CS 5001

LECTURE 10

ALGORITHMIC COMPLEXITY, SEARCHING & SORTING

KEITH BAGLEY

FALL 2023

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Course PSA
- Course Check in
- Review of Computation & Algorithms
- Algorithm Efficiency
- Simple Algorithm Analysis
- Simple Searching
- Sorting
- Q&A

COURSE PSA

[Review Similarity](#) > [redacted]

[redacted]

```python
    emoji_mappings = {}
    try:
        with open(emoji_file_name, 'r') as emoji_file:
            metadata = emoji_file.readline().strip().s
            emoji_types = metadata[1:] # Extract emoji
            english_index = None

            # find the index of 'english'
            for i, emoji_type in enumerate(emoji_types
                if emoji_type.lower() == 'english':
                    english_index = i
                    break

            if english_index is None:
                print("Error: 'english' not found in e
                return emoji_mappings

            # read emoji mappings and store them in th
            for line in emoji_file:
                data = line.strip().split()
                english = data[english_index].lower()

                emoji_data = {}
                for i, emoji_type in enumerate(emoji_t
                    emoji_type = emoji_type.lower()
                    if i != english_index:
```

```python
    '''
    emoji_mapping = {}
    try:
        with open(emoji_file_name, 'r') as emoji_file:
            metadata = emoji_file.readline().strip().s
            # Extract emoji types from metadata
            emoji_types = metadata[1:]
            english_index = None

            # Find the index of 'english'
            for i, emoji_type in enumerate(emoji_types
                if emoji_type.lower() == 'english':
                    english_index = i
                    break

            if english_index is None:
                print("Error: 'english' is not found i
                return emoji_mapping

            # Read emoji mappings and save them in the
            for line in emoji_file:
                data = line.strip().split()
                english = data[english_index].lower()

                emoji_data = {}
                for i, emoji_type in enumerate(emoji_t
                    emoji_type = emoji_type.lower()
```

Show Common Matches

[redacted]

emojini.py

100%

Print Report

1 - 73	1	26 - 92
73 - 126	2	93 - 143
128 - 149	3	7 - 26

[redacted]

Don't let this be you!
Maintain your integrity!

ANY infractions on the Project will automatically FAIL the course!

CHECK-IN: ALMOST THERE!

Final Exam Next Week!
Similar to what we did for Midterm:
"Window of opportunity" to take it

WHAT'S LEFT?

- Algorithmic Analysis Introduction (plus search/sort) – Today
- Review Lab - Thursday
- Software Engineering Overview – next week ½ lecture
- Overview of Khoury Co-op/Career Services (guest speaker ½ lecture)
- Final Exam (Dec 03 – 06 window)
- Submit Project

- Celebrate, Rest, and Recuperate

QUOTE OF THE WEEK

INTRO TO ALGORITHMIC ANALYSIS

- We'll cover this topic lightly today. You'll go into much more detail in your upcoming 5008 course next semester

REVIEW: ALGORITHMS

- Algorithm (simple definition): An ordered set of unambiguous steps that solves a problem in a finite amount of time
- The definition allows us to talk computationally & implement using machines
- The definition also allows us to explore algorithms manually. Algorithms may or may not be automated using computers

ALGORITHM EFFICIENCY

How efficient is that algorithm?

Independent of…

ALGORITHM EFFICIENCY

How efficient is that algorithm?

Independent of…

- Programming Language

C is more efficient than Python, but that's not the point

ALGORITHM EFFICIENCY

How efficient is that algorithm?

Independent of…

Programming Language

- **Processor Speed**

`Upgrading to a better computer is not the point`

ALGORITHM EFFICIENCY

How efficient is that algorithm?

Independent of…

Programming Language

Processor Speed

- **Implementation Details**

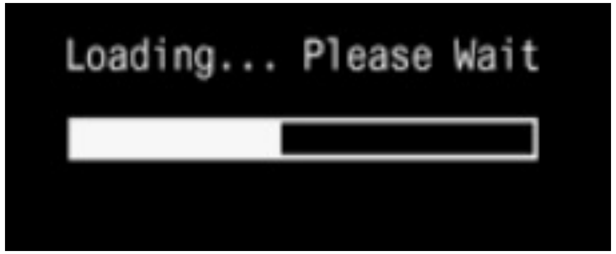
You wrote five conditionals when it should have been two.

WHAT IS IT, AND CAN WE DO BETTER?

- Given an algorithm, we can ask the questions:
  - Which complexity class (category) is it in?
  - Can we do better than the complexity class? Different algorithm more suitable/efficient?

WHY DO WE CARE?

- Turn Left Here! ☺



COMPLEXITY CLASS (BIG PICTURE)

- (1) Count the steps
- (2) Drop the coefficients & lower-order terms
- (3) Identify the upper-bound
  - This is the “worst-case analysis” we talked through last lecture

(1) COUNT THE STEPS

- Can use a language like Python BUT
  - Often, we haven’t coded the algorithm yet
  - SO – more often we use pseudo-code to spec out the algorithm & count steps

START WITH PSUEDOCODE

```
FUN(A):
    variable
    for loop
        stuff inside a for loop
    conditional
        do this if true
    return a thing
```

COUNT THE STEPS

```
FUN(A):
    variable
    for loop
        stuff inside a for loop
    conditional
        do this if true
    return a thing
```

Input: A is a list of length *n*

COUNT THE STEPS

FUN(A):

variable

for loop

stuff inside a for loop

conditional

do this if true

return *a thing*

```
One line of
code == one
"step"
```

COUNT THE STEPS

FUN(A):

variable

for loop

stuff inside a for loop

conditional

do this if true

return *a thing*

Loop runs *n* times? Count *n* x steps inside.

COUNT THE STEPS

FUN(A):

variable

for loop

stuff inside a for loop

conditional

do this if true

return *a thing*

Assume the worst and count this step.

COUNT THE STEPS

FUN(A):

variable

for loop

stuff inside a for loop

conditional

do this if true

return *a thing* → One more step

DROP COEFFICIENTS & LOWER-ORDER TERMS

- (1) Count the steps
- (2) Drop the coefficients & lower-order terms
- (3) Identify the upper-bound
  - This is the “worst-case analysis” we talked through last lecture

Slow-growing terms are relatively small compared to fast-growing ones

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$5n + 20$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \Longrightarrow n$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{2}0 \Longrightarrow \boldsymbol{n}$$

$$\mathbf{2}\boldsymbol{n} + \mathbf{lg}\ \boldsymbol{n}$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \implies n$$

$$\cancel{2}n + \cancel{\lg n} \implies n$$

Remember:
We're dropping these terms because the slower-growing terms have minimal impact. We're looking at the "big picture" here to get a category, not super-precision

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \Longrightarrow n$$

$$\cancel{2}n + \cancel{\lg}\, n \Longrightarrow n$$

$$n^2 + \lg n$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \Longrightarrow n$$

$$\cancel{2}n + \cancel{\lg n} \Longrightarrow n$$

$$n^2 + \cancel{\lg n} \Longrightarrow n^2$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \implies n$$

$$\cancel{2}n + \cancel{\lg n} \implies n$$

$$n^2 + \cancel{\lg n} \implies n^2$$

$$4n^2 + 256n$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \implies \boldsymbol{n}$$

$$\cancel{2}n + \cancel{\lg n} \implies \boldsymbol{n}$$

$$n^2 + \cancel{\lg n} \implies \boldsymbol{n^2}$$

$$\cancel{4}n^2 + \cancel{256n} \implies \boldsymbol{n^2}$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \Longrightarrow n$$

$$\cancel{2}n + \cancel{\lg n} \Longrightarrow n$$

$$n^2 + \cancel{\lg n} \Longrightarrow n^2$$

$$\cancel{4}n^2 + \cancel{256n} \Longrightarrow n^2$$

$$n^3 + n^2 + \lg n + 25$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \Longrightarrow \boldsymbol{n}$$

$$\cancel{2}n + \cancel{\lg n} \Longrightarrow \boldsymbol{n}$$

$$n^2 + \cancel{\lg n} \Longrightarrow \boldsymbol{n^2}$$

$$\cancel{4}n^2 + \cancel{256n} \Longrightarrow \boldsymbol{n^2}$$

$$n^3 + \cancel{n^2} + \cancel{\lg n} + \cancel{25} \Longrightarrow \boldsymbol{n^3}$$

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \implies \mathbf{n}$$

$$\cancel{2}n + \cancel{\lg n} \implies \mathbf{n}$$

$$n^2 + \cancel{\lg n} \implies \mathbf{n^2}$$

$$\cancel{4}n^2 + \cancel{256n} \implies \mathbf{n^2}$$

$$n^3 + \cancel{n^2} + \cancel{\lg n} + \cancel{25} \implies \mathbf{n^3}$$

128

DROP COEFFICIENTS & LOWER-ORDER TERMS

$$\cancel{5}n + \cancel{20} \implies n$$

$$\cancel{2}n + \cancel{\lg n} \implies n$$

$$n^2 + \cancel{\lg n} \implies n^2$$

$$\cancel{4}n^2 + \cancel{256n} \implies n^2$$

$$n^3 + \cancel{n^2} + \cancel{\lg n} + \cancel{25} \implies n^3$$

$$128 \implies 1$$

IDENTIFY THE UPPER BOUND

- (1) Count the steps
- (2) Drop the coefficients & lower-order terms
- (3) Identify the upper-bound
  - This is the “worst-case analysis” we talked through last lecture

Which “class” does the algorithm fall in (NB: not class/object from last week, but which “category”)?

O(1)	**O(n lg n)**
O(lg n)	**O(n^2)**
O(n)	**O(n^k) (for a constant k)**

EXAMPLE

FUN(A):

$result = A[1] + 5$

return $result$

EXAMPLE

$\textsc{Fun}(A)$:

$result = A[1] + 5$ `# 1 step`

return *result* `# 1 step`

(1) Number of steps: 2
(2) Drop coefficients and lower order terms: 2 ==> 1
(3) Upper-bound: $O(1)$

EXAMPLE 2

FUN(A):

$sum = 0$

for $i = 1$ to $A.length$

$sum = sum + A[i]$

return sum

EXAMPLE 2

```
FUN(A):
    sum = 0                     # 1 step
    for i = 1 to A.length       # n steps
        sum = sum + A[i]        # 1 step
    return sum                  # 1 step
```

(1) Number of steps: $1 + n(1) + 1 = n + 2$
(2) Drop coefficients and lower order terms: n
(3) Upper-bound: $\boldsymbol{O(n)}$

EXAMPLE 3

FUN(A)

for $i = 1$ **to** $A.length$

  for $j = i$ **to** $A.length$

    if $i == j$

        continue

    else if $A[i] == A[j]$

      return *true*

return *false*

EXAMPLE 3

```
FUN(A)
    for i = 1 to A.length              # n steps
        for j = i to A.length          # n steps
            if i == j                  # 1 step
                continue               # 1 step
            else if A[i] == A[j]       # 1 step
                return true            # 1 step
    return false                       # 1 step
```

(1) Number of steps: $n(n(1+1)) + 1 = 2n^2 + 1$
(2) Drop coefficients and lower order terms: n^2
(3) Upper-bound: $\mathbf{O(n^2)}$

EXAMPLE 4: RECURSION

```
FUN(A)
    if A.length == 1
        return A[1]
    else
        return A[1] + FUN(A[2...n])
```

EXAMPLE 4: RECURSION

```
FUN(A)
    if A.length == 1                  # 1 step
        return A[1]                   # 1 step
    else                              # 1 step
        return  A[1] + FUN(A[2...n])  # 1 + FUN(n-1)
```

Recursive algorithms can be more challenging to estimate.
Formally: use a recursion tree, Master method, or substitution

(covered in Algo or discrete)

EXAMPLE 4: RECURSION

```
FUN(A)
    if A.length == 1                      # 1 step
        return A[1]                       # 1 step
    else                                  # 1 step
        return  A[1] + FUN(A[2...n])      # 1 + FUN(n-1)
```

Informally:

- we make the list smaller by one each time
- we eventually look at every element

EXAMPLE 4: RECURSION

```
FUN(A)
    if A.length == 1                    # 1 step
        return A[1]                     # 1 step
    else                                # 1 step
        return  A[1] + FUN(A[2...n])    # 1 + FUN(n-1)
```

Informally:

- we make the list smaller by one each time
- we eventually look at every element

***O*(*n*)**

SEARCHING: LINEAR

```
FUN(A, ITEM):
for i = 1 to A.length          # n steps
    if A[i] == item            # 1 step
        return i               # 1 step
return -1                      # 1 step
```

$O(n)$

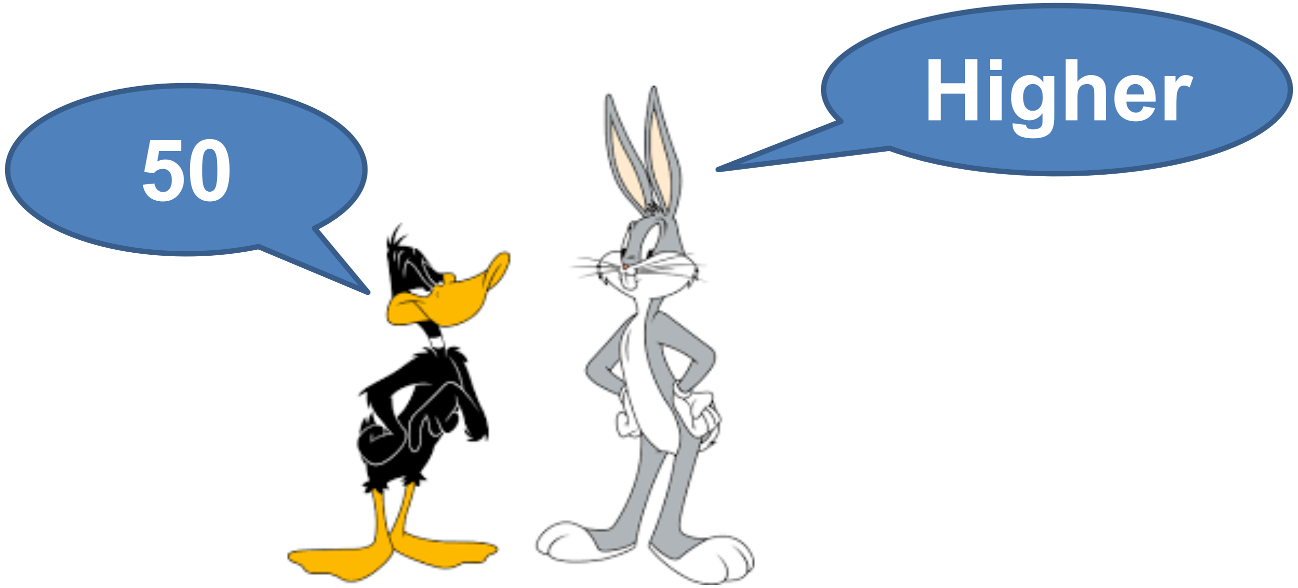
BACK TO THE QUESTION: CAN WE DO BETTER?

```
Fun(A, item):  #  A is either sorted or unsorted
for i = 1 to A.length      # n steps
    if A[i] == item        # 1 step
        return i           # 1 step
return -1                  # 1 step
```

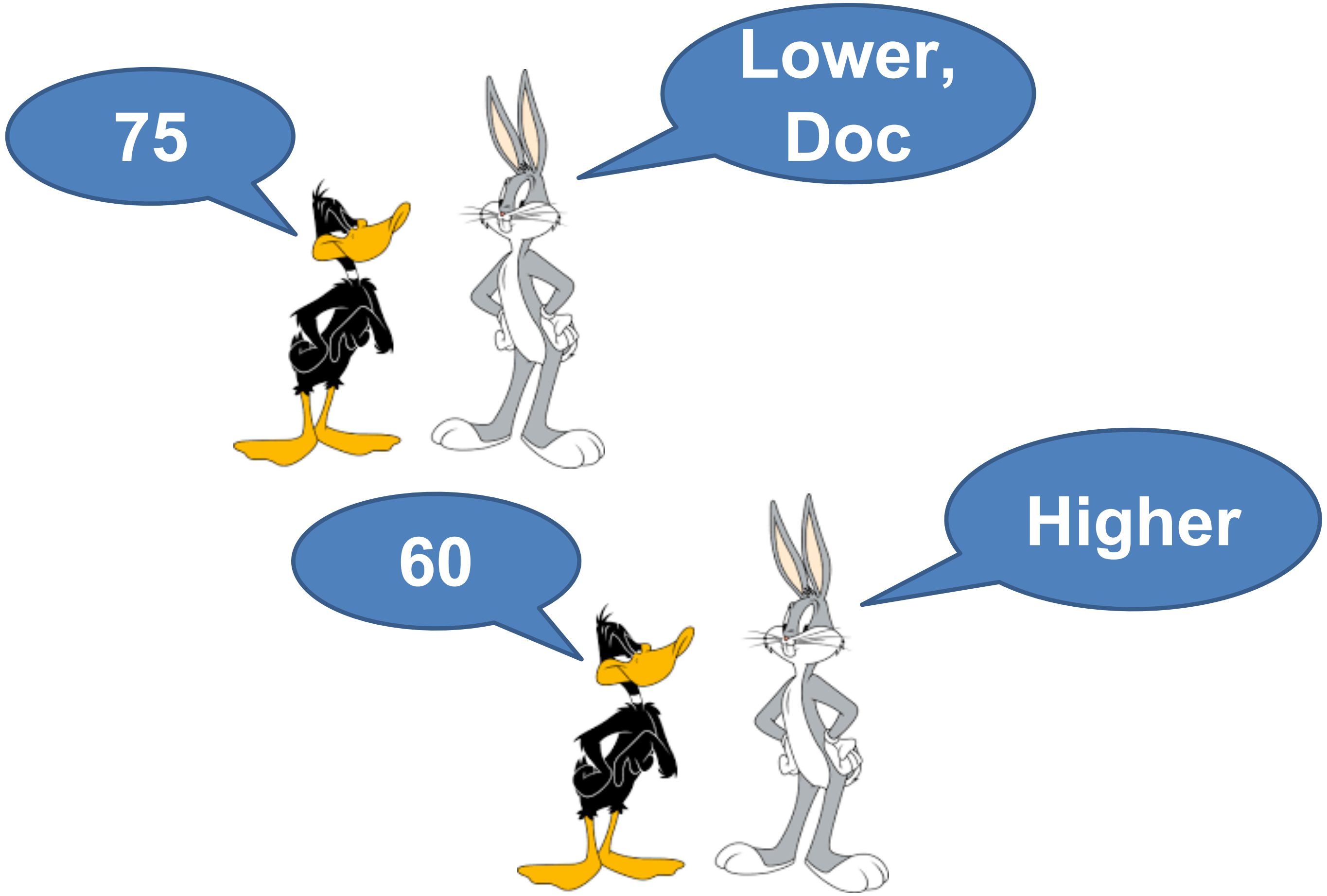
$O(n)$

CAN WE DO BETTER?

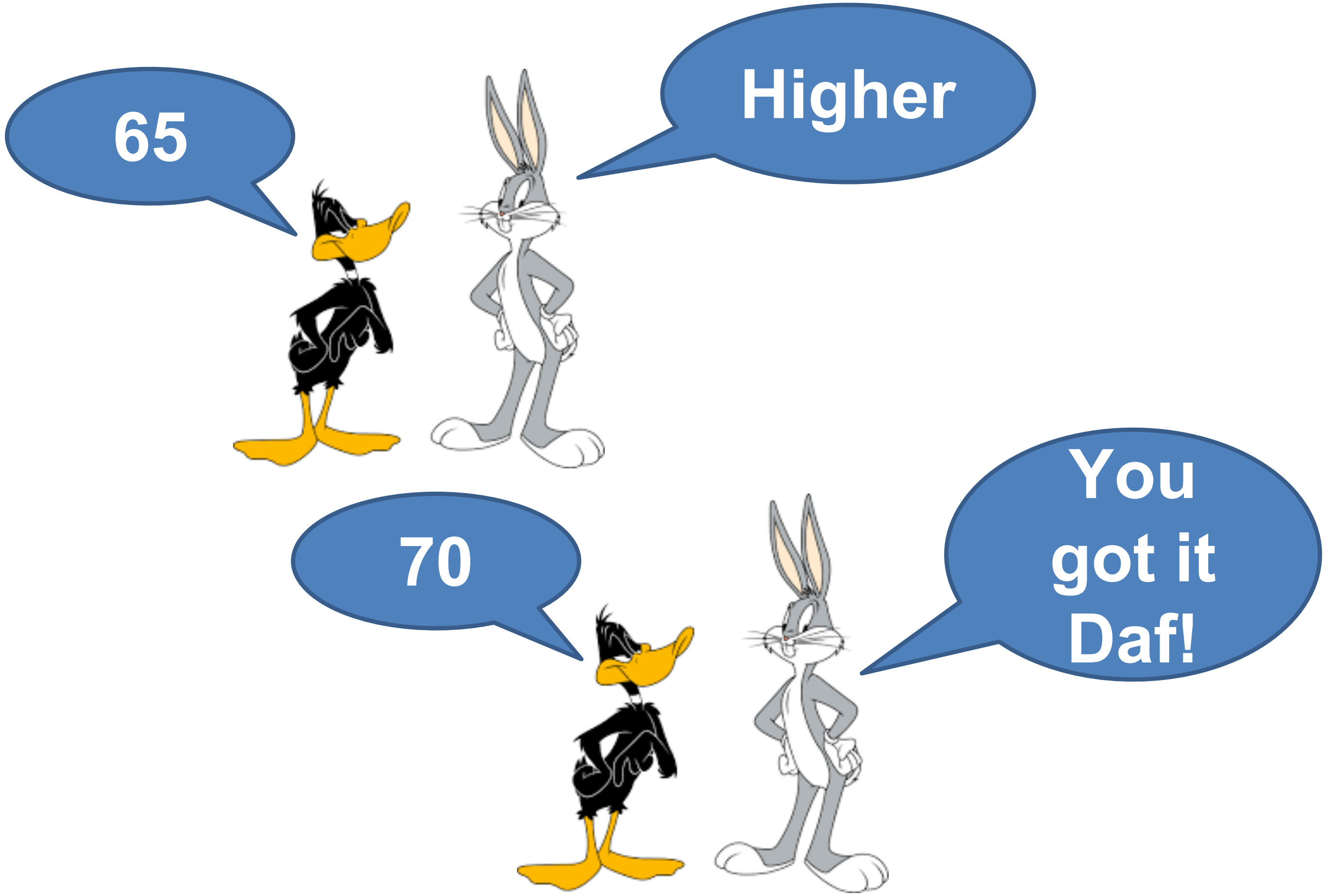
- What if we played a ”guess the number” game with the computer?
- High/Low until we find the number
- Pick a number between 1 and 100



FASTER THAN LINEAR SEARCH



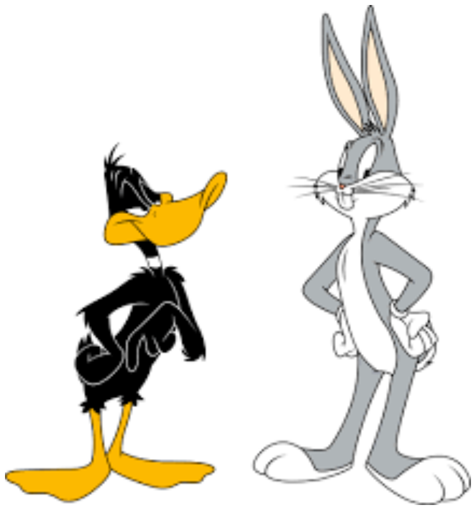
YES! IF CERTAIN CONDITIONS ARE MET



BINARY SEARCH

- Informally, Daffy did a search like a binary search
  - Daffy "triangulated" on the answer by reducing his "search space" and eliminating a large number of possibilities
  - Was able to do this because sequence of numbers was ordered/sorted

Daffy's Guesses 50 75 60 65 70



BINARY SEARCH - OLD SCHOOL PHONE BOOK

- If you've seen one of these, you probably used it to sit on!
- Before whitepages.com – physical book
  - Sorted alphabetically by last name
  - Linear search would be untenable. In practice, people use "approximate target" then something similar to binary search

BINARY SEARCH

- Is similar to Daffy's approach, but systematically reduces the search space by ½ each time
- Binary search is much faster than linear
- *Requires collection to be sorted

Number of items remaining to examine (worst case)

100 → 50 → 25 → 13 → 7 → 4 → 2 → 1

BINARY SEARCH

```
Fun(A, item):  #  Require: A sorted
high = len(A), low = 0        # 1 step
while low <= high             # (log2 n) steps
    mid = (high+low)/2        # 1 step
     if A[mid] == item        # 1 step
         return mid           # 1 step
     if A[mid] > item         # 1 step
         high = mid -1        # 1 step
     else                     # 1 step
         low = mid + 1        # 1 step

return -1                     # 1 step
```

NB: n, n/2, (n/2)/2, ((n/2)/2), …n/(2**k) -> $\log_2(n)$

$O(\log n)$

SORTING

- We’ve just covered a really, really fast algorithm for searching (binary search)
- Wahoo! Great!
- We can have fast look up and retrieval.
- …But how do we get set up to use binary search since it requires our collection to be sorted?

SORTING

- Putting things in order

SORTING

- There are a number of sorting algorithms that we can use
- Some algorithms are more efficient than others
- Some algorithms are less efficient but easier to implement
- We do algorithmic analysis to find the “best fit” for what we want to do

SORTING: FIRST CONCEPTS

- Even if we develop an amazing sorting algorithm, we know we can’t do better than O(n)
  - Why? In order to sort a collection we must look at every element in the collection

SORTING: FIRST CONCEPTS

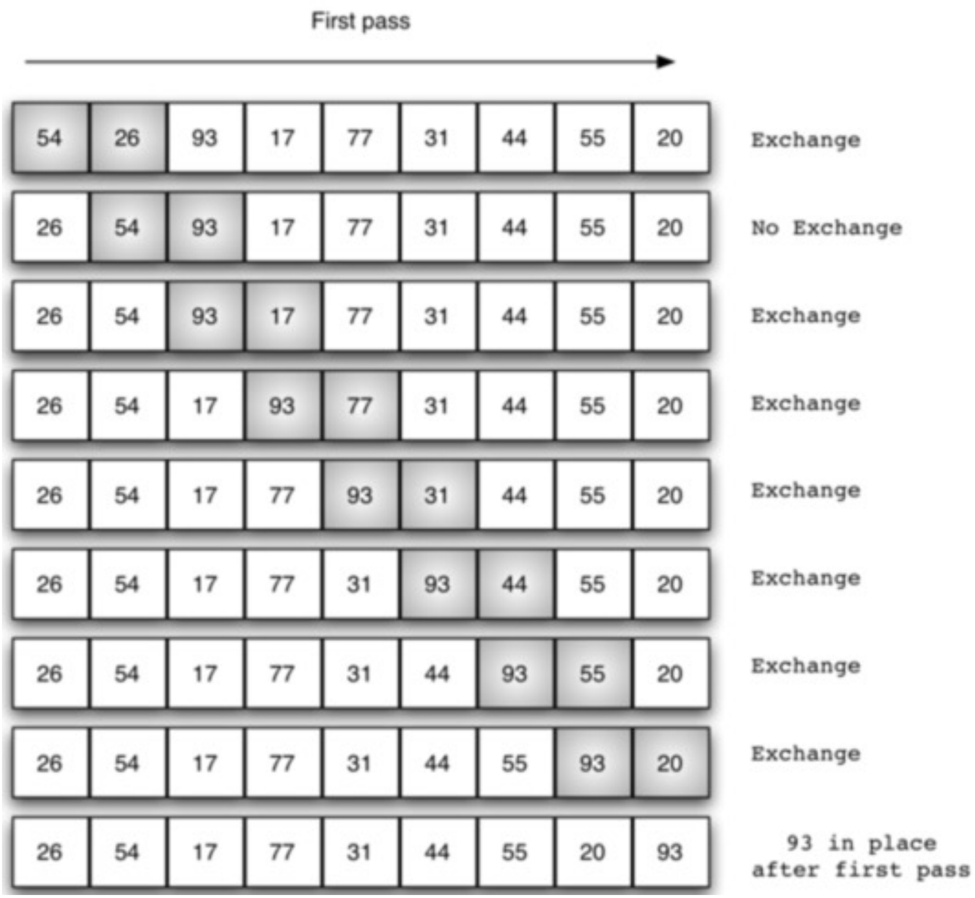
- If we need to sort often (due to a changing collection) we may opt to back off and settle for linear search
  - Why? Because if we need to sort frequently, that adds “overhead” to our sort/search pair
- If we can sort once (or very infrequently), we can “amortize” the cost of sorting over the many search attempts. Pay once upfront and then reap the benefits for millions of searches

SORTING: BUBBLE SORT

- Binary Search requires a sorted sequence.
  - How do we get there?
- One option: Bubble Sort
  - Make multiple passes through a list.
  - Compare adjacent items and exchanges those that are out of order.
  - Each pass through the list places the next largest value in its proper place
  - Each item “bubbles” up to the location where it belongs.

Content from this section primarily from:
Problem Solving with Algorithms and Data Structures Using Python SECOND EDITION
By Bradley Miller

BUBBLE SORT



Graphics © Bradley Miller

$O(n^2)$

Requires ½ n^2 – n passes

Pass	Comparisons
1	$n-1$
2	$n-2$
3	$n-3$
...	...
$n-1$	1

```python
def bubble_sort(my_list):
    for passnum in range(len(my_list)-1,0,-1):
        for i in range(passnum):
            if my_list[i]>my_list[i+1]:
                temp = my_list[i] # could have used Python tuple assignment for swap
                my_list[i] = my_list[i+1]
                my_list[i+1] = temp
```

Pro: Bubble Sort is straightforward
Con: $O(n^2)$ isn't great performance

BUBBLE SORT: OBSERVATIONS

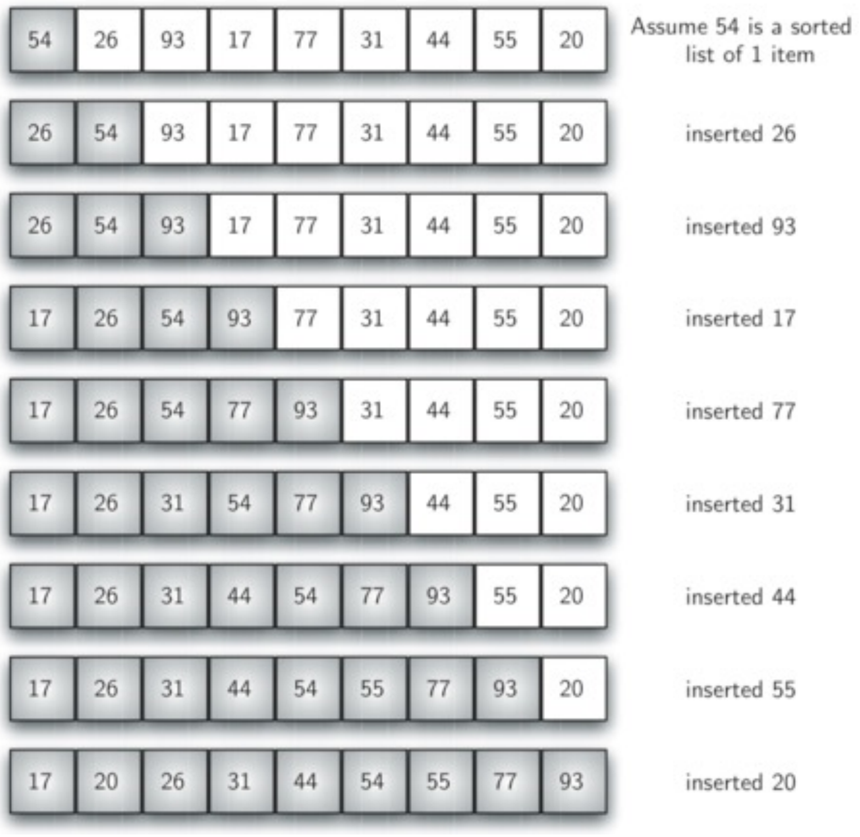
- At $O(n^2)$, Bubble Sort is not efficient. For small n, it's not horrible, but it's not good either
- For "benchmark" comparisons, there are other $O(n^2)$ algorithms that perform better than Bubble Sort
  - Notice how we do a complete shift of everything on each iteration? Some other algorithms in this same class do not do that, hence have better benchmark scores

- On the other hand, Bubble Sort is easy to understand, easy to code (if you need to) and may perform "good enough" for small n to use it for very simple situations

INSERTION SORT

- Manage two logical collections (partition the one physical collection)
  - One sub-collection is sorted and grows as we insert elements from the unsorted sub-collection.
    - Shift the elements greater than the one we're inserting to the right
  - The unsorted sub-collection shrinks as we "remove" elements (actually shift them across the sorted boundary)
  - Start with a 1-element sorted collection

INSERTION SORT



$O(n^2)$

```python
def insertion_sort(alist):
    for index in range(1,len(alist)):
        current_value = alist[index]
        position = index

        # if element at current position > what we want to "insert", shift right
        while position > 0 and alist[position-1] > current_value:
            alist[position] = alist[position-1]
            position = position-1

        # fill the "open" slot with the value we're inserting
        alist[position] = current_value

```

Pro: Less swaps than Bubble Sort
Con: Still $O(n^2)$

Graphics © Bradley Miller

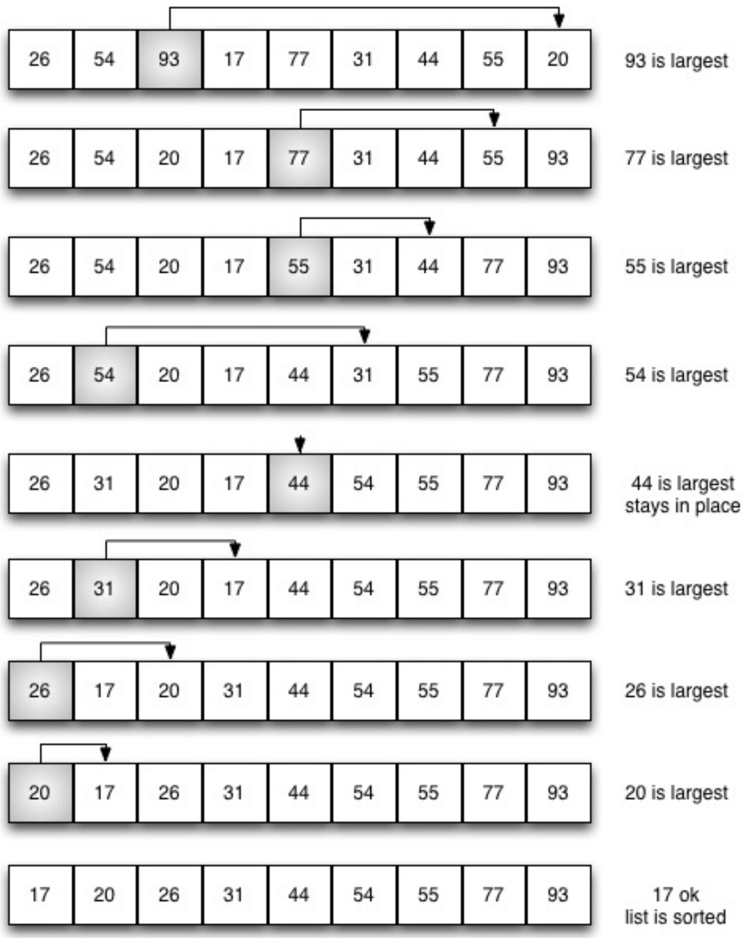
GENERAL OBSERVATIONS

- Oh – another thing:
- We're covering algorithmic analysis, and some of your future courses may have you work through the process of implementing some of them, so you understand how things work
- In many cases, however, these algorithms have already been implemented. In an industry job, it's more likely that you'll simply select the algorithm you want rather than re-inventing the wheel
  - Tons of code exists that implements many of these
  - Every once-in-a-while, you'll need to hand-craft these, but more than likely you'll reuse existing code

SELECTION SORT

- Selection Sort is similar to Bubble Sort, but does fewer swaps
  - Make multiple passes through a list.
  - Look for the largest value as it makes a pass. Place the largest value (for this pass) in the proper location

SELECTION SORT



Graphics © Bradley Miller

$O(n^2)$

```python
def selection_sort(alist):
   for fillslot in range(len(alist)-1,0,-1):
       position_of_max=0
       for location in range(1,fillslot+1):
           if alist[location]>alist[position_of_max]:
               position_of_max = location
    # use tuple assignment to swap the values
       (alist[fillslot], alist[position_of_max]) = (alist[position_of_max], alist[fillslot])
```

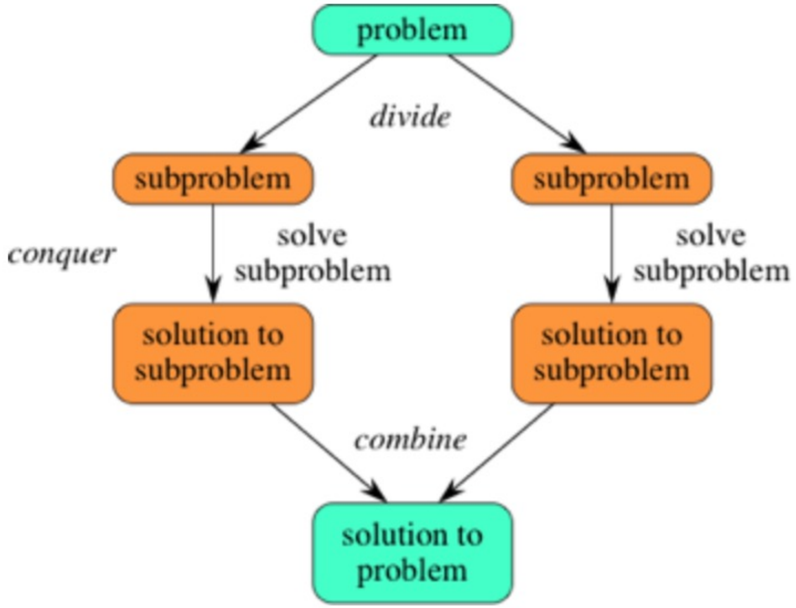
Pro: Less swaps than Bubble Sort
Con: Still $O(n^2)$

SORTING: CAN WE DO BETTER?

- Iterative sort is easy to understand and (relatively) easy to code, but can we do better than $O(n^2)$?
- Yes, if we change our approach
- Enter “divide and conquer” algorithms

DIVIDE AND CONQUER

- Algorithm that subdivides the problem into smaller subproblems, solves the subproblems and then recombines.
- Sounds a lot like recursion, right?
- That's because it is. Divide and Conquer algorithms use recursion to solve the problem.



MERGE SORT

- Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item -> Base Case

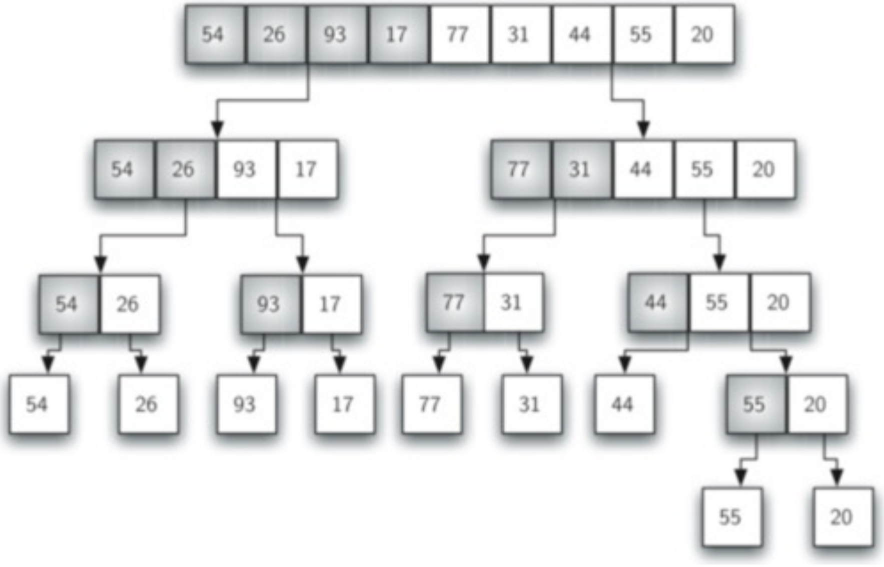
Approach:

Divide: iteratively splitting a list into two sublists of equal length until each sublist has one element.

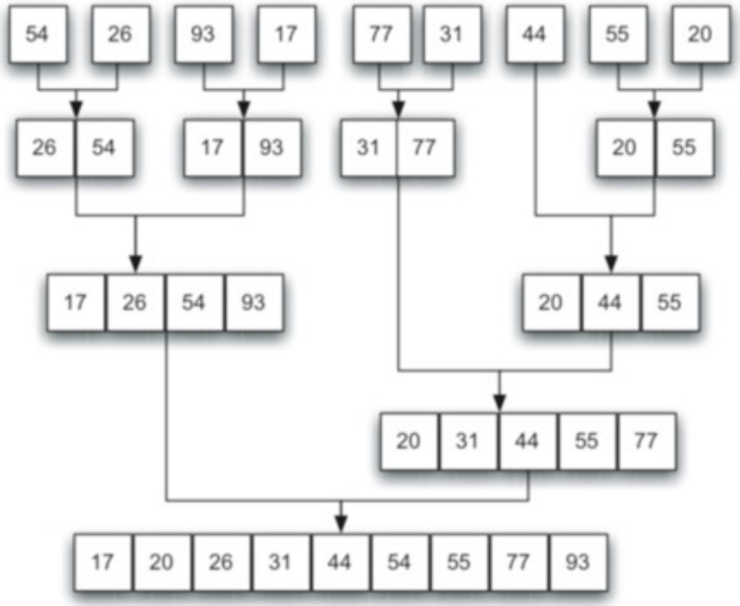
Conquer and Combine:
A pair of sublists is successively merged into a list with the elements in increasing order.
The process ends when all the sublists have been merged.

MERGE SORT

Divide



O(n log n)

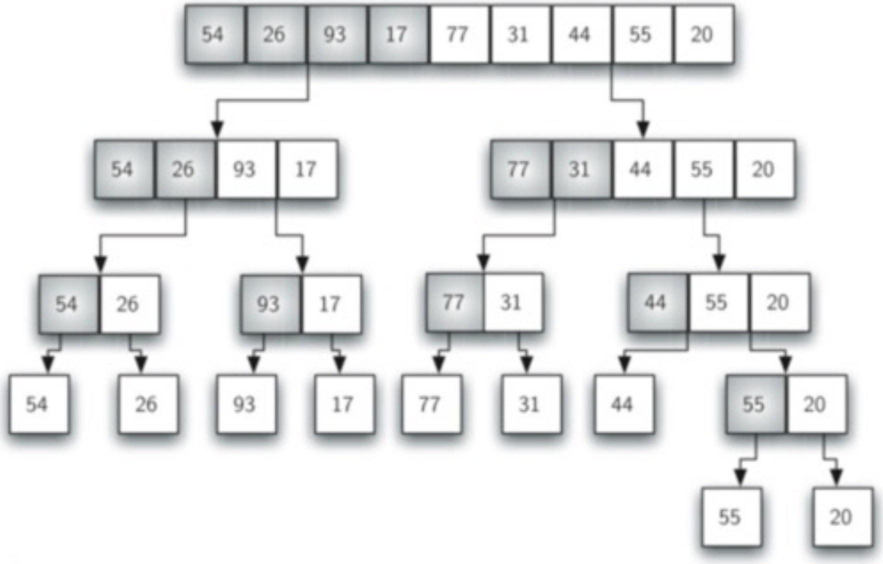


Merge & Conquer

Graphics © Bradley Miller

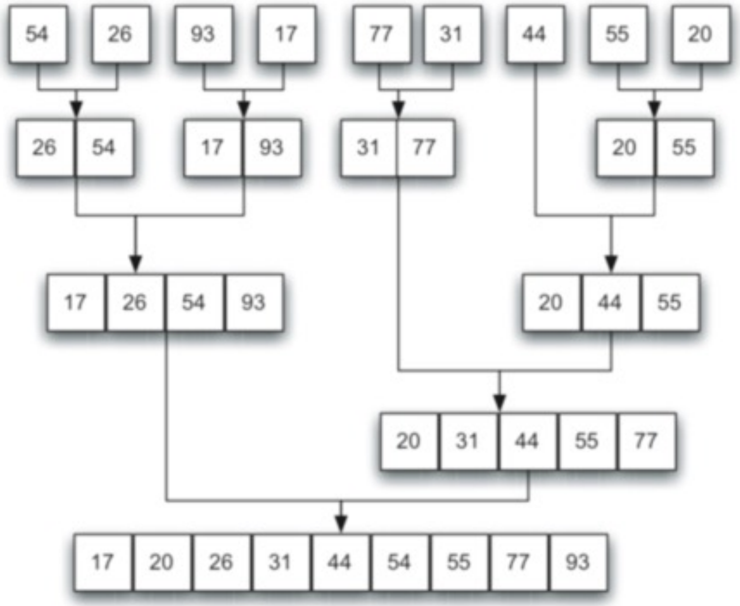
MERGE SORT

Divide



O(n log n)

```
def merge_sort(lst):
    print("Splitting ",lst)
    if len(lst) <= 1:
        print('Hit base case!')
    if len(lst)>1: # while not base case (size 1 or 0)
        mid = len(lst)//2  # remember binary search? log(n) to split
        left_half = lst[:mid] # use Python slices to split in half
        right_half = lst[mid:]

        merge_sort(left_half) # recursively call ourself with each half
        merge_sort(right_half)
```



Merge & Conquer

Graphics © Bradley Miller

MERGE SORT

- Merge Sort is fast!
- The fastest comparison-based sorting algorithms have **O(n log n)** time complexity
- But **it requires extra space (2x)** to hold the interim values while dividing and merging back to original collection
- Space requirements can be significant for large data

- *Note: We often look at time complexity, but space requirements are also important. Sometimes the trade-off for algorithms is time vs. space!*

QUICK SORT

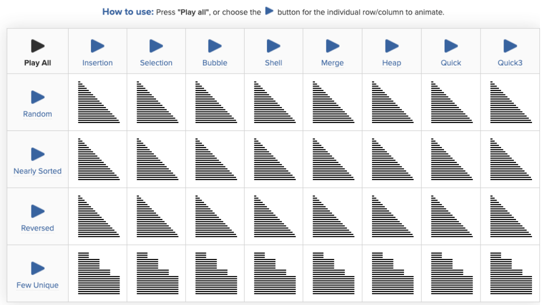
- Quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage.

- Trade-off, it is possible that the list may not be divided in half. When this happens, performance degrades.

  - **O(n log n)** without Merge Sort's extra space requirements
  - **O(n2)** worst case if collection cannot be divided evenly

OTHERS

- Selection Sort
- Shell Sort
- Great news: Most libraries & frameworks have written these already so you can USE them without needing to REWRITE them
  - You should still understand how they work & the consequences
- You’ll learn more about these in Algo next term!
- Let’s watch some graphically:
  - https://www.toptal.com/developers/sorting-algorithms



FINAL EXAM NOTES

- I **will not** ask you to implement these on an exam. But I will ask conceptual questions and expect you to know about efficiency when selecting searching & sorting algorithms

FINAL EXAM

- Cumulative, but skewed towards material since the midterm
- Of course, many topics "build" on each other, but make sure you review:
  - Dictionaries
  - Classes & Objects (including Functions as Objects)
  - Exception Handling
  - Files
  - Recursion (small functions)
  - Big-Oh (conceptual, and possibly "write a function that conforms to…")

FORMAT: SIMILAR TO MIDTERM EXAM

- Short Answer / Multiple Choice
- Understand / Trace the code
- Write some code (snippets, functions, classes)

FORMAT: SIMILAR TO THE EXAM 1

Fill in the blank

Use a dictionary to map X

What does the file contain?

What's the upper bound?

What is the Output of the given Code?

Write a function that takes as a parameter another function that does X

Write a class that passes the given tests

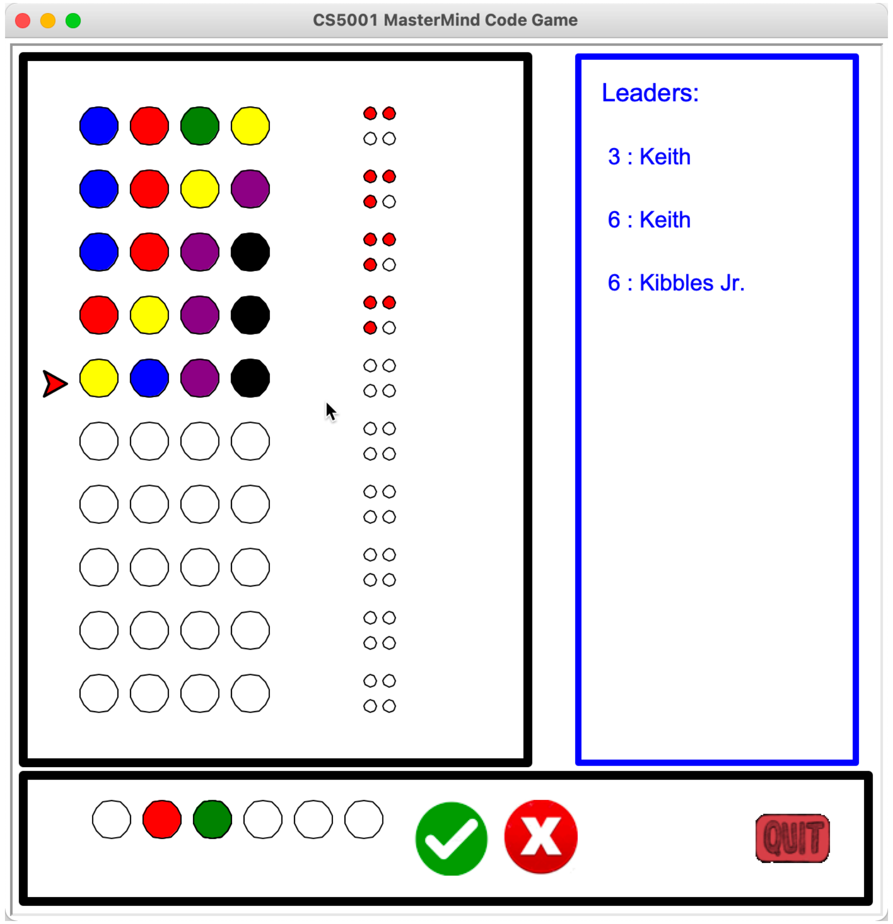
Write a function that does X and has an upper bound of O(…)

I haven't written the Final Exam yet, but here the *style* of my exams is the same between Midterm and Final

LOOK HOW FAR YOU'VE COME IN 3 MONTHS!

```python
def main():
    amount = int(input("Welcome to PDQ Bank! How much to withdraw? $ "))

    old_amount = amount # save the old amount for our final message to user

    num_fifties = amount // FIFTY
    amount -= num_fifties * FIFTY

    num_twenties = amount // TWENTY
    amount -= num_twenties * TWENTY

    num_tens = amount // TEN
    amount -= num_tens * TEN

    num_fives = amount // FIVE
    amount -= num_fives * FIVE

    num_ones = amount

    print("Cha-ching!\nYou asked for $", old_amount)
    print("That breaks down to:\n",
          num_fifties, "fifties\n",
```

Wow!



Where you started…

Where you ended

AS OSCAR & FELIX WOULD SAY

Yay Yay!

Proud Family
Disney+

Q & A

- Questions?

THANKS!

- Stay safe, be encouraged, & see you next week!