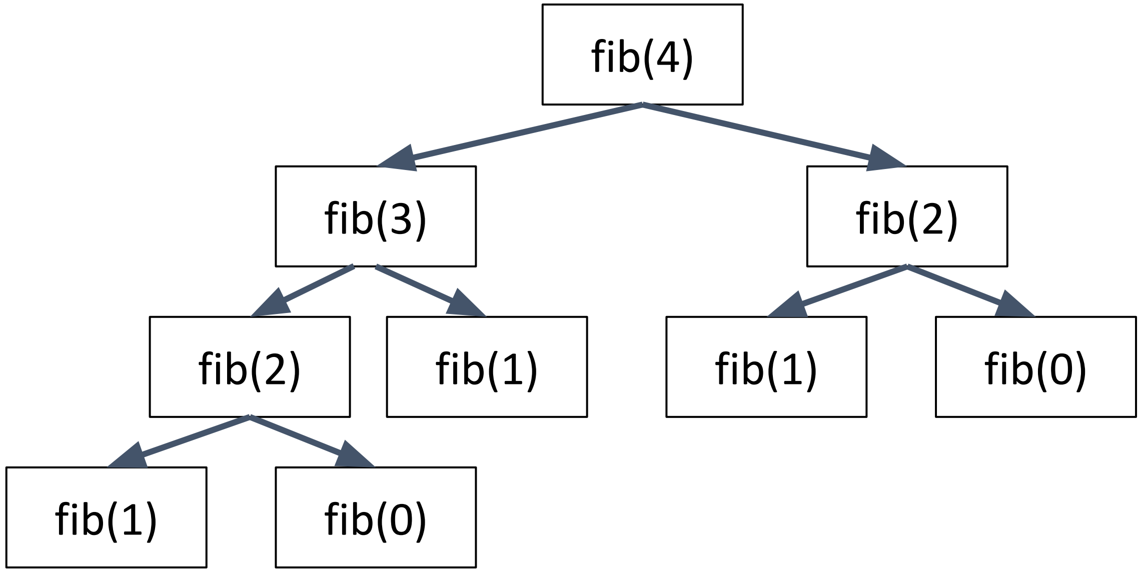
$F(n) = F(n-1) + F(n-2), n > 1$

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

Two recursive calls!

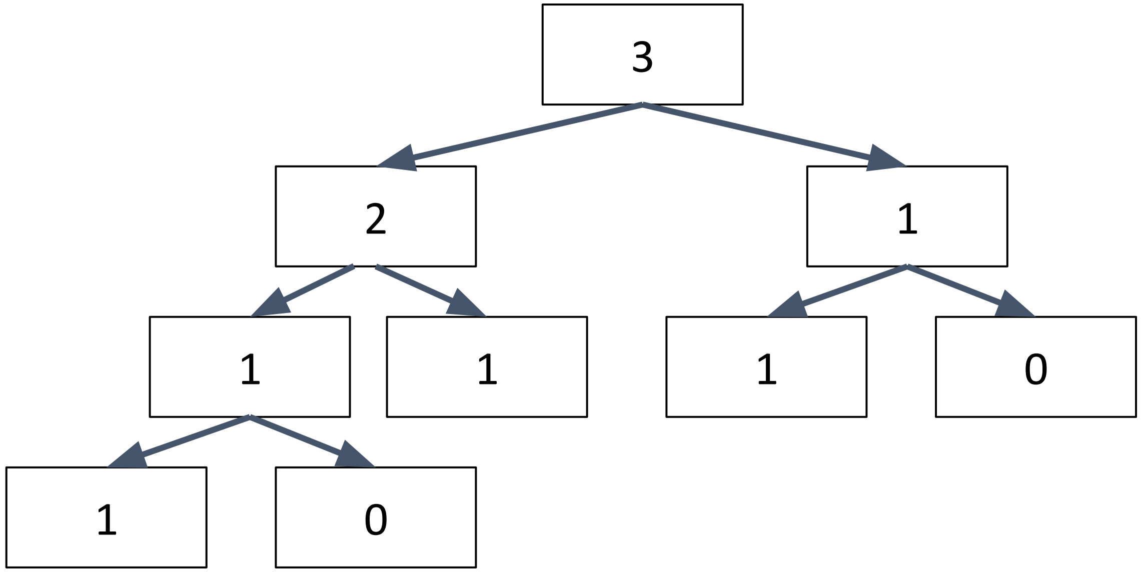
Fibonacci Recursive Call Tree

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1



Fibonacci Recursive Call Tree

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1



Lists: Code Reading

Activity: What does this print?

```
def foo(W):
    W.pop(2)
    return W + [1]

X = [2, 4, 6]
Y = X.append(3)
Z = foo(X)

print("X:",X)
print("Y:",Y)
print("Z:",Z)
```

Hints:
Is `Y` aliased to `X`?

Is `W` aliased to `X`?

Will `lst + <anything>` make a new list?

Does `lst.append` make a new list?

Graphs: Code Writing

Activity: `getOddWeights(g)`

Given a directed, weighted graph, return a list of all the odd weights in the graph.

For example, given the following graph:

```
g = {
    "A": [ ["B", 4], ["C", 7] ],
    "B": [ ["C", 1], ["D", 3] ],
    "C": [ ],
    "D": [ ["B", 1] ],
}
```

`getOddWeights(g)` would return `[7,1,3,1]`

Big-O Calculation

Big-O Essentials: What to Count?

When measuring the Big-O complexity of an algorithm, we must specify what it is we're counting. Some popular choices:

- comparisons: `target == lst[i]`
- assignments: `y[i+1] = x[i]`
- recursive calls: `recSearch(tree["left"], target)`
- all 'actions' in the program (all of the above, plus more)

Big-O Essentials: Find the Dominant Term

When calculating Big-O, we don't care about coefficients. An algorithms that makes 3n comparisons is considered just as fast as an algorithm that makes 2n comparisons: both are O(n).

Only the dominant term matters:

$O(1) < O(\log n) < O(n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

Big-O Essentials: Mind the Exponent

When dealing with Big-O equations, n is the size of the input and k is some constant number.

$O(n^k)$ is polynomial in n and considered tractable, because k is **constant**

$O(k^n)$ is exponential in n and considered "slow" (intractable) because n is **variable** and will grow over time

When is an algorithm O(n)?

Any algorithm that processes each element once is O(n).

- Add up the elements of a list
- Sum the numbers from 1 to n
- See if a list contains an odd number
- Find the index of the first even number

When is an algorithm $O(n^2)$?

Doing an $O(n)$ operation on every element of a list means the total number of operations is $O(n^2)$.

Common example: nested for loops that both do n iterations:

```
for i in range(len(lst)):
    for j in range(len(lst)):
        if (i != j) and (lst[i] == lst[j]):
            print(lst[i], "is duplicated")
```

When is an algorithm $O(n^2)$?

An algorithm can be $O(n^2)$ even if it has just one loop!

```
for i in range(len(lst)):
    if lst[i] in (lst[:i] + lst[i+1:]):
        print(lst[i], "is duplicated")
```

The `in` test on a list is itself $O(n)$ and it is inside a for loop that does n iterations, so the algorithm is $O(n^2)$.

When is an algorithm O(log n)?

If we cut the problem size in half each time and only consider one of the halves, we can make $\log_2(n)$ such cuts, so the algorithm is O(log n).

For example, binary search cuts the list in half each time, so it is O(log n).

Suppose we want the first digit of a long number:

```
while n > 9:
    n = n // 10
```

This code makes $\log_{10}(n)$ divisions, so it is also O(log n).

When is an algorithm $O(2^n)$?

If we have a recursive algorithm operating on an input of size n and each call makes two recursive calls of size n-1, then the algorithm is $O(2^n)$. The number of calls **doubles** every time we increase the size by 1.

```
def abCombos(n, s):
    if n == 0:
        print(s)
    else:
        abCombos(n-1, s + "a") # first recursive call
        abCombos(n-1, s + "b") # second recursive call
```

When is an algorithm $O(2^n)$?

If we have a recursive algorithm and each call produces a result twice as long as the previous result, then the algorithm is also $O(2^n)$.

```
def allSubsets(lst):
    if lst == [ ]:
        return [ lst ]
    else:
        result = [ ]
        subsets = allSubsets(lst[1:])
        for s in subsets:
            result.append(s)
            result.append([ lst[0] ] + s)
        return result
```

Activity: Compute the Big-O

Consider the following function. What is its Big-O runtime in the worst case?

```
def example(s):
    result = ""
    for i in range(len(s)//2, len(s)):
        result = s[i] + result

    for j in range(len(s)//2):
        if s[j].isupper():
            result = result + s[j].lower()
        else:
            result = result + s[j]
    return result
```

Heuristics

A **heuristic** is a search technique used by an algorithm to **find a good-enough approximate solution** to a problem.

Heuristics may not find the best answer to an NP problem, but they **often achieve good results**.

A heuristic can **generate scores to rank potential next steps** that the algorithm can take at each decision point. By choosing the highest-scored next step, the algorithm is more likely to find a working solution quickly.

Heuristic solutions are not necessarily optimal solutions.

Big-O Essentials: What to Count?

When measuring the Big-O complexity of an algorithm, we must specify what it is we're counting. Some popular choices:

- comparisons: `target == lst[i]`
- assignments: `y[i+1] = x[i]`
- recursive calls: `recSearch(tree["left"], target)`
- all 'actions' in the program (all of the above, plus more)

Hashing reminder

A good hash function must:

1. Turn immutable values into integers

2. For any input, always return the same output

3. Generally return different outputs for different inputs