Announcements

- Assignment 2 due August 3rd
  - Office hours:
    - Friday: 1 - 5PM and 6 - 10PM (BA7172)
- Exercise 9 and Lab 9 are up
  - Exercise 8 is due tonight at 11PM
- T2 scans/A1 remarks tomorrow (?)
- Course evaluations are up
Assignment 2 + PythonTA

- I'll ignore the "too many nested block", "too many local variables" and "too many statements" errors when marking
  - Worry about the other parts instead
Outline

- Efficiency
- Sorting Algorithms
Last Week

- Runtime complexity (efficiency) of Binary Search Trees
  - Searching in them is fast!
  - Relative to the height of the BST itself.
Runtime Complexity

"How does the runtime change as our input grows?"
Runtime Complexity

[5, 3, 2, 1, 4]

"How long would calling contains() take (relative to the size of the list):"
"How long would calling contains() take (relative to the size of the list)?"

If we're checking for the value 10:
Look through all items in the list.
Runtime Complexity

[5, 3, 2, 1, 4]

What if our list has 10 items? 20? 100?
Runtime Complexity

\[5, 3, 2, 1, 4\]

What if our list has 10 items? 20? 100?

We would have to check through the entire list in order to say an item's not in it.
Runtime Complexity

\[1, 3, 4, 5, 6, 7, 9\]

What if our list is sorted?
Runtime Complexity

[1, 3, 4, 5, 6, 7, 9]

Is 8 in this list?
Binary Search

[1, 3, 4, 5, 6, 7, 9]

Look at the middle item.
Binary Search

\[1, 3, 4, 5, 6, 7, 9\]

Look at the middle item.

\(8 > 5\) so look to the right of 5.
Binary Search

[1, 3, 4, 5, 6, 7, 9]

Look at the middle item.
Binary Search

[1, 3, 4, 5, 6, 7, 9]

Look at the middle item.
Binary Search

\[1, 3, 4, 5, 6, 7, 9]\n
We only have 1 item left.

Nothing else to look at!
Runtime Complexity

\[ [1, 3, 4, 5, 6, 7, 9] \]

We only looked at 3 items!

Much better than looking through the whole list!
Binary Search

- Halve the number of items we have to look at until we reach an empty list or a list with only 1 item
Binary Search

- Look at 1 item when the size is n
- Look at 1 item when the size is n / 2
- Look at 1 item when the size is n / 4
- Look at 1 item when the size is n / 8
- ...
- Look at 1 item when the size is 1
Binary Search

- Look at 1 item when the size is $n$
- Look at 1 item when the size is $n/2$
- Look at 1 item when the size is $n/4$
- Look at 1 item when the size is $n/8$
- ... 
- Look at 1 item when the size is 1

How many items do we look at in total?

How many times can we halve $n$ before it's $\leq 1$?
Halving n

- We can halve $n \log(n)$ times
  - $\log(n) = \log_2(n)$
Halving $n$

- Math! Look for the first number $2^k$ that is $\geq n$

$$2^k \geq n$$
Halving $n$

- Math! Look for the first number $2^k$ that is $\geq n$

$$2^k \geq n$$

$$k \geq \log_2(n)$$
**lg(n)**

- Usually appears when we have to split a problem into even halves.
  - Not every problem that halves will involve $\lg(n)$, but many do.
Contains

- Unsorted list: n items at most
- Sorted list: $\log_2 n$ items at most
Contains

- Unsorted list: n items at most
- Sorted list: \( \log n \) items at most
- BST: height items at most
  - \( \log n \) if balanced, n if unbalanced
Contains

- Unsorted list: n items at most
- Sorted list: \(\log n\) items at most
- BST: height items at most
  - \(\log n\) if balanced, \(n\) if unbalanced
- Tree? LinkedList? Stack? Queue? BinaryTree?
  - \(n\) item for all of these at most
**Big-O**

- We can count "how many lines we have to run"
  - I.e. "The number of steps we take."
Big-O

```python
sum = 0
for i in range(n):
    print(i)
    sum += i

return sum
```
sum = 0
for i in range(n):
    print(i)
    sum += i

return sum
sum = 0
for i in range(n):
    print(i)
    sum += i

return sum
Big-O

```python
sum = 0
for i in range(n):
    print(i)
    sum += i
return sum
```

1 + n * (1 + 1) + 1 steps

Runs n times

1 step * n times

1 step
sum = 0

for i in range(n):
    print(i)
    sum += i

return sum

1 step

Runs n times
1 step * n times
1 step * n times
1 step

2n + 2 steps
Big-O

- $2n + 2$
- As $n$ gets larger, the $+ 2$ has a negligible effect.
- The coefficient $2$ also isn't too important -- it doesn't change.
- Simplify!
Big-O

● 2n + 2 is in O(n)
  ○ O(n): "Order of n"
  ○ "The runtime grows linearly with regards to n."

● We care about how runtime grows in relation to n
Big-O (Formally)

- Upper bound on growth
- If something is in O(n), its runtime is always \( \leq cn + B \) for some \( c \) and \( B \)
  - \( c \) is any coefficient
  - \( B \) is any number
  - Previous example, \( c \) and \( B \) were both 2.
Big-O (Less formal)

● Simpler to understand:
"Runtime grows linearly/proportionally with respect to n"
Orders of Complexity

● O(1): Constant time
  ○ No matter how n changes, the number of steps are the same
Orders of Complexity

● O(1): Constant time
  ○ No matter how n changes, the number of steps are the same

● O(lgn): Logarithmic time
  ○ Double n in order to see an increase in steps taken
Orders of Complexity

- O(n): Linear time
  - Steps taken are proportional to n
Orders of Complexity

- O(n): Linear time
  - Steps taken are proportional to n
- O(n\log n):
  - Almost linear. Easier explained via examples.
Orders of Complexity

- **O(n):** Linear time
  - Steps taken are proportional to n

- **O(nlgn):**
  - Almost linear. Easier explained via examples.

- **O(n^2):** Quadratic time
  - n steps taken for each item in n
Orders of Complexity

● $O(2^n)$: Exponential time
  ○ $n$ grows by 2, the amount of work done doubles
  ○ Minimax for A2 does this!
    ■ $n$ is "the maximum number of attacks we can perform in a game"
Exponential Time: Minimax

Each state can split into 2 at most.

= $2^n$ states in total
**Big-O**

- Simplify and put runtime in terms of \( n \)
  - Keep the fastest growing part
  - Drop everything else (smaller order values, coefficients, etc.)
Big-O

- $7n^2 + 3n + 4$
  - Simplifies to $O(n^2)$
  - $n$ doesn't grow as fast as $n^2$ -- it'll be outshadowed by $n^2$ eventually.
Complexity Order

For non-small $n$s:

$$1 \leq \log n \leq n \leq n \log n \leq n^2 \leq 2^n$$
Complexity Order

For non-small ns:

\[ 1 \leq \log_8 n \leq 8 \leq 8 \log_8 n \leq 8^2 \leq 2^8 \]

Plug 8 in for n!
Complexity Order

For non-small ns:

1 <= 3 <= 8 <= 24 <= 64 <= 256
Growth Rate
Growth Rate

The graph shows the growth rate for different functions as a function of n. The functions represented include:

- $O(1)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^n)$
- $O(n^{n+2})$

The x-axis represents the variable n, and the y-axis represents the runtime. The graph illustrates how each function's runtime increases as n grows.
Counting Steps

sum = 0
for i in range(n):
    print(i)
    for j in range(n):
        print(i * j)
        sum += i * j

return sum
Counting Steps

```python
sum = 0  # 1 step
for i in range(n):
    print(i)  # 1 step
    for j in range(n):
        print(i * j)  # 1 step
        sum += i * j  # 1 step

return sum  # 1 step
```
Counting Steps

sum = 0 1 step
for i in range(n): Runs n times
    print(i) 1 step * n
    for j in range(n):
        print(i * j) 1 step * n
        sum += i * j 1 step * n

return sum 1 step
Counting Steps

\[
\text{sum} = 0 \\
\text{for } i \text{ in range}(n): \text{ Runs n times} \\
\quad \text{print}(i) \text{ 1 step * n} \\
\text{for } j \text{ in range}(n): \text{ Runs n times} \\
\quad \text{print}(i * j) \text{ 1 step * n * n} \\
\quad \text{sum += i * j} \text{ 1 step * n * n} \\
\text{return sum} \text{ 1 step}
\]
Counting Steps

sum = 0  1 step
for i in range(n):
    print(i)   Runs n times
    for j in range(n):
        print(i * j)  Runs n^2 times
        sum += i * j  Runs n^2 times

return sum  1 step
Counting Steps

sum = 0  # 1 step
for i in range(n):
    print(i)  # Runs n times
    for j in range(n):
        print(i * j)  # Runs n^2 times
        j

sum += i * j  # Runs n^2 times

return sum  # 1 step
Counting Steps

```
sum = 0  
for i in range(n):
    print(i)  
    for j in range(n):
        print(i * j)
    sum += i * j
return sum
```

1 step

Runs n times

2n^2 + n + 1

Runs n^2 times

Runs n^2 times

1 step

59
Counting Steps

sum = 0
for i in range(n):
    print(i)
    for j in range(n):
        print(i * j)
        j
        j
    sum += i * j
return sum
Counting Steps

def find_max(t):
    if not t.right:
        return t.value

    return find_max(t.right)
def find_max(t):
    if not t.right:
        return t.value
    return find_max(t.right)

Counting Steps

How many times?
1 step
1 step
1 step
Counting Steps

Count the longest path of 'right subtrees' from root.
def find_max(t):
    if not t.right:
        return t.value
    return find_max(t.right)

At most: height of t
1 step * height
1 step* height
1 step * height
Counting Steps

def find_max(t):
    At most: height of t
    if not t.right:
        O(height)
    1 step * height
    O(lgn) for balanced BST
    1 step* height
    O(n) for unbalanced
    1 step * height
Finding Runtime from Code

- Count manually
- Or: Consider how many items you need to look at in relation to n
- Practice!
Finding Runtime from Times

- Time how long it takes to run code

<table>
<thead>
<tr>
<th>n</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
</tbody>
</table>
Finding Runtime from Times

Linear! $O(n)$ runtime
Given this graph, what runtime complexity fits best?

A) $O(1)$  
B) $O(lgn)$  
C) $O(n)$  
D) $O(n^2)$
Answer:

D) $O(n^2)$
Given this graph, what runtime complexity fits best?

A) O(1)    B) O(lgn)    C) O(n)    D) O(n^2)

Of these choices, n^2 is the only one that curves upwards.
Given these times, what runtime complexity fits best?

<table>
<thead>
<tr>
<th>n</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.0</td>
</tr>
<tr>
<td>10</td>
<td>5.0</td>
</tr>
<tr>
<td>100</td>
<td>5.0</td>
</tr>
<tr>
<td>1000</td>
<td>5.0</td>
</tr>
</tbody>
</table>

A) O(1)  B) O(lgn)  C) O(n)  D) O(n^2)
Answer:

A) O(1)
Break!
For 10 minutes. :)
Sorting Algorithms

- Bubble Sort
- Insertion Sort
- Selection Sort
Selection Sort

Split list into sorted and unsorted parts. Sorted is initially empty.

5 3 1 9 8 4

Sorted  Unsorted
Selection Sort

Find the smallest number in our unsorted part.

Sorted  Unsorted

5  3  1  9  8  4
Selection Sort

Extend the sorted area by 1.

Sorted  Unsorted
**Selection Sort**

Swap that item with the smallest item found.

Sorted  Unsorted
Selection Sort

Find the smallest number in our unsorted part.

Sorted  |  Unsorted
1       | 3       | 5       | 9       | 8       | 4
Selection Sort

Extend the sorted area by 1.

Sorted 1 3 5 9 8 4 Unsorted
Selection Sort

Don't need to swap 3 with anything.

Sorted  Unsorted
Selection Sort

Find the smallest number in our unsorted part.

Sorted 1 3 5 9 8 4 Unsorted
Selection Sort

Extend the sorted area by 1.

1 3 5 9 8 4

Sorted  Unsorted
Selection Sort

Swap with the smallest number.

1 3 4 9 8 5

Sorted  Unsorted
Selection Sort

Repeat, swapping 9 with 5.

1 3 4 5 8 9

Sorted  Unsorted
Selection Sort

8 stays in the same place.
Selection Sort

9 stays in the same place.

Sorted | Unsorted

1 | 3 | 4 | 5 | 8 | 9
Selection Sort

- Algorithm is roughly:
  for i in range(n):
      Find the smallest number in lst[i:]
      Swap lst[i] with the smallest.
Selection Sort

Looking for the smallest number means looking through the entire unsorted part.

\[[5, 3, 1, 9, 8, 4]\] Look at all 6 items
\[[1, 3, 5, 9, 8, 4]\] Look at 5 unsorted items
\[[1, 3, 5, 9, 8, 4]\] Look at 4 unsorted items
\[[1, 3, 4, 9, 8, 5]\] Look at 3 unsorted items
Selection Sort

- That means there are
  \[ n + (n - 1) + (n - 2) + \ldots + 1 \]
  Steps for 'searching for the smallest' in total

- This is in \( O(n^2) \)
  - Math proof for that; I'm not getting into it.
Selection Sort

- Algorithm is roughly:

  ```python
  for i in range(n):
      Find the smallest number in lst[i:]
      Swap lst[i] with the smallest.
  ```

  Runs n times.
Selection Sort

- Algorithm is roughly:
  
  ```python
  for i in range(n):
    Find the smallest number in lst[i:]
    Swap lst[i] with the smallest.
  ```

  Takes O(n) steps.

  Runs n times.
Selection Sort

- Runtime is at most $n \times n = O(n^2)$ steps
  - Big-O is an upper bound! We can simplify things a bit if we'd like!
- Insertion + Bubble Sort are also $O(n^2)$ algorithms
- Can we do better than $O(n^2)$ when sorting?
Merge Sort

Split it into 2 halves.

5 3 1 8

6 4 7 2
Merge Sort

Sort the 2 halves (recursively).

1 3 5 8

2 4 6 7
Merge Sort

Merge these lists together.

1 3 5 8

2 4 6 7
Merge Sort

Merge these lists together.

1 3 5 8
2 4 6 7

1
Merge Sort

Merge these lists together.

1 3 5 8

1 2

2 4 6 7
Merge Sort

Merge these lists together.

1 3 5 8
2 4 6 7
1 2 3
Merge Sort

Merge these lists together.

1 3 5 8

2 4 6 7

1 2 3 4
Merge Sort

Merge these lists together.

1 3 5 8 2 4 6 7

1 2 3 4 5
Merge Sort

Merge these lists together.

1 3 5 8
2 4 6 7
1 2 3 4 5 6
Merge Sort

Merge these lists together.

1 3 5 8

2 4 6 7

1 2 3 4 5 6 7
Merge Sort

Merge these lists together.

1 3 5 8
2 4 6 7
1 2 3 4 5 6 7 8
Merge Sort

- Merging 2 sorted lists takes $O(n)$ time
  - Where $n$ is the sum of the size of both lists
  - We only have to look at each item once!
Merge Sort

- Calling mergesort recursively means we're writing a recursive function
  - Base case?
Merge Sort

Suppose we keep splitting our list:

5  3  1  8  6  4  7  2
Merge Sort

Suppose we keep splitting our list:

5 3
1 8
Merge Sort

Suppose we keep splitting our list:

5  3
Merge Sort

A list with 0 or 1 elements is already sorted!

5
Merge Sort

def mergesort(lst):
    if len(lst) <= 1:
        return(lst)
def mergesort(lst):
    if len(lst) <= 1:
        return(lst)

    ● Split lst in half
    ● Sort each half
    ● Merge the 2 halves
def mergesort(lst):
    if len(lst) <= 1:
        return(lst)
    left = lst[:len(lst)//2]
    right = lst[len(lst)//2:]
    ● Sort each half
    ● Merge the 2 halves
def mergesort(lst):
    if len(lst) <= 1:
        return(lst)
    left = lst[:len(lst)//2]
    right = lst[:len(lst)//2]
    sorted_left = mergesort(left)
    sorted_right = mergesort(right)
    ● Merge the 2 halves
sorted_left = mergesort(left)
sorted_right = mergesort(right)
new_list = []
left_index = 0
right_index = 0
Merge Sort

```python
ew_list = []
left_index = 0
right_index = 0
while left_index < len(sorted_left)
    and right_index < len(sorted_right):
```
while left_index < len(sorted_left) and right_index < len(sorted_right):
    if sorted_left[left_index] < sorted_right[right_index]:
        new_list.append(sorted_left[left_index])
        left_index += 1
while left_index < len(sorted_left) and right_index < len(sorted_right):
    if sorted_left[left_index] < sorted_right[right_index]:
        new_list.append(sorted_left[left_index])
        left_index += 1
    else:
        new_list.append(sorted_right[right_index])
        right_index += 1
while left_index < len(sorted_left) and right_index < len(sorted_right):
    if sorted_left[left_index] < sorted_right[right_index]:
        new_list.append(sorted_left[left_index])
        left_index += 1
    else:
        new_list.append(sorted_right[right_index])
        right_index += 1
return new_list + sorted_left[left_index:] + sorted_right[right_index:]
Merge Sort

- Many different approaches to merging
  - Removing from the front until they're both empty is one approach
- If you take advantage of the fact that the list are sorted, the runtime will be $O(n)$. 
Merge Sort Runtime Complexity

1. Break list into 2 halves
2. Sort each half recursively
3. Merge the halves
Merge Sort Runtime Complexity

1. Break list into 2 halves   \( O(n) \)
2. Sort each half recursively   
3. Merge the halves   \( O(n) \)

Runtime is \( O(n) \) * the number of recursive calls we make.
Merge Sort

Consider the number of steps at each division.

5 3 1 8 6 4 7 2
Merge Sort

Consider the number of steps at each division.

```
  5  3  1  8  6  4  7  2
  5  3  1  8  6  4  7  2
```
Merge Sort

Consider the number of steps at each division.
Merge Sort

Consider the number of steps at each division.

5 3 1 8 6 4 7 2

5 3 1 8 6 4 7 2

5 3 1 8 6 4 7 2

5 3 1 8 6 4 7 2

5 3 1 8 6 4 7 2
Merge Sort

At most: 4 levels of recursion.
Merge Sort

1 step at each case in the bottom level.
Merge Sort

Merging takes 2 steps (8 total in that level)
Merge Sort

Merging takes 4 steps (8 total in that level)
Merge Sort

Merging takes 8 steps in our initial call.

```
 5  3  1  8  6  4  7  2
 5  3  4  1  8   6  4  7  2
 5  2  3  1  2  8   6  2  4  7  2
 1  1  1  1  1  1  1  1  1  1
```
Merge Sort

- $O(n)$ steps at each level
- $\lg n$ levels in total
  - Because we keep halving it as much as we can.
- Runtime: $O(n) \times \lg n = O(n \lg n)$
Recursive Runtime Complexity

- Usually you can split it into levels of recursive calls
  - How many levels are there at most?
  - How many steps do you take at each level?
  - Runtime complexity is the product of those 2.
Binary Search

- Ig£ levels at most
  - Kept halving the list we had to look through!
- Looked at only 1 item per level
- Ig£ * O(1) = O(lgn)
Quick Sort

- Split list into 3 parts
  - One containing all values < something
  - One containing all values == something
  - One containing all values > something
- "Something" is a pivot value
Quick Sort

Suppose 3 is our pivot.

5 1 3 8 6 4 7 2
Quick Sort

Split into parts < 3, == 3, > 3

1 2 3 5 8 6 4 7 2

1 2 3 5 8 6 4 7
Quick Sort

If 5 was our pivot instead:

\[
\begin{array}{cccccccc}
5 & 1 & 3 & 8 & 6 & 4 & 7 & 2 \\
1 & 3 & 4 & 2 & 5 & 8 & 6 & 7
\end{array}
\]
Quick Sort

Sort each part recursively.

1 3 4 2 5 8 6 7
Quick Sort

Sort each part recursively.

1 2 3 4 5 6 7 8
Quick Sort

Put these lists together and return it.

1 2 3 4 5 6 7 8
Quick Sort

● Same base case as Merge Sort
  ○ A list with only 1 element is already sorted!

● Picking a pivot
  ○ We can pick whatever we want: Typically, the first item, last item, or middle item.
  ○ For now: Use the item at index 0 as the pivot.
Quick Sort

def quicksort(lst):
    if len(lst) <= 1:
        return lst
    ● Get the pivot (item at index 0)
    ● Split it into 3 parts
    ● Sort each part
    ● Put the parts together + return
Quick Sort

def quicksort(lst):
    if len(lst) <= 1:
        return lst

    pivot = lst[0]
    ● Split it into 3 parts
    ● Sort each part
    ● Put the parts together + return
Quick Sort

pivot = lst[0]
left = []
middle = []
right = []
Quick Sort

pivot = lst[0]
left = []
middle = []
right = []
for item in lst:
Quick Sort

for item in lst:
    if item < pivot:
        left.append(item)
Quick Sort

for item in lst:
    if item < pivot:
        left.append(item)
    elif item > pivot:
        right.append(item)
for item in lst:
    if item < pivot:
        left.append(item)
    elif item > pivot:
        right.append(item)
    else:
        middle.append(item)
Quick Sort

[End loop for splitting lst]

sorted_left = quicksort(left)
sorted_right = quicksort(right)
Quick Sort

[End loop for splitting lst]

sorted_left = quicksort(left)

sorted_right = quicksort(right)

return sorted_left + middle + sorted_right
Quick Sort Runtime Complexity

- Picking a pivot
- Splitting the list into 3 parts
- Sorting the parts
- Returning a list
Quick Sort Runtime Complexity

- Picking a pivot \( \mathcal{O}(1) \) for picking \([0]\)
- Splitting the list into 3 parts \( \mathcal{O}(n) \)
- Sorting the parts ?
- Returning a list \( \mathcal{O}(n) \)

How many levels of recursion do we have?
Quick Sort

Picking the median as pivots

5 1 3 8 6 4 7 2
Quick Sort

Picking the median as pivots

1 3 4 2 5 8 6 7
Quick Sort

Picking the median as pivots

1 3 4 2 5 8 6 7
1 2 3 4 6 7 8
Quick Sort

Picking the median as pivots

1 3 4 2
5
8 6 7

1 2 3 4
6 7 8

1 2
6 8
Quick Sort

Picking the median as pivots

1 3 4 2 5 8 6 7
1 2 3 4 6 7 8
1 2 6 8
1 2
Quick Sort

Picking the median as pivots: lgn levels

1 3 4 2 5 8 6 7
1 2 3 4 6 7 8
1 2 6 8
1 2 6 8
Quick Sort

Picking the median as pivots: $\log n$ levels

$\log n$ levels $\times O(n)$ steps per level
$= O(n \log n)$ runtime complexity
Quick Sort

Picking the smallest number as pivots.

5 1 3 8 6 4 7 2
Quick Sort

Picking the smallest number as pivots.

1  5  3  8  6  4  7  2
Quick Sort

Picking the smallest number as pivots.
Quick Sort

Picking the smallest number as pivots.

1 5 3 8 6 4 7 2

2 5 3 8 6 4 7

3 5 8 6 4 7
Quick Sort

(And so on...)

1  5  3  8  6  4  7  2

2

3

3  5  8  6  4  7
Quick Sort

Smallest number = n levels

1 5 3 8 6 4 7 2

2 5 3 8 6 4 7

3 5 8 6 4 7
Quick Sort

Smallest number = n levels

1 5 2 9 6 4 7 3

n levels * O(n) steps per level
= O(n^2) runtime complexity
Quick Sort

- Best case scenario: $O(n \log n)$
  - Pivots chosen are always along the middle

- Worst case scenario: $O(n^2)$
  - Pivots chosen are around the min/max
Homework

- Assignment 2 due Friday (tomorrow)
  - Office hours tomorrow in BA7172
- Exercise 9 is out (due Thursday)
  - Lab 9 is out
- Exercise 8 due tonight (11 PM)