LAB 9 ADDENDUM: QUICKSORT
QUICKSORT

The basic speedy idea of Quicksort
Quicksort

• Sort these lists:

When \( n = 0, \) or \( n = 1 \)

• \( \text{if } n < 2: \)
  • Stop!
Quicksort \( n = 2 \)

- Check if first element is smaller than the second
- If it isn’t, swap them

*After next slide: note that this step is actually picking the leftmost item to be the pivot, and then partitioning into a [3] “less-than” sublist and an empty “greater-than” sublist*
Quicksort \( n = 3 \)

- Pick an element, we call it the *pivot*
  - Let’s start with the leftmost item as the pivot
Quicksort  \( n = 3 \)

- Find all the elements smaller than the pivot, and all the elements larger than the pivot
  - We call this *partitioning*
- We now have a sub-list of all elements smaller, and a sub-list of all elements larger
- The two sub-lists are not sorted! Just partitioned based on our pivot
Quicksort \( n = 3 \)

- If they were sorted, then we could just combine the lowerList + pivot + greaterList (but we haven’t sorted them yet)
- We call quicksort on our lower-list [3] and it’s only one element! We know how to sort that!
- We call quicksort on our greater list [7], and …ditto
- Now we can combine lowerList + pivot + greaterList
Quicksort $n > 2$

- Pivot is 5
- Partition!
- We have a lower list of 2 and a greater list of 1, we know how to sort those!
  - Call quicksort again, on each
- Sorted!
Quicksort

No lower list!

After each quicksort call:
We know our pivot is sorted, but nothing else!

Combine all sub-lists at the end!
Quicksort

How many levels of quicksort calls?
How many levels for a list of length 8?
Pre-sorted Quicksort

- 2 elements = 2 levels of calls to quicksort
- 4 elements = 4
- 8 elements = 8
- 16 elements = 16

- What is this growth rate?
  - Number of levels grows by $O(n)$

Levels of recursive calls to quicksort works a little like our “outer loop” for iterative sorting methods.
For each call to quicksort, how many operations?

During partitioning, has to look at each element to see if it’s less than or greater than the pivot!

As always, this is technically: $n$

But we drop the constants!
Quick Sort Operations

• n operations for each call

• **What is this growth rate?**
  - Number of operations at each level grows by \(O(n)\)
  - Add another level? Add another n operations  

• What is the run-time of Quick Sort when the list is sorted?
  - \(O(\text{Number of calls} \times \text{Number of operations/call})\)
  - \(O(n \times n)\)
  - \(O(n^2)\)  

\(O(n^2)\) isn’t great. (Especially for a recursive algorithm)
Quicksort

1 2 3 4

Let’s pick a random pivot!

How many levels quicksort calls?
How many call levels for a list of length 8?
Pre-sorted Quicksort

- 2 elements = 1 levels of calls to quicksort
- 4 elements = 2
- 8 elements = 3
- 16 elements = 4

What is this growth rate similar to?
- Paper folding from Wednesday!
- Number of levels grows by $O(\log n)$
Quick Sort Operations

- **Number of operations?**
  - n operations for each call (still)

- **What is this growth rate?**
  - Number of operations at each level grows by $O(n)$
  - Add another level? Add another n operations

- **What is the run-time of Quick Sort when the list is sorted?**
  - $O(\text{Number of calls} \times \text{Number of operations/call})$
  - $O(n \times \log n)$
  - $O(n \log n)$ **Best case scenario!**

**Worst Case?** We already saw it, sorted list with a bad pivot selected repeatedly: $O(n^2)$
Sorting Algorithm Run-times

- **Insertion Sort**
  - Best Case: $O(n)$
  - Worst Case: $O(n^2)$

- **Bubble Sort**
  - Best Case: $O(n)$
  - Worst Case: $O(n^2)$

- **Selection Sort**
  - Best Case: $O(n^2)$
  - Worst Case: $O(n^2)$

- **Quick Sort**
  - Best Case: $O(n \log n)$
  - Worst Case: $O(n^2)$
Quick Sort

https://www.youtube.com/watch?v=ywWBy6J5gz8
PARTITIONING

Diving into Partitioning for Quicksort
_partition(d, low, high)

• Modify the list, d, such that all the values less than d[low] are to the left of d[low] and all the values greater than d[low] are to the right of d[low]

• Modify d only between low, high (inclusive)

• Using only swap! No insert/delete items!

• d[low] essentially functions as our pivot value
• Return the index/location of this pivot value
_partition(d, low, high)

• d is a list to partition
• low, high are the indices of d to partition (inclusive)
• Return the location of the pivot value
• Modify d so that it is partitioned

• _partition([2,1,3],0,2)' should return 1 and leave d as [1,2,3].
• _partition([5,9,8],0,2)' should return 0 and leave d as [5,8,9].
• _partition([5,9,8,2,1,3],3,5)' should return 4 and leave d as [5,9,8,1,2,3].
_partition(d, low, high)

• Use the `swap(d, i, j)` function to swap values
  ▪ It’s instrumented – will count swaps

• Use the `less(a, b)` function to check less than of values
  ▪ It’s instrumented – will count comparisons

• Use `d[low]` as the pivot
  ▪ Extra credit looks at using a different pivot
_partition(d, low, high)_

- There are ~3 loops
- Each time through the main loop we want to:
  - Move high as close to low as possible (if appropriate)
  - Swap the values located at low, high (if appropriate)
    - Increment low
    - Pivot is now at high
  - Move low as close to high as possible (if appropriate)
  - Swap the values located at low, high (if appropriate)
    - Decrement high
    - Pivot is now at low
- If low == high, you’re looking at one element, no need to partition 1!
\_\text{partition}(d, 0, 2)

\begin{align*}
\text{d} &= \begin{bmatrix} 2 & 1 & 3 \end{bmatrix} \\
\text{low} &\quad \text{high} \\
\text{Is } d[\text{low}] < d[\text{high}] \text{? Yes!} \\
\text{...then } d[\text{high}] \text{ is on the correct side} \\
\text{...then we can decrement high}
\end{align*}

\begin{align*}
\text{d} &= \begin{bmatrix} 2 & 1 & 3 \end{bmatrix} \\
\text{low} &\quad \text{high} \\
\text{Is } d[\text{low}] < d[\text{high}] \text{? No!} \\
\text{...then } d[\text{high}] \text{ is on the wrong side} \\
\text{...then we must swap} \\
\text{(pivot will be located at high)}
\end{align*}

\begin{align*}
\text{d} &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\
\text{low} &\quad \text{high} \\
\text{...since we swapped, then we know that} \\
\text{the not-pivot is now on the correct side} \\
\text{...so we can increment low}
\end{align*}

\begin{align*}
\text{d} &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\
\text{high} &\quad \text{low} \\
\text{Once low is not lower than high, then we} \\
\text{don’t have anything left to partition, and} \\
\text{the pivot value is in the right location!}
\end{align*}

At the end of one call to partition, the only value we know is in the correct location is the pivot value!
\_partition(d, 0, 2)

\[
\begin{array}{ccc}
\text{low} & \text{high} \\
1 & 2 & 3 \\
\end{array}
\]

Is \text{d[low]} < \text{d[high]}? Yes!
...then \text{d[high]} is on the correct side
...then we can decrement high

\[
\begin{array}{ccc}
\text{low} & \text{high} \\
1 & 2 & 3 \\
\end{array}
\]

Once \text{low} is not lower than \text{high}, then we don’t have anything left to partition, and the pivot value is in the right location!

Note that in this example, \text{low} never changes!
\_partition(d, 0, 2)

```
\begin{align*}
d &= \begin{array}{cc}
4 & 3 & 2 \\
\text{low} & \text{high} \\
\end{array} \\
\text{Is } d[\text{low}] < d[\text{high}]? \text{ No!} \\
\text{...then } d[\text{high}] \text{ is on the wrong side} \\
\text{...then we must swap}
\end{align*}
```

```
\begin{align*}
d &= \begin{array}{cc}
2 & 3 & 4 \\
\text{low} & \text{high} \\
\end{array} \\
\text{...since we swapped, then we know that} \\
\text{the not-pivot is now on the correct side} \\
\text{...so we can increment low} \\
\text{...and pivot is now located at high}
\end{align*}
```

```
\begin{align*}
d &= \begin{array}{cc}
2 & 3 & 4 \\
\text{low} & \text{high} \\
\end{array} \\
\text{Is } d[\text{low}] < d[\text{high}]? \text{ Yes!} \\
\text{...then } d[\text{low}] \text{ is on the correct side} \\
\text{...then we can increment low}
\end{align*}
```

```
\begin{align*}
d &= \begin{array}{cc}
2 & 3 & 4 \\
\text{high low} \\
\end{array} \\
\text{Once low is not lower than high, then we} \\
\text{don’t have anything left to partition, and} \\
\text{the pivot value is in the right location!}
\end{align*}
```
\texttt{\_partition(d, 0, 2)}

\begin{align*}
\text{d} &= \begin{bmatrix}
1 & 4 & 3 \\
\text{low} & \text{high}
\end{bmatrix} \\
\text{Is } d[\text{low}] < d[\text{high}]? & \text{ Yes!} \\
\ldots \text{then } d[\text{high}] & \text{ is on the correct side} \\
\ldots \text{then we can decrement high}
\end{align*}

\begin{align*}
\text{d} &= \begin{bmatrix}
1 & 4 & 3 \\
\text{low} & \text{high}
\end{bmatrix} \\
\text{Is } d[\text{low}] < d[\text{high}]? & \text{ Yes!} \\
\ldots \text{then } d[\text{high}] & \text{ is on the correct side} \\
\ldots \text{then we can decrement high}
\end{align*}

\begin{align*}
\text{Once low is not lower than high, then we don’t have anything left to partition, and the pivot value is in the right location!}
\end{align*}

\textbf{Note that in this example, the list is partitioned but not sorted! Partition only sorts the pivot into the correct location.}
_partition(d, 0, 3)

This longer list decrements high as much as possible, and increments low as much as possible...but doesn’t have to loop back to decrementing high

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Is $d[\text{low}] < d[\text{high}]$? Yes!
...then $d[\text{high}]$ is on the correct side
...then we can decrement high

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Is $d[\text{low}] < d[\text{high}]$? No!
...then $d[\text{high}]$ is on the wrong side
...then we must swap

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Once low is not lower than high, then we don’t have anything left to partition, and the pivot value is in the right location!

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Is $d[\text{low}] < d[\text{high}]$? Yes!
...then $d[\text{high}]$ is on the correct side
...then we can decrement high

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

...since we swapped, then we know that the not-pivot is now on the correct side
...so we can increment low
\_partition(d, 0, 3)

This longer list decrements high as much as possible, and increments low as much as possible...but doesn’t have to loop back to decrementing high

Is \(d[\text{low}] < d[\text{high}]\)? No!
...then \(d[\text{high}]\) is on the wrong side
...then we must swap

Is \(d[\text{low}] < d[\text{high}]\)? Yes!
...then \(d[\text{low}]\) is on the correct side
...then we can increment low

...since we swapped, then we know that the not-pivot is now on the correct side
...so we can decrement high

Once \(\text{low}\) is not lower than \(\text{high}\), then we don’t have anything left to partition, and the pivot value is in the right location!
This one is nearly identical to the previous example with [3,2,5,1], but uses a duplicate value.

\_partition(d, 0, 3)

**Initial Array:** 3 2 5 3

**Partitioning:**
- **Is d[low] < d[high]?: No!**
  - Then d[high] is on the wrong side.
  - Then we must swap.

**After Swap:** 3 2 5 3

**Next Partitioning:**
- **Is d[low] < d[high]?: Yes!**
  - Then d[low] is on the correct side.
  - Then we can increment low.

**After Incrementing Low:** 3 2 5 3

**Next Partitioning:**
- **Is d[low] < d[high]?: No!**
  - Then d[low] is on the wrong side.
  - Then we must swap.

**After Re-Swapping:** 3 2 5 3

**Final State:** 1 2 3 5

Once low is not lower than high, then we don't have anything left to partition, and the pivot value is in the right location!
\_partition(d, 0, 2)

low, high can be used to look at only a subset of d. It’s not too different from what we’ve already been doing.

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is \_d[low] < \_d[high]? Yes!
...then \_d[high] is on the correct side
...then we can decrement high

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...since we swapped, then we know that the not-pivot is now on the correct side
...so we can increment low

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is \_d[low] < \_d[high]? No!
...then \_d[high] is on the wrong side
...then we must swap

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once low is not lower than high, then we don’t have anything left to partition, and the pivot value is in the right location!
_partition(d, 0, 3)

Is d[low] < d[high]? Yes!
...then d[high] is on the correct side
...then we can decrement high

...since we swapped, then we know that the not-pivot is now on the correct side
...so we can increment low

Is d[low] < d[high]? No!
...then d[high] is on the wrong side
...then we must swap

...since we swapped, then we know that the not-pivot is now on the correct side
...so we can decrement high...
_partition(d, 0, 4)

This longer list decrements high as much as possible, and increments low as much as possible...and at the end starts the main loop over again.

Is d[low] < d[high]?
Yes!
...then d[high] is on the correct side
...then we can decrement high

...since we swapped, then we know that the not-pivot is now on the correct side
...so we can increment low

Is d[low] < d[high]?
No!
...then d[high] is on the wrong side
...then we must swap

...since we swapped, then we know that the not-pivot is now on the correct side
...so we can decrement high

Start the main loop over again?
Is d[low] < d[high]?
Yes!
...then d[high] is on the correct side
...then we can decrement high...
_partition(d, 0, 6)

A longer example with less explanation
_partition(d, 0, 6)

A longer example with less explanation
_partition(d, 0, 6)

A longer example showing lots of swapping, and more than one round through the main loop.

Moving high closer to low

4 5 6 1 2 3 7

Moving low closer to high

3 2 6 1 4 5 7

Moving low closer to high

3 5 6 1 2 4 7

Moving high closer to low

3 2 6 1 4 5 7

Start the main loop over again– moving high closer to low

3 4 6 1 2 5 7

Whichever index contains the pivot, you want to move the other index closer to it, as much as possible (considering less than!)
\_partition(d, low, high)

• No need to do this recursively, iteratively is fine!

• When you swap, you know that the not-pivot is in the correct side, so you should increment-low or decrement-high – whichever isn’t the pivot
If you implement _partition differently, your run-time graphs may look surprising.

*Especially* if you’re using operations other than *swap*!

*Insert/delete/append are expensive!*  
Avoid using them!
THESE ALL MAKE GREAT TEST CASES