Exam 2 Review

15-110 – Monday 03/25

Announcements

- **Exam2 on Wednesday!**
  - Bring your paper notes (<= 5 pages), something to write with, and your andrewID card
  - Arrive **early if possible** – we're checking IDs at the door
  - Lecture 2 only: DH 2315

- Hw5 includes **Code Review #2**! Same rules as Code Review #1.
  - Timeslot signups will be released tomorrow

Review Topics

- Recursion
- Hashed Search
- Big-O Calculation

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

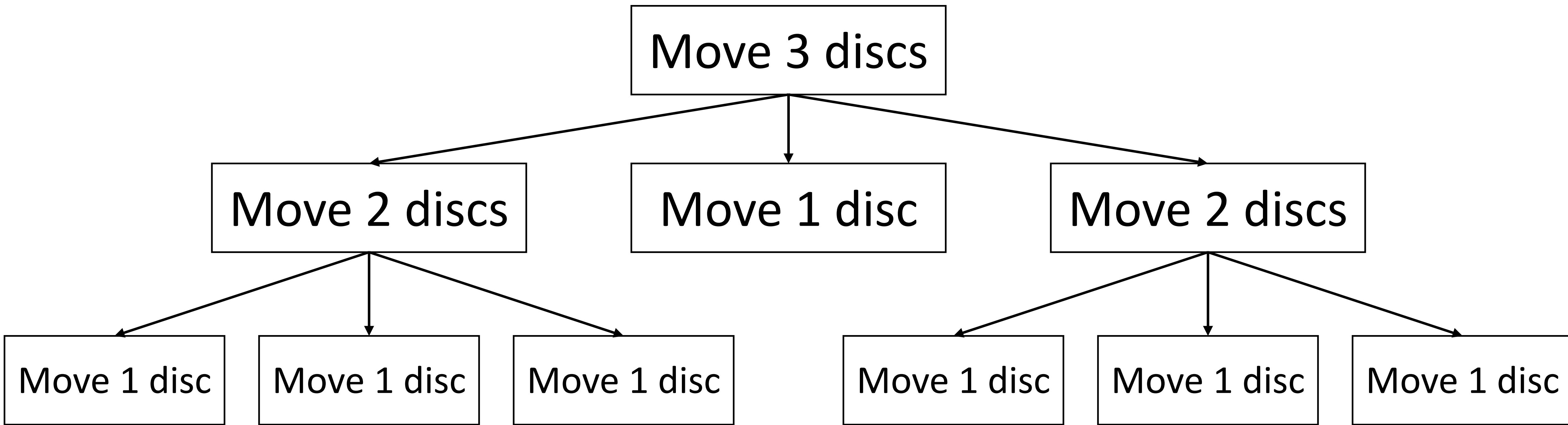
Recursion is most powerful when we make **multiple recursive calls** in the recursive case. This allows us to solve problems we can't solve without recursion.

Towers of Hanoi (https://www.mathsisfun.com/games/towerofhanoi.html) is an example of a problem you can only solve using recursion.

- Move all but one of the discs to the temporary position – **recursive call**
- Move the remaining disc to the goal – **base case**
- Move the all-but-one discs to the goal – **recursive call**

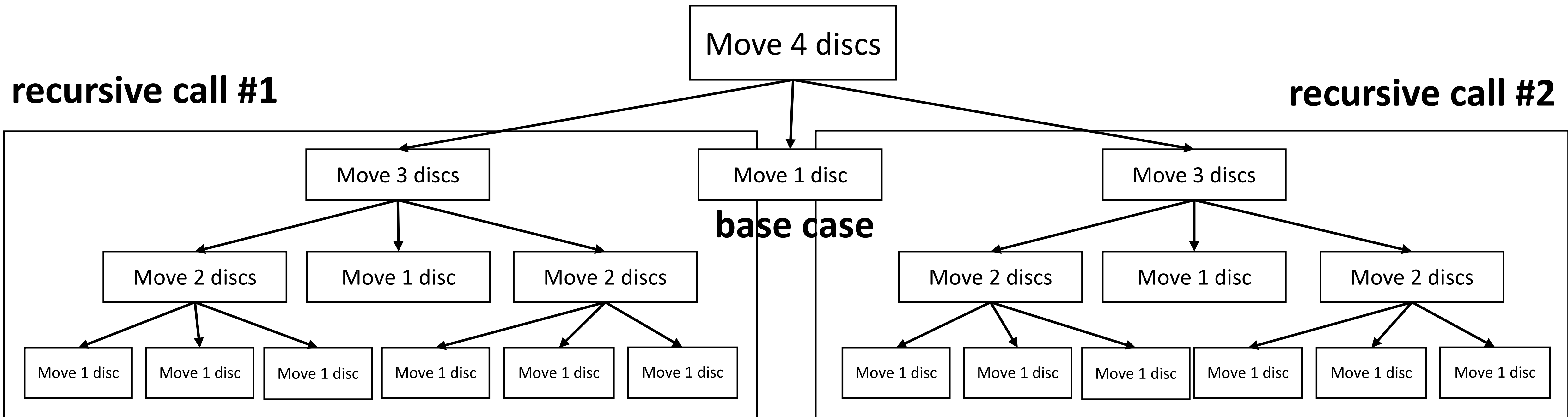
Towers of Hanoi – Visualize Moving 3 Discs

To move 3 discs, move two, then the one remaining, then move the two again.



Towers of Hanoi – Visualize Moving 4 Discs

To move 4 discs, move 3 discs by **following the same procedure we outlined before,** just with a different destination. Use **abstraction** to solve the problem!



Hashed Search

Big Idea

Why do we care about hash functions?

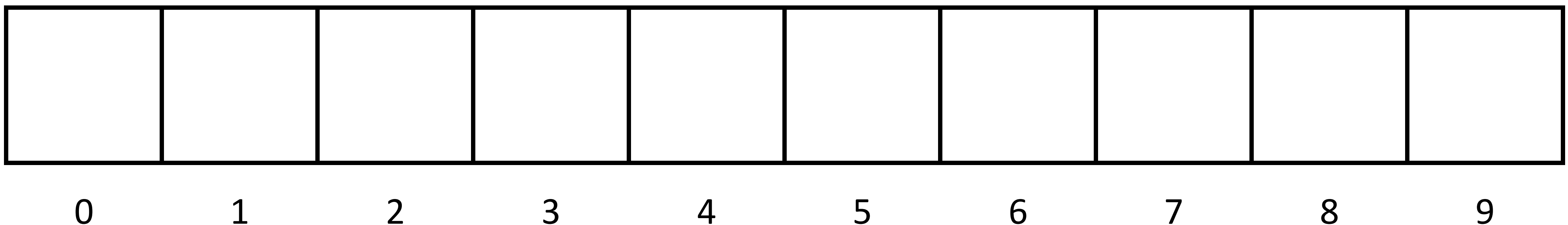
We search all the time, so we want the fastest possible search. Storing items in a hashtable lets us look up whether an item is in the table in **O(1)** time. You can't get faster than that!

How can we search in constant time? The algorithm needs to know **where** the value it's looking for will be stored **if** that value is actually in the table.

Hashtables

A **hashtable** is like a big, empty list of a designated size. Like in a list, each slot ('bucket') in the table is associated with an **integer index**, from 0 to len(table)-1.

When we want to put a value in the hashtable, we insert it at a specific index based on the result of a **hash function**.



Hash Functions

A **hash function** is a function that maps Python values to integers. Those integers can then be used to find an index in the hashtable to store the value.

We can use the built-in Python hash function or write our own. Either way, the hash function must follow two rules:

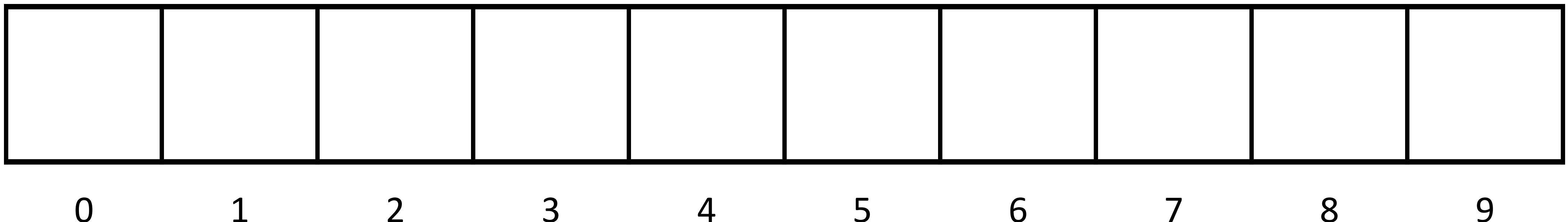
- The result returned when the function is called on some value must not change across calls
- The function should usually return different numbers when called on different values

Storing/Finding Values in Hashtables

Both storing a value in a hashtable and checking whether a value is in a hashtable follow the same procedure, which produces which index to check.

1. Run the hash function on the value to get the hashed value.
2. Mod the hashed value by the hashtable size to get the final index

Demo: Let's practice with some strings and the built-in hash function.



Why O(1)?

Why is looking up a value in a hashtable O(1) time?

We don't need to check every bucket in the hashtable. Only look in one bucket- the one with the index associated with the hashed value.

Important: this only works if the value we're searching for can't change (it's **immutable**) and if the hashtable is **large enough** for the stored values to spread out. (10 buckets isn't nearly enough!)

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
```