

GCIS-123

Software Development & Problem Solving

7: Sorting

RIT

**Golisano College of
Computing and
Information Sciences**

SUN	MON (10/3)	TUE	WED (10/5)	THU	FRI (10/7)	SAT
Unit 06: Arrays, Recursion, & Searching					Unit 07: Sorting	
	Loops to Recursion		Binary Search Assignment 6.1 Due (start of class)		Mini Practicum Unit 6 Assignment 6.2 Due (start of class)	
SUN	MON (10/10)	TUE	WED (10/12)	THU	FRI (10/14)	SAT
	Fall Break		Unit 07: Sorting (cont).			
	No Class				Problem Solving 7 Assignment 7.1 Due (start of class)	



SUN	MON (10/10)	TUE	WED (10/12)	THU	FRI (10/14)	SAT
	Fall Break		Unit 07: Sorting (Cont).			
	No Class				Problem Solving 7 Assignment 7.1 Due (start of class)	
SUN	MON (10/17)	TUE	WED (10/19)	THU	FRI (10/21)	SAT
	Midterm Exam 2 (10%)		Unit 08: Python Lists, Tuples, & Reference Types			
	Written (3%) Practical (7%)		Mini-Practicum Unit 7 Assignment 7.2 Due (start of class)			

7.1 Accept the Assignment

You will create a new repository at the beginning of every new unit in this course. Accept the GitHub Classroom assignment for this unit and clone the new repository to your computer.



- Your instructor will provide you with a new GitHub classroom invitation for this unit.
- Upon accepting the invitation, you will be provided with the URL to your new repository, click it to verify that your repository has been created.
- You should have created a `SoftDevI` directory as part of your classwork last week.
 - If you have not, create it now.
 - Navigate to your `SoftDevI` directory and clone the new repository there.
- Open your repository in VS Code and make sure that you have a terminal open with the **PROBLEMS** tab visible.

PROBLEMS



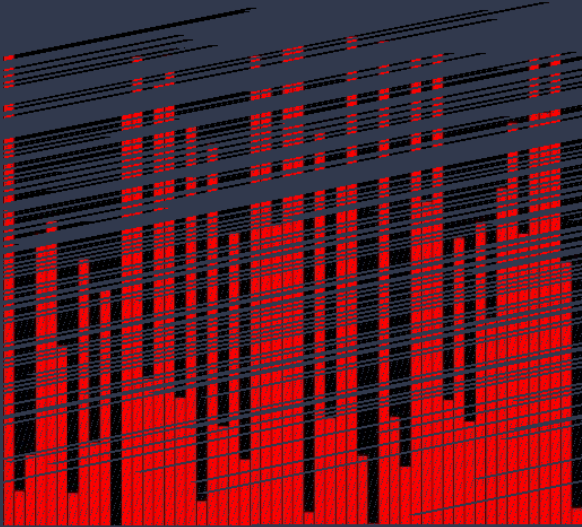
Whenever you see this image, it's a reminder to check your PROBLEMS tab.



If VS Code has identified any problems, try to fix them. If you're not sure how, raise your hand and ask for help!

And remember: it's not about *avoiding* making mistakes. It's about learning how to fix them as we make them.

Sorting



As we did in the last unit with linear search and binary search, in this unit we will compare algorithms that accomplish the same task (sorting) but with significant differences in time complexity.

- In the last unit we explored several different algorithms and analyzed their complexities, including:
 - Linear Search - $O(n)$
 - Binary Search - $O(\log_2 n)$
- This week will focus on sorting algorithms. We will explore:
 - Naive Sorts
 - Merge Sort
 - Quicksort
- Today we will focus specifically on *naive sorts*. These are very simple sorting algorithms that are fairly easy to grasp, but are inefficient. In particular, we will look at:
 - Insertion Sort
 - Function parameters
 - Comparators
 - Lots of complexity comparisons!

- A **data structure** is a grouping of related elements, e.g. an array.
- **Sorting** refers to arranging the elements within a data structure into some order.
 - Smallest to largest
 - Largest to smallest
 - Alphabetical order
 - etc.
- When we talk about sorting, it is important that we have a shared understanding of a few basic terms.
 - An **in-place** sort is done by rearranging the elements within the data structure. This is considered to be **destructive** because the original ordering is destroyed.
 - A sort that is **not** in-place leaves the original data structure intact and creates a sorted **copy**.
 - Given any two elements in the data structure, we must be able to determine which comes first in **natural order**. The natural order may change depending on the problem being solved.

Sorting

Q: You are sorting the players in a tournament based on their final scores. In what order do you sort them?

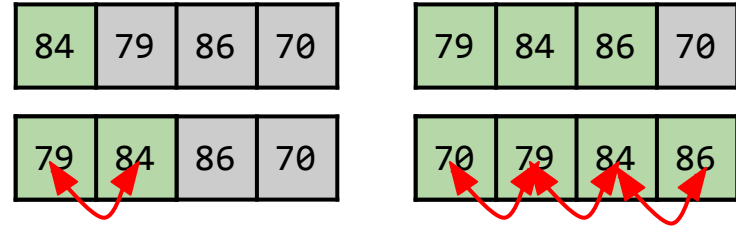


A: It depends! Is it soccer, or golf? The **natural order** of the elements may change based on the problem domain!

Insertion Sort

- There are *lots* of different sorting algorithms, and we will explore several of them this week.
- Today we will be talking about **insertion sort**.
- The concept of insertion sort is very simple.
 - Begin by dividing the array into two **partitions**: a **sorted** partition and an **unsorted** partition.
 - At the beginning of each iteration, the **leftmost** element in the unsorted partition is moved to the **rightmost** position in the sorted partition.
 - If the new element is out of place, it is **swapped** with its neighbor to the left.
 - This **swapping** continues until the element is in the correct position.
 - Repeat until the sorted partition is the **entire list**.

Begin by dividing the array into **sorted** and **unsorted** partitions. At first the sorted partition only has **one element**.



With each iteration, we move the **leftmost** element in the unsorted partition to the **rightmost** position in the sorted partition.

Sometimes the new element needs to be **swapped** one or more times to its left.

And sometimes no swap is needed.

7.2 Beginning Insertion Sort

Hearing an algorithm described to you (even with pictures!) may not be enough to understand the nuts and bolts of how it works under the hood. Implementing the algorithm may be a lot more instructive! So let's start doing that now by writing some helper functions that the Insertion Sort algorithm will need.

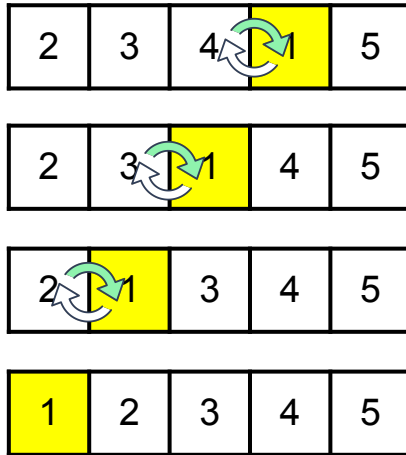
```
original array...  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
swapped indexes 0 and 5...  
[6, 2, 3, 4, 5, 1, 7, 8, 9, 10]  
  
swapped indexes 2 and 3...  
[6, 2, 4, 3, 5, 1, 7, 8, 9, 10]  
  
swapped indexes 8 and 9...  
[6, 2, 4, 3, 5, 1, 7, 8, 10, 9]
```

We will be writing a lot of helper functions to support the various sorts that we implement.

- Create a new Python module in a file named "sorts.py" and create a function named "swap" that declares a parameter for `an_array` and two indexes named `a` and `b`.
 - Use a temporary variable to swap the values at the indexes.
- Add a `main` function to test your swap function.
 - Create an array with the values 1-10.
 - Print the array before and after swapping the values at several indexes.

7.3 Continuing Insertion Sort

It is usually easier to implement large, complex algorithms in a series of discrete steps rather than trying to do it all at once. Let's work on the next increment in the Insertion Sort algorithm: a function that will shift a value that is less than its neighbor to the left.



- Open the `sorts` module and create a new function named "shift" that declares parameters for `an_array` and an `index`.
 - Use a loop that compares the value at the `index` to the value to its left (`index-1`). If the value at the `index` is less, swap the two values and decrement `index`.
 - Continue until the value is no longer less than the value to its left.
 - Remember to stop if `index` reaches 0!
- Test your `shift` function by calling it from `main`.
 - Create an array with the values 10-1.
 - Call `shift` with several different indexes and verify that the values end up in the correct position by printing the array before and after the shift.

Shifting refers to *repeatedly* swapping an element with the neighbors to its left until it is in the right spot.

7.4 Finishing Insertion Sort

There is just one more step to implementing the Insertion Sort algorithm: use the swap and shift helper functions that we've already written to implement the sort! Let's do that now.

```
original array...  
[10, 1, 7, 10, 2, 4, 2, 6, 3, 1]  
after sorting...  
[1, 1, 2, 2, 3, 4, 6, 7, 10, 10]
```

```
original array...  
[4, 5, 5, 2, 9, 9, 9, 9, 10, 8]  
after sorting...  
[2, 4, 5, 5, 8, 9, 9, 9, 9, 10]
```

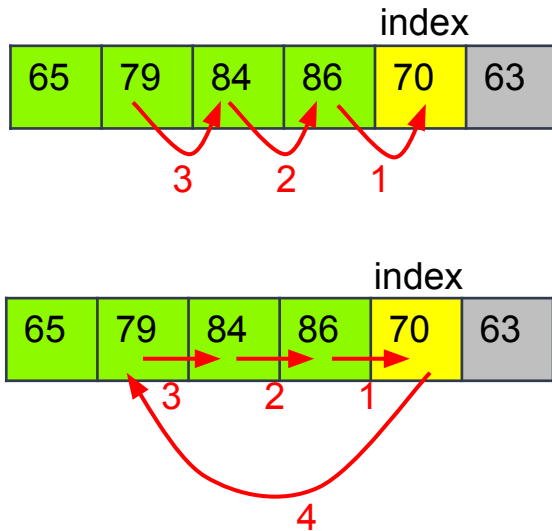
```
original array...  
[7, 10, 4, 8, 7, 6, 10, 8, 4, 8]  
after sorting...  
[4, 4, 6, 7, 7, 8, 8, 8, 10, 10]
```

You should observe that each array is properly sorted.

- Open your `sorts` module and create a new function named `"insertion_sort"` that declares a parameter for `an_array`.
 - Use a loop to iterate over the indexes in `an_array`.
 - Call your `shift` function for each index!
- Test your `insertion_sort` function by calling it from `main`.
 - Create an array with random values from 1-10.
 - Print the array before and after sorting.
 - Repeat this a few times.

7.5 Shifting == Swapping?

There is often more than one way to implement an algorithm. Each time we swap two elements in the array we need to perform two read/write operations. Is there a more efficient way to shift elements? Yes!



- Open your sorts module and define a function named "shift_wo_swap" that declares parameters for `an_array` and an `index`.
 - Use a temporary variable named `target` that stores the value at `index`.
 - You will need to find the proper location of `target`, called `target_index`, at which `target` is to be inserted. Initialize `target_index` to `index`.
 - You will shift all elements in range 0 to `index-1` which are greater than `target` to the right by one position.
 - Use a loop that compares `target` to the value to its left (`index-1`). If `target` is less, move the value to its right and decrement `target_index`.
 - Continue until `target` is no longer less than the value.
 - Finally, place `target` at `target_index`.

7.6 Insertion Sort Without Swaps

Our new, more efficient shifting algorithm is no good if we're not actually using it! Let's create a modified version of the Insertion Sort algorithm that uses it.

```
original array...  
[10, 1, 7, 10, 2, 4, 2, 6, 3, 1]  
after sorting...  
[1, 1, 2, 2, 3, 4, 6, 7, 10, 10]
```

```
original array...  
[4, 5, 5, 2, 9, 9, 9, 9, 10, 8]  
after sorting...  
[2, 4, 5, 5, 8, 9, 9, 9, 9, 10]
```

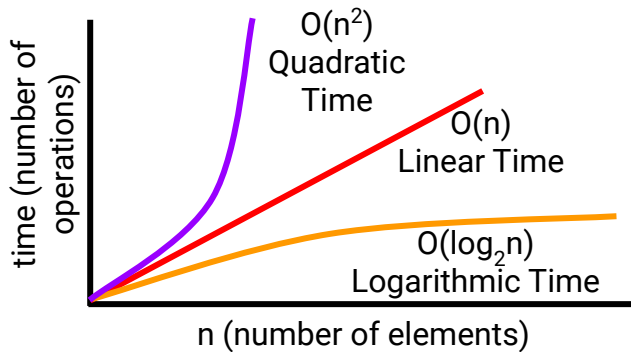
```
original array...  
[7, 10, 4, 8, 7, 6, 10, 8, 4, 8]  
after sorting...  
[4, 4, 6, 7, 7, 8, 8, 10, 10, 10]
```

- Open your `sorts` module and define a function named `"insertion_sort_wo_swap"` that declares parameters `for an_array`
- Test your `insertion_sort_wo_swap` function by calling it from `main`.
 - Create an array with random values from 1-10.
 - Print the array before and after sorting.
 - Repeat this a few times.

You should observe that each array is properly sorted.

Insertion Sort Complexity

Insertion sort runs in **quadratic time** in the average and worst cases. The time complexity grows much more quickly than the size of the input.



However, if the data is **sorted** or **nearly sorted**, insertion sort performs quite well. This makes it **very** useful under certain conditions.

- The performance of insertion sort changes dramatically depending on the input.
 - The algorithm must iterate over the entire array, which is an **$O(n)$** operation.
 - Each time an element is added to the sorted partition, it must be **shifted** 0 or more times.
- If the input is **sorted**, then insertion sort doesn't need to shift any elements at all.
 - This means that the **best case** time complexity is **$O(n)$** .
 - This also means that **mostly sorted** data will be close to **$O(n)$** .
- If the input is **random** then each element has to be shifted an average of about $\frac{1}{2}$ the number of elements in the sorted partition.
 - This is an $O(\frac{1}{2}n)$ operation.
 - The $O(\frac{1}{2}n)$ operation needs to be done for each of the n elements in the array.
 - Therefore the **average** complexity is **$O(\frac{1}{2}n * n)$** or **$O(n^2)$** .
- If the input is in **reverse** order, then every element must be shifted all the way to the left.
 - Like the average case, the **worst case** is **$O(n^2)$** .

7.7 Timing Insertion Sort

Performing a thoughtful analysis of an algorithm is one way to determine its complexity, but it doesn't tell us how long the algorithm *really* takes to run. Exactly how many seconds is $O(n^2)$ anyway? Let's find out by timing Insertion Sort on a few arrays!



```
sorted: 0.0021825 seconds
random: 7.6523956 seconds
reverse: 15.063918 seconds
```

- Create a new Python module in a file named `"sort_times.py"` and create a function named `"insertion_sort_function_timer"` that declares `an_array` parameter.
 - You will need to `import` `sorts`, `time`, and `array_utils` modules.
 - Time how long it takes for insertion sort to sort `an_array` and return the value .
- From your `main` function, time how long it takes to sort the following arrays:
 - An array of 3,000 elements in sorted order.
 - An array of 3,000 random elements.
 - An array of 3,000 elements in reverse sorted order.

Function Parameters

One function may declare a **parameter for another function**.

A function's `__name__` may be printed, and the function may be **called** (even with parameters)

```
1 def func_caller(a_func, x):
2     print(a_func.__name__)
3     result = a_func(x)
4     print(result)
5
6 def square_it(y):
7     return y * y
8
9 def double_it(z):
10    return z * 2
11
12 func_caller(square_it, 5)
13 func_caller(double_it, 5)
```

A function is passed in as an argument by specifying its name *without* parentheses.

- We are already used to thinking about different **types** in Python, e.g. `int`, `float`, `str`, and `Array`.
- In Python, there is also a type that represents **functions**.
 - You can refer to a function by using its name without parentheses, e.g. `print`
- Like values of other types, functions can be:
 - Assigned to variables, e.g. `p = print`
 - Printed, e.g. `print(input)`
 - Stored in arrays, e.g. `an_array[4] = turtle.up`
- Also like other types, a function may be **passed as an argument into another function**.
 - At this point the function may be called using the parameter name.
 - It may also be **passed into another function** and **called** from there!
- Every function has a name that can be retrieved using dot notation, e.g.
 - `print(func.__name__)`

7.8 Functions as Parameters

Python functions can be used in ways similar to any other type: assigned to variables, stored in arrays, and even passed into another function as an argument! Let's try that out now by passing one function into another and using it!

```
>>> runner(evens, 10)
running 'evens'...
30
>>> runner(evens, 100)
running 'evens'...
2550
>>> _
```

The `runner` function should call whichever other function is passed in as a parameter.

- Create a new Python module in a file named "`activities.py`" and create a function named "`evens`" that, declares a parameter for an integer `n`.
 - Compute and return the **sum of the even numbers from 0 to n**.
- Create another new function named "`runner`" that declares parameters for another **function** and a **number**.
 - Print the `__name__` of the **function** before you call it.
 - Call the **function** parameter with the **number**.
 - Print the value that is returned by the function.
- Call `runner` from `main` with your `evens` function and the number of your choice.

7.9 Racing Insertion Sorts

Sometimes writing code that is more efficient is harder and takes longer. Does it really matter? Is shifting without swaps *really* that much more efficient? Let's find out by racing our two Insertion Sorts against each other!

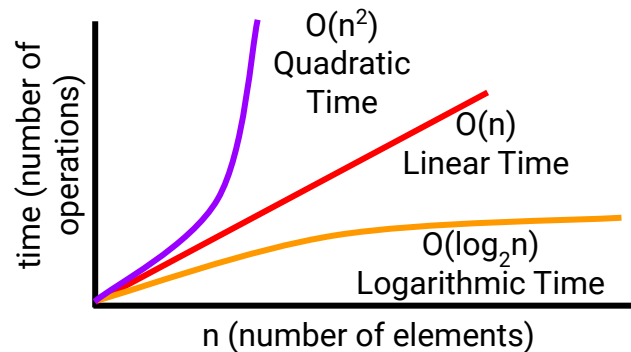


```
insertion sort (swaps): 0.0021825 seconds  
insertion sort (no swaps): ???
```

- Open "`sort_times.py`" and change the name of `insertion_sort_function_timer()` to "`sort_function_timer()`" and declare an additional parameter named `sort_function`.
- Modify the function so that it times `sort_function` to sort an array and return the value .
- From your `main` function, time how long it takes to sort an array using the two versions of insertion sort.
 - Create an array of 3000 random elements.
 - Recall that insertion sort is destructive. Copy the random array before sorting it.
 - Use each array to pass into `sort_function_timer()` .
 - Which one is faster than the other and why?

- We know that **linear search** has a **time complexity** of $O(n)$ and **binary search** has a time complexity of $O(\log_2 n)$.
- All things being equal, an $O(\log_2 n)$ algorithm grows in complexity *much* more slowly than an $O(n)$ algorithm.
- However, if the data structure is not sorted, you only have two choices:
 - Use a linear search
 - Sort first, then use a binary search.
- So far, the only sorting algorithm that we have looked at is **insertion sort**, which has a time complexity of $O(n^2)$.
- Therefore, the cost of sorting first and using binary search is $O(n^2 + \log_2 n)$.
 - This is far worse than a linear search.

Linear or Binary Search?



If you are performing only a single search, linear search alone will be more efficient than sorting first and then using binary search.

7.10 Searching Backward?

Binary Search makes decisions about where to search in an array by comparing the target to the mid value. What happens if the array is sorted in *reverse order*? Will Binary Search be able to find the target? Let's find out!

11	7	5	3	2
0	1	2	3	4
start		mid		end

- Open your `searches` module and navigate to your `main` function.
 - Create an array with the values `1000-1` in descending order.
 - Use binary search to search for several values that you know are in the array and print the results.

SEARCHING
... BACKWARDS

Comparators

- Right now our searches and sorts are **hard coded** to compare values in **increasing order**.
 - `if target < mid`
 - `if arr[index] > arr[index+1]`
- This means that we can't search or sort values into any other order.
- A **comparator** is a function that performs a comparison between two values **a** and **b**.
 - It returns `True` if a comes before b, and `False` otherwise.
 - Exactly **how** the comparator compares the values is up to the programmer.
- We can modify our searches and sorts to use comparators instead of hard-coding.
 - This allows us to **change** search or sort order!

Remember that a function can declare a **function parameter** and that some other function can be passed in as an argument.

```
1 def print_first(comparator, a, b):
2     if comparator(a, b): # if True, a is first
3         print(a, b)
4     else: # b is first
5         print(b, a)
6
7 def more_than(a, b):
8     return a >= b
9
10 print_first(more_than, 5, 10)
```

We can change the behavior of `print_first` by passing different comparator functions into it.

7.11 Two Comparators

Remember that "natural order" depends on the problem that you are trying to solve. Do you sort tournament scores from highest to lowest (soccer) or from lowest to highest (golf)? Let's write a couple of comparator functions that can be used to compare in either ascending or descending order.

a or b?

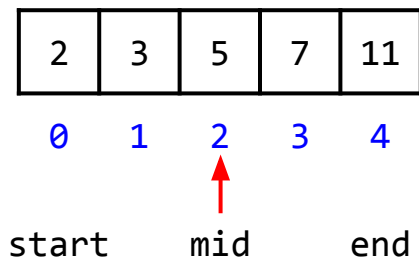
The purpose of a comparator is to help decide which of two values should come first in sorted order.

Depending on the problem that you are trying to solve, the smaller or the larger value may be considered "first."

- Open your `searches` module and create a new function named `increasing_comparator` that declares parameters for two integers `a` and `b`.
 - The function should return `True` if `a` is less than `b` and `False` otherwise.
- Create a second function called `decreasing_comparator` that declares parameters for two integers `a` and `b`.
 - The function should return `True` if `a` is greater than or equal to `b` and `False` otherwise.
- Test your comparators from your `main` function.

7.12 Binary Search 2.0

Our Binary Search function is currently hard coded so that it only works if the data has been sorted in ascending order. Let's make it more flexible by using a comparator function instead of hardcoding the comparison between the target and the midpoint.



Instead of comparing the values to each other directly (e.g. using `<`), use a **comparator** function instead.

- Open your searches module and navigate to your `binary_search` function.
 - Add a parameter for a `comparator` that uses the `increasing_comparator` by default. It should be the *third* parameter (after `target`).
 - Change the `if/elif/else` statement to use the `comparator` instead of directly comparing the value at the `midpoint` to the `target`. If the `comparator` returns `True`, the target comes before the midpoint. Otherwise, it comes after.
 - Don't forget to pass the `comparator` into the recursive function calls!
- Call `binary_search` from `main` with an array sorted in increasing order to make sure that it still works.

7.13 Binary Search 2.0

In theory, our Binary Search function should work with any kind of comparator. But how do we verify that it can search data that is sorted in descending order? By testing it, of course! Let's do that now.

11	7	5	3	2
----	---	---	---	---

0 1 2 3 4



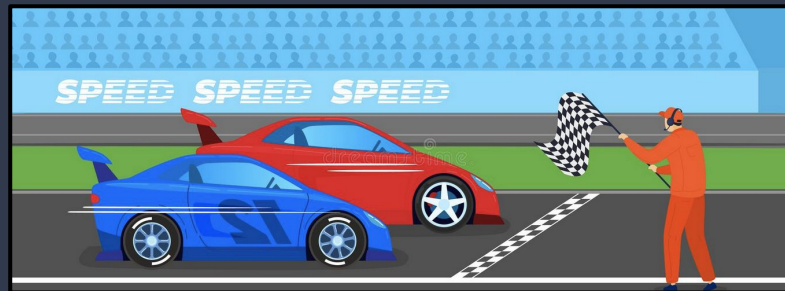
start mid end

SEARCHING
BACKWARDS...

- Open your `searches` module and navigate to your `main` function.
 - Create an array with the values 1,000,000 to 1 in descending order.
 - Call your `binary_search` function using your `decreasing_comparator` and search for the following values:
 - 1
 - 500,000
 - 1,000,000
 - 0

- The strategy of insertion sort is simple: looping over an array performing insertion of one element at a time, which we call shift.
- Insertion sort makes incremental progress. The length of the sorted part of the array is increased by 1 in each iteration.
- Due to this insertion sort has a runtime complexity of **$O(N^2)$** for most cases.
- There are another series of sorts that work more **efficiently** by:
 - Breaking the array into parts (normally 2)
 - Sorting the parts
 - Putting the parts back together
- The one we are discussing today is **Merge Sort**

A Faster Sort



Not all sorts are created equal. Some perform far better than others in most circumstances. We'll be taking a close look at one of them today!

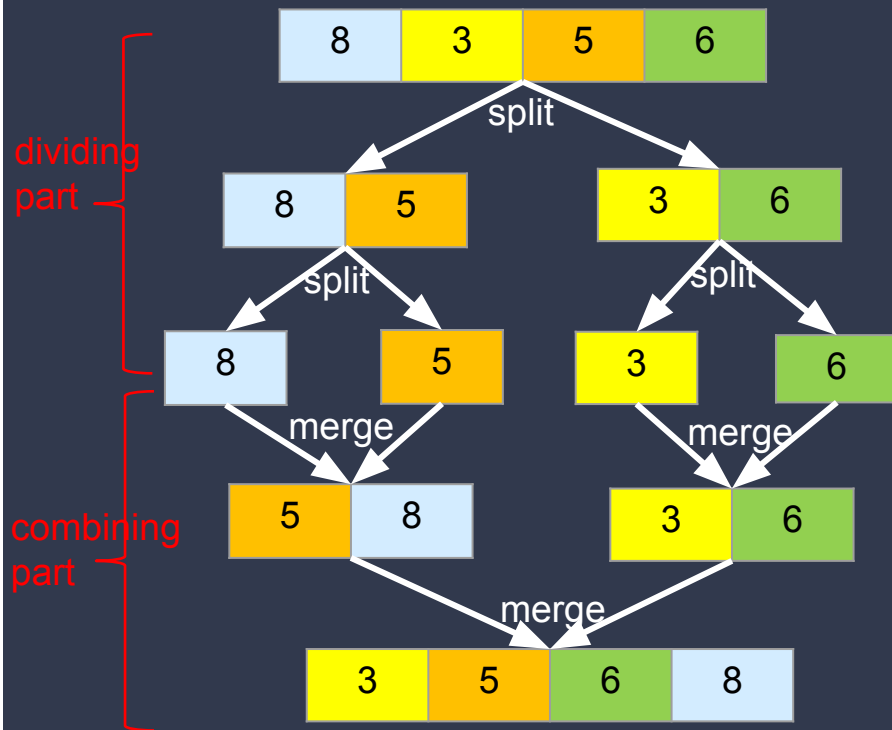
Merge Sort Algorithm

```
define merge_sort (A)
  if A has at most 1 element
    return A
  otherwise
    - split A into two halves
      whose sizes differ by at most 1
    - recursively merge_sort each half
    - merge the two sorted halves
    - return the merged result
```

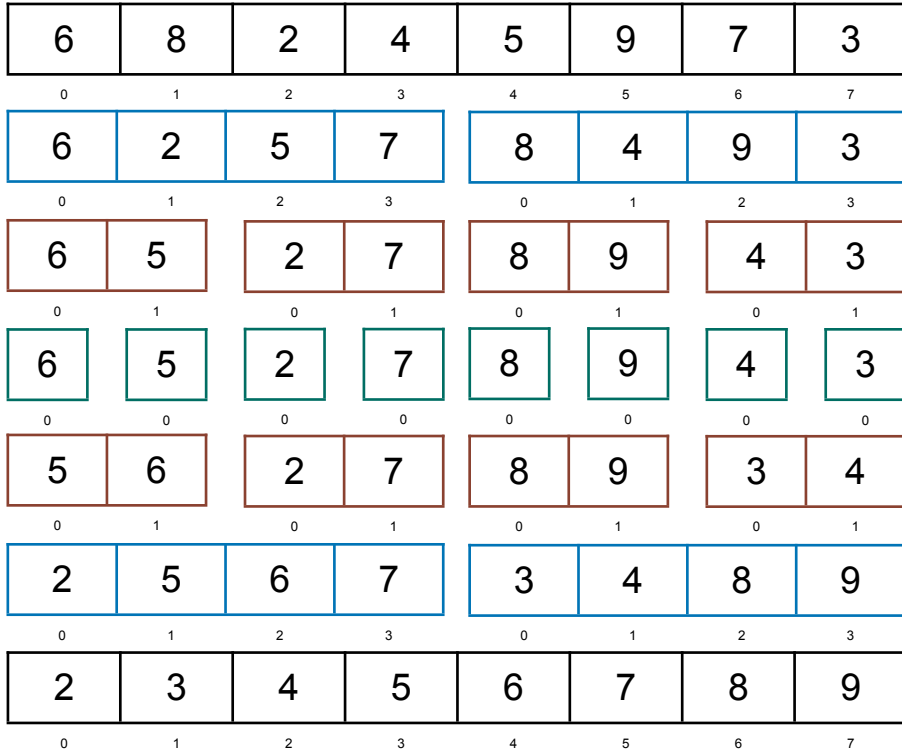
- Take a minute to look at the merge sort algorithm listed on the left
- The general idea is to keep **splitting** the array in **half** (as evenly as possible)
 - Until the array has a size of 1
- Then **merge** the new arrays back together in the **correct order**
- A couple questions to ask yourself before moving on:
 - Under what circumstances would the split arrays differ in size?
 - Why split the array in half?

- Merge sort is a **divide-and-conquer** algorithm
- In `merge_sort`, splitting into `half1` and `half2` is the **dividing** part
 - Recursive calls **conquer** those parts by sorting them
- In our implementation we will place all of the values at **even indexes** into `half1`
- All the values at **odd indexes** will be placed into `half2`
 - Make sure you are referring to the **indexes** and not the value at an index
- The merge function **combines** the results
 - The two sorted arrays are merged together in sorted order

Merge Sort Specifics



Diagramming Merge Sort



We will start with an array, which we split

Even indexes to the left, odd to the right

Repeat, notice no sorting has occurred

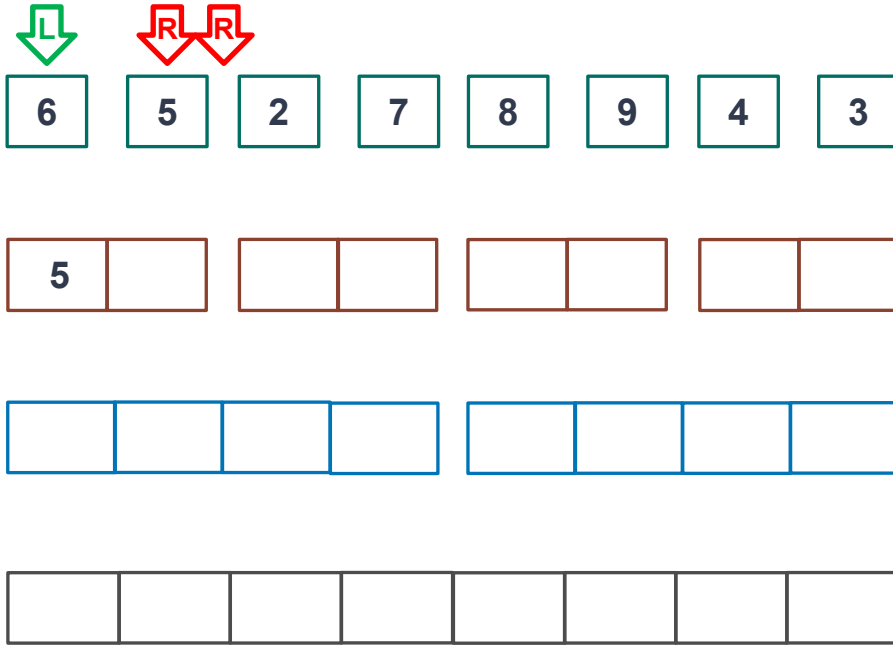
Continue until all arrays are size 1

Merge the arrays back together ...

by comparing the first element in each

placing the smallest in the new array, repeat

Merge Visualization



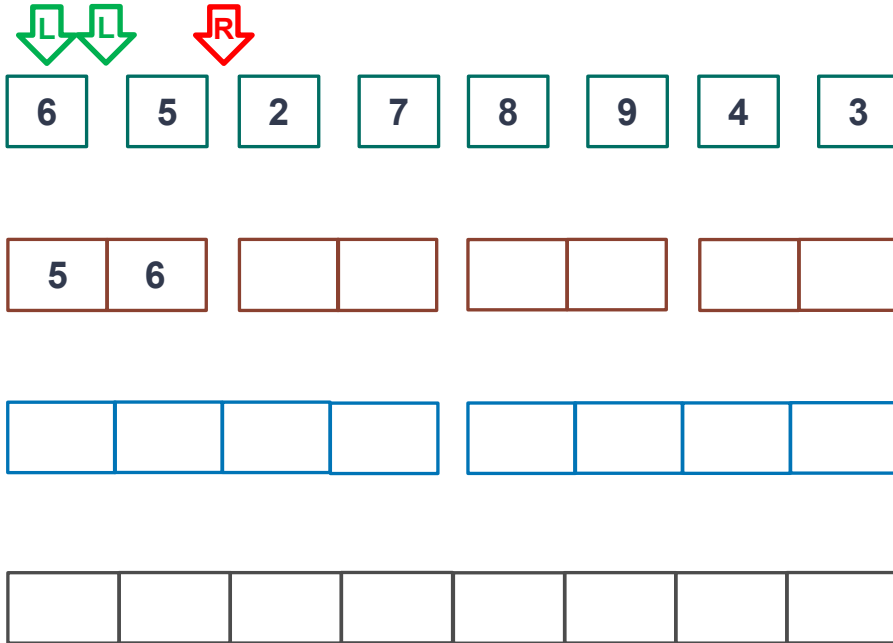
Start at index 0 of left and right

Choose the smallest of the two

Add it to index of new array

Increment indexes

Merge Visualization

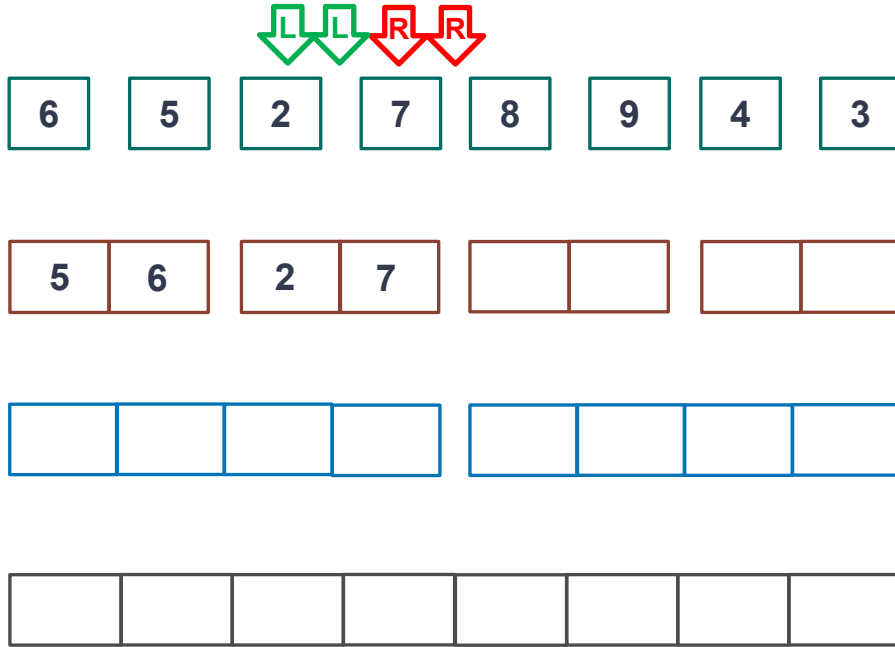


If left or right index is \geq to its length ...

Stop

If either left or right has elements in it add them at the new array's index

Merge Visualization



Repeat for all elements at that level

Merge Visualization



Repeat for all elements at that level

Merge Visualization



Repeat for all elements at that level



Merge Visualization



Repeat for all elements at that level

Merge Visualization



Repeat for all elements at that level

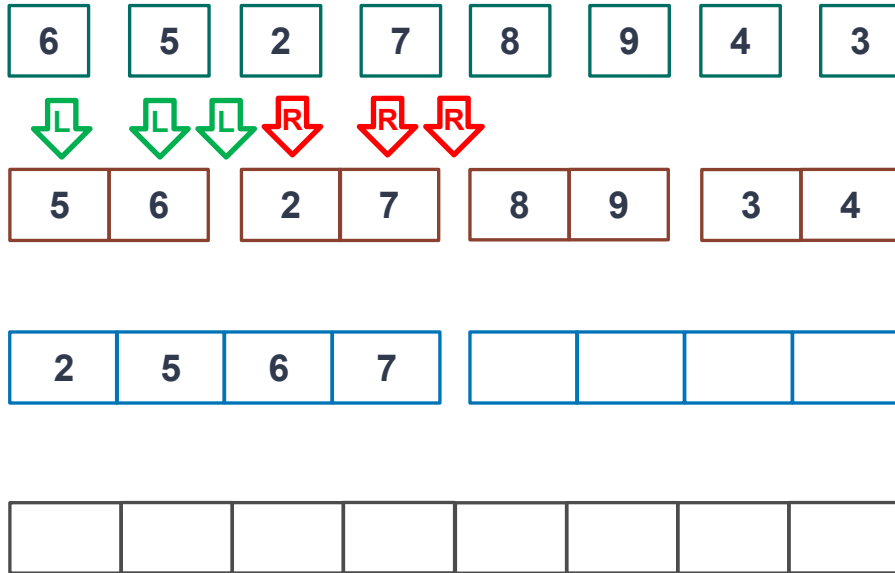


Merge Visualization



Repeat for all elements at that level

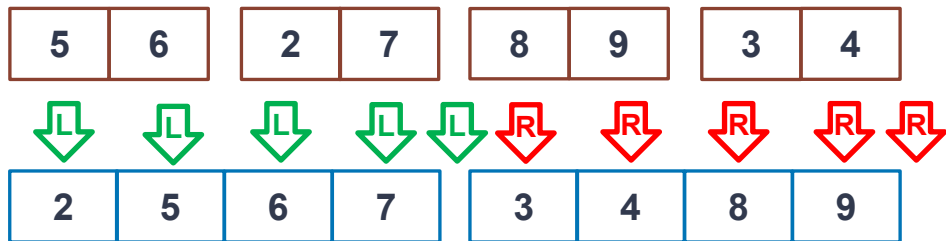
Merge Visualization



When done, move to the next level

And repeat

Merge Visualization



When done, move to the next level

And repeat

Remember, when left or right finishes, just take the rest of the other

7.14 Your Turn

Practicing a sort "on paper" is a useful way to see how it works (which will hopefully make it easier to implement).



Type your solution into a file named "activities.txt".

10	3	5	6	1	4	8	7
----	---	---	---	---	---	---	---

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

7.14 Your Turn

Practicing a sort "on paper" is a useful way to see how it works (which will hopefully make it easier to implement).



Type your solution into a file named "activities.txt".

10	3	5	6	1	4	8	7
10	5	1	8	3	6	4	7
10	1	5	8	3	4	6	7
10	1	5	8	3	4	6	7
1	10	5	8	3	4	6	7
1	5	8	10	3	4	6	7
1	3	4	5	6	7	8	10

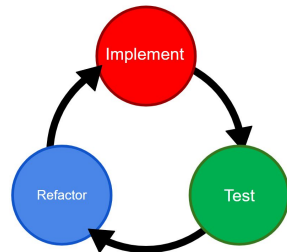
7.15 Starting Merge Sort

As we have done before with complex algorithms, we will implement merge sort using incremental development and test as we go to ensure that it functions as expected each step of the way. Let's start by creating a new module and the unit test for the simplest case.

```
def merge_sort(an_array)
    if an_array has at most 1 element
        return an_array
    otherwise
        - split an_array into two halves
          whose sizes differ by at most 1
        - recursively merge_sort each half
        - merge the two sorted halves
        - return the merged result
```

- Create a `merge_sort.py` file and define a `merge_sort` function that declares a parameter for `an_array`.
 - Write only the code necessary for the easiest case: *if the length of the array is less than 2, return the array.*
- Create `merge_sort_test` module and write at least one test that verifies that your nascent `merge_sort` function works on arrays of length 0 or 1.

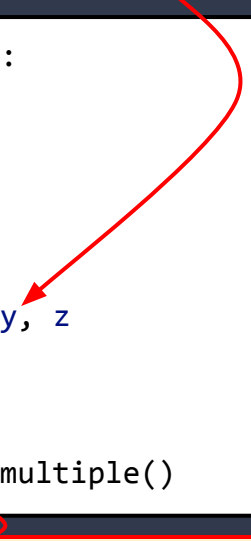
Incremental development and testing is an excellent way to tackle large, complex algorithms a little bit at a time.



Multiple Return Values

A Python function may return multiple values using a `return` statement and a comma-separated list.

```
1 def multiple():  
2     x = 5  
3     y = "abc"  
4     z = 23.4  
5  
6     return x, y, z  
7  
8 def main():  
9     a, b, c = multiple()
```



When a function returns more than one value, the individual values may be assigned to the same number of variables using a comma-separated list.

- Up until this point in the course the Python functions that we have written have only returned a single value.
 - The value that we **explicitly** return using a `return` statement, e.g. `return an_array`
 - In the event that we do not use a `return` statement, Python **implicitly** returns `None`
- Python functions may return **more than one value** as well.
- This is accomplished using a `return` statement followed by a comma-separated list of values.
 - e.g. `return x, y, z` will return all 3 values.
- When calling a function that returns multiple values, the values must be assigned to the same number of variables, again using a comma-separated list.
 - e.g. `a, b, c = my_function()`
- Assigning the return values to the wrong number of variables will result in a `ValueError` (wrong number of values to unpack).

Split

```
define split(A)
    evens = Array()
    odds = Array()
    for each index i in A
        if i is even
            add A[i] to evens
        otherwise
            add A[i] to odds
    return evens, odds
```

Remember, you can use commas to return multiple values, e.g:

```
return half_1, half_2
```

- There are several approaches for the split operation
- One way is to split down the middle
 - This approach has some nice properties, however it is not the only way to do it
- Here we use another method that can be a little bit faster depending on the underlying implementation of your array: the **even-odd** split.
 - The idea is to put every element at an **even-index** in one array, and every element at an **odd-index** in another
- We will sketch out the algorithm using **pseudocode**, a shorthand style of "fake coding" that splits the difference between written language and code.

7.16 An Odd Number of Elements

The split function will need to handle arrays that have an odd number of elements (and so the two "halves" will be different lengths). Practice merge sort "on paper" using an array with an odd number of elements.



Write your solution in
"activities.txt".

7	3	5	13	1	9	11
---	---	---	----	---	---	----

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

7.16 An Odd Number of Elements

The split function will need to handle arrays that have an odd number of elements (and so the two "halves" will be different lengths). Practice merge sort "on paper" using an array with an odd number of elements.



Write your solution in
"activities.txt".

7	3	5	13	1	9	11
7	5	1	11	3	13	9
7	1	5	11	3	9	13
7	1	5	11	3	9	13
1	7	5	11	3	9	13
1	5	7	11	3	9	13
1	3	5	7	9	11	13

7.17

Divide and Conquer

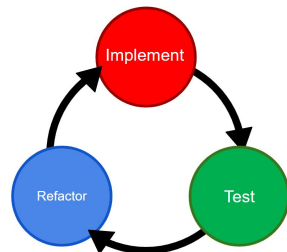
Just as we did with insertion sort, we will create helper functions to make Merge Sort a little more straightforward. Let's begin by writing a helper function that will split the elements in an array into two halves.

```
def split(A)
    evens = Array()
    odds = Array()
    for each index i in A
        if i is even
            add A[i] to evens
        otherwise
            add A[i] to odds
    return evens, odds
```

```
def multiple():
    return 5, "abc", 23.4

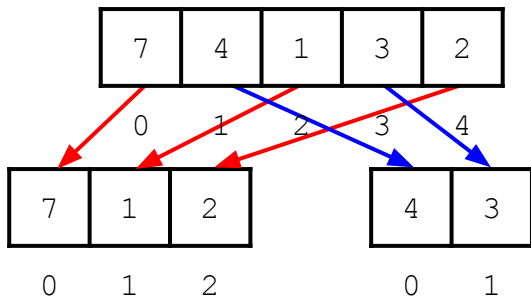
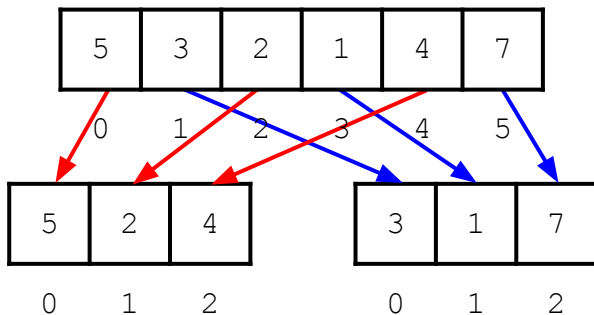
a, b, c = multiple()
```

- Open your `merge_sort` module and define a new function named `split` that declares a parameter for `an_array`.
 - Create one array to hold the values at **even indexes** and another to hold the values at **odd indexes**.
 - How do you know how large each array should be? What if `an_array` has an odd number of elements?
 - Use a loop to copy all of the values from `an_array` into the appropriate half.
 - Return **both** halves.



7.18 Testing Split

We'll need to make sure that the `split` function is working as intended for arrays that contain 2 or more elements. Some arrays may contain an odd number of elements, so we'll need to make sure that the `split` function can handle that.



- Open `merge_sort_test` and create at least one test for each of the following `split` scenarios:
 - An array with an **even** number of elements
 - An array with an **odd** number of elements
 - Do you need to worry about arrays with 0 or 1 element?
- Each test should verify that the splits contain the correct elements in the correct order.
- Verify that all of your tests are passing before pushing your working code to GitHub.

Merge

```
def merge(sorted1, sorted2)
  result = Array()
  i1 = 0
  i2 = 0
  while i1 < len(sorted1) and
    i2 < len(sorted2)
    if sorted1[i1] <= sorted2[i2]
      add sorted1[i1] to result
      i1 = i1 + 1
    otherwise
      add sorted2[i2] to result
      i2 = i2 + 1
  if i1 < len(sorted1)
    add rest of sorted1 to result
  otherwise if i2 < len(sorted2)
    add rest of sorted2 to result
  return result
```

- The final step in the Merge Sort algorithm is to merge the two sub-arrays into a single, sorted array.
- We will assume that the sub-arrays that we want to merge are already sorted. It is therefore sufficient to compare the first elements in each to determine the smallest element of the new, merged array
- We move past the smallest element (in whichever array it was in), and continue in this fashion until one all of the elements in one or the other array has been merged into the new array
- We then copy the remaining elements from the other array into the merged array.

7.19 Divide and Conquer

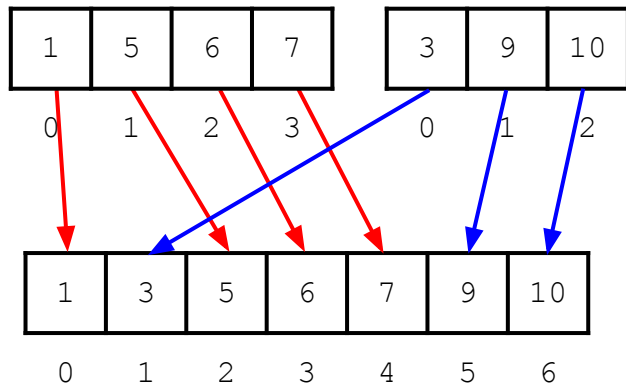
In the divide step of Merge Sort, the array is split until each piece contains only a single element. In the conquer step, the smaller arrays are merged back together in sorted order before returning the new, sorted array. This is by far the most complex part of the algorithm.

```
def merge(sorted1, sorted2)
    result = Array()
    i1 = 0
    i2 = 0
    while i1 < len(sorted1) and
        i2 < len(sorted2)
        if sorted1[i1] <= sorted2[i2]
            add sorted1[i1] to result
            i1 = i1 + 1
        otherwise
            add sorted2[i2] to result
            i2 = i2 + 1
    if i1 < len(sorted1)
        add rest of sorted1 to result
    otherwise if i2 < len(sorted2)
        add rest of sorted2 to result
    return result
```

- Open the `merge_sort` module and define a new function named "merge" that declares parameters for two arrays.
 - Use the given pseudocode to begin sketching out your algorithm using comments.
 - Fill in the Python code for any of the steps that you know that you can do.
 - Ask for help if you get stuck!

7.20 Testing Merge

Just as we did with the split function, we'll need to make sure that our merge function is working by writing a few tests. Remember: the arrays may be different lengths!



One of the two arrays will be finished copying first. Any remaining elements in the other array will need to be copied afterwards.

- Open `merge_sort_test` and create at least one test for each of the following `merge` scenarios:
 - Two arrays with 1 element each.
 - Two arrays with 2 elements each.
 - Two arrays with different lengths.
 - All of the elements in the first array are copied first.
 - All of the elements in the second array are copied first.
- Each test should verify that the array returned contains all of the elements in sorted order.
- Verify that all of your tests are passing before pushing your working code to GitHub.

7.21 Finishing Merge Sort

At this point we have all of the helper functions that we will need, and so it is time to finish the Merge Sort function. Assuming that the split and merge functions are working, the remainder of the Merge Sort algorithm is surprisingly straightforward to implement.

```
def merge_sort(an_array)
    if an_array has at most 1 element
        return an_array
    otherwise
        - split an_array into two halves
          whose sizes differ by at most 1
        - recursively merge_sort each half
        - merge the two sorted halves
        - return the merged result
```

The real work of Merge Sort is done by its helper functions. With those out of the way, implementing the core algorithm is fairly straightforward.

- Open the `merge_sort` module and navigate to the `merge_sort` function.
 - Use the pseudocode to the left to finish the core merge sort algorithm so that it works on arrays of length 2 or more.

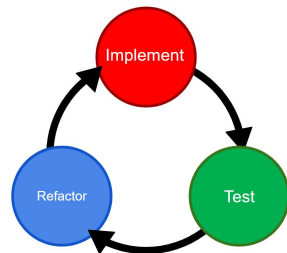
7.22 Testing Merge Sort

We'll need to add a few more tests to the `merge_sort_test` module to make sure that Merge Sort is working for arrays with two or more elements.

```
define merge_sort(an_array)
  if an_array has at most 1 element
    return an_array
  otherwise
    - split an_array into two halves
      whose sizes differ by at most 1
    - recursively merge_sort each half
    - merge the two sorted halves
    - return the merged result
```

Once we have enough tests to verify that Merge Sort works as intended, we've finished the algorithm!

- Open `merge_sort_test` and create at least one test for each of the following `merge_sort` scenarios:
 - An unsorted array with 2 elements.
 - An unsorted array with 3 elements.
 - A sorted array with 4 or more elements.
- Each test should verify that the array returned contains all of the elements in sorted order.
- Verify that all of your tests are passing before pushing your working code to GitHub.



7.23 Sorting an Already-Sorted Array

Some sorting algorithms perform better under certain circumstances. For example, insertion sort runs very quickly if the data is already sorted. Is merge sort any more or less efficient when working with data that is already sorted? Complete the following diagram to find out.



Enter your solution in the `activities.txt` file that you created previously.

2	4	6	8	9	11	13	15

7.23 Sorting an Already-Sorted Array

Some sorting algorithms perform better under certain circumstances. For example, insertion sort runs very quickly if the data is already sorted. Is merge sort any more or less efficient when working with data that is already sorted? Complete the following diagram to find out.



Enter your solution in the `activities.txt` file that you created previously.

2	4	6	8	9	11	13	15
2	6	9	13	4	8	11	15
2	9	6	13	4	11	8	15
2	9	6	13	4	11	8	15
2	9	6	13	4	11	8	15
2	6	9	13	4	8	11	15
2	4	6	8	9	11	13	15

Split Complexity Analysis

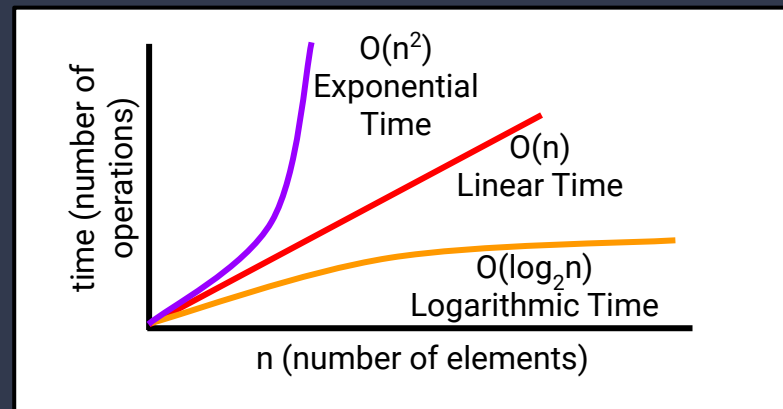
- Understanding the time complexity of the Merge Sort algorithm requires us to analyze the complexity of each of its parts.
- First, the `split` function:
 - $T(n) = 4n + 4$
 $= O(n)$

Remember, we drop all scalars and lower order terms for Big-O notation

```
define split(A)
    evens = Array()
    odds = Array()
    for each index i in A
        if i is even
            add A[i] to evens
        otherwise
            add A[i] to odds
    return evens, odds
```

- The `merge` function:
 - Loop over
 - The index for either `sorted1` or `sorted2` is incremented
 - The total number of iterations cannot be more than the size of `sorted1` plus the size of `sorted2`
 - the number of iterations has an upper bound of the sum of the sizes of `sorted1` and `sorted2`
- $T(n) = O(n)$

Merge Complexity Analysis



Merge Sort Time Complexity Analysis

- So far we know that:
 - `split` runs in $O(n)$ time
 - `merge` runs in $O(n)$ time
- So what is the overall complexity of merge sort?
 - Merge Sort splits the array in half until the sub-arrays have only one element.
 - How many times can an array with n elements be split in half?
 - How many times do the sub-arrays need to be merged back together?
 - In both cases it is $\log_2 n$ times.
- The complexity of Merge Sort is therefore:

$$O(n) \cdot \log_2 n + O(n) \cdot \log_2 n = O(n \log_2 n)$$

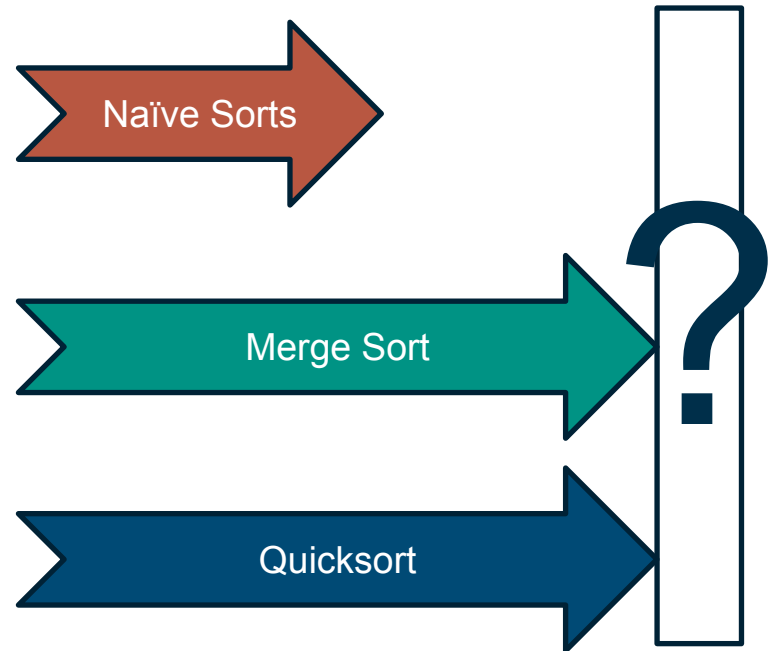
```
define merge_sort (A)
    if A has at most 1 element
        return A
    otherwise
        split A into two halves
            whose sizes differ by at most 1
        sorted1 = merge_sort(half1)
        sorted2 = merge_sort(half2)
        return merge(sorted1, sorted2)
```

We have to **split** the array (an $O(n)$ operation) $\log_2 n$ times, and then we need to **merge** (also $O(n)$) $\log_2 n$ times as well.

A Different Fast Sort

- **Quicksort** is another divide-and-conquer algorithm
 - Pick an element from the list and call it the **pivot**
 - Divide, or partition, the array into **three** parts
 - The resulting three arrays contain all the elements:
 - the array of elements **less than** the pivot
 - the array of elements **equal to** the pivot
 - the array of elements **greater than** the pivot

In our implementation we will use the element at **index 0** as the pivot point

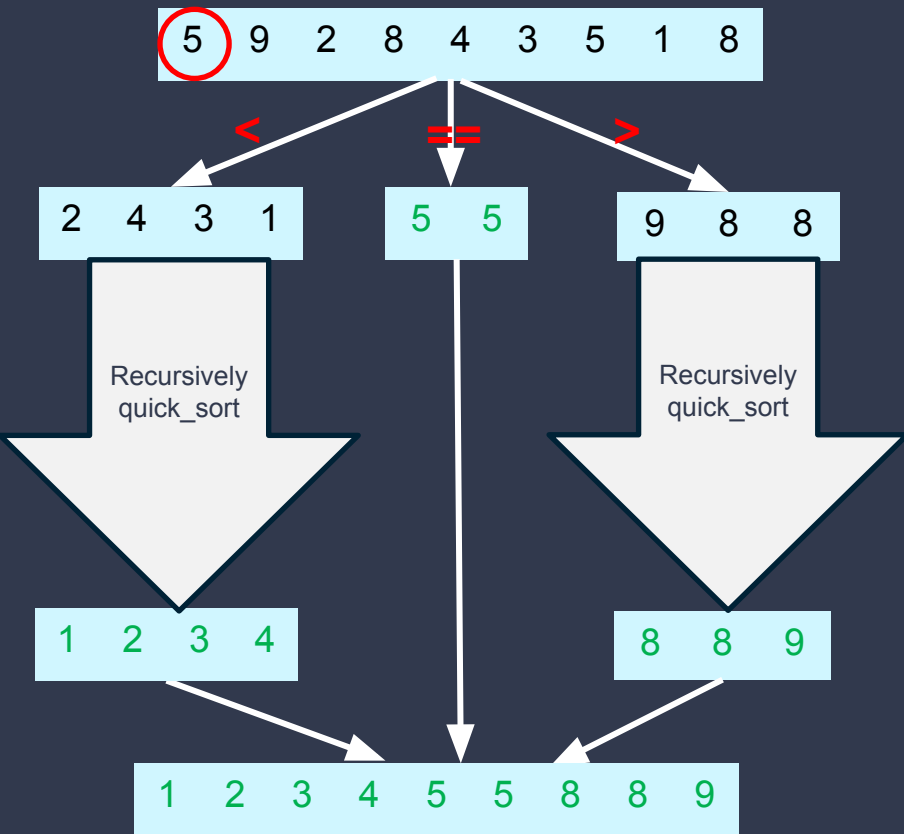


Quicksort Algorithm

```
def quick_sort(A)
    if A has at most 1 element
        return A
    otherwise
        pivot = A[0]
        less, same, more = partition(pivot, A)
        sorted_less = quick_sort(less)
        sorted_more = quick_sort(more)
        return sorted_less + same + sorted_more
```

- The algorithm for quicksort is pretty straightforward; the **partition** step is where all the work occurs
- There are a few things we will need to deal with due to using arrays
- First, sometimes one or more of the arrays will end up being empty; how do we represent an **empty array**?
 - We can achieve this by creating an array of size 0
- The **less, same, more** line may look a little funny
- However, this is the syntax for dealing with **multiple returns** in python
 - This is a python thing, most other languages won't use this syntax
- The last part says we need to **concatenate** the pieces back together
- We'll need to write something to do this for arrays as there is **no + operator** associated with them

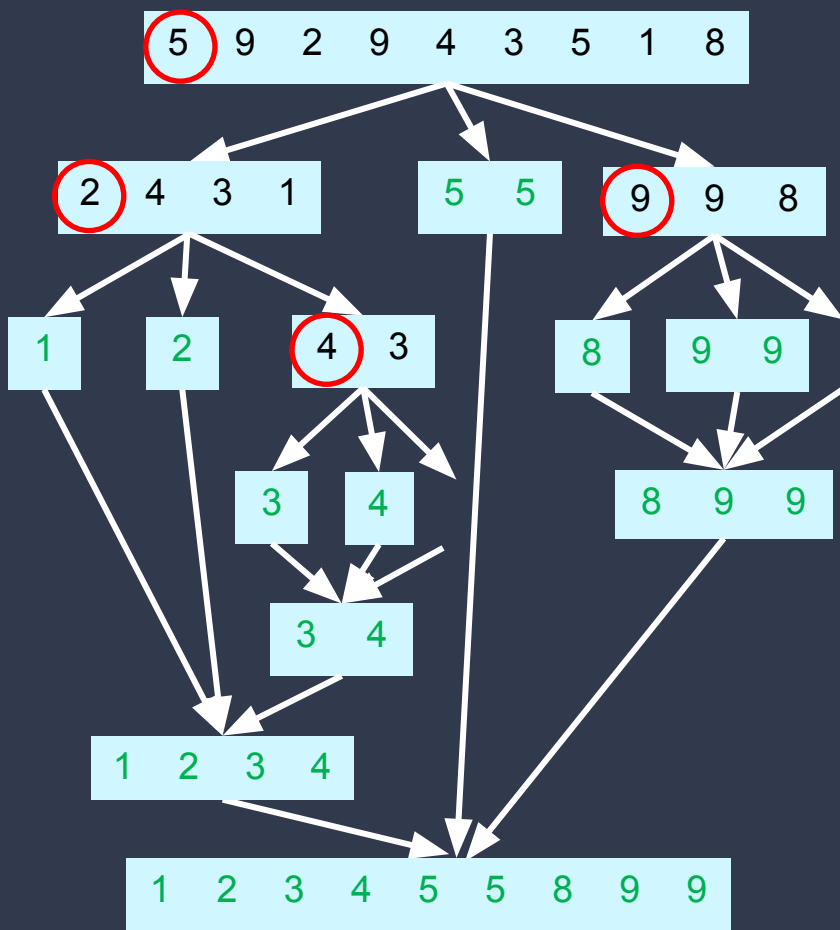
Quicksort Example



Example:

- if the input list is `[5, 9, 2, 8, 4, 3, 5, 1, 8]`
- We will use the first element as the **pivot**.
- quick sort partitions the list into three parts:
 - `less = [2, 4, 3, 1]`
 - `same = [5, 5]`
 - `more = [9, 8, 8]`
- Recursively sort `less` and `more` yields
 - `[1, 2, 3, 4]`
 - `[8, 8, 9]`
- Putting **all three** together yields
 - `[1, 2, 3, 4, 5, 5, 8, 8, 9]`

Quicksort Unrolled



Example: quick_sort on [5, 9, 2, 9, 4, 3, 5, 1, 8]

- Partition [5, 9, 2, 9, 4, 3, 5, 1, 8] into [2, 4, 3, 1], [5, 5], and [9, 9, 8]
- Recursively sort [2, 4, 3, 1] and [9, 9, 8]:
 - partition [2, 4, 3, 1] into [1], [2], [4, 3]
 - Recursively sort [4, 3]
 - partition [4, 3] into [3], [4], []
 - putting [3], [4], [] together yields [3, 4]
 - Putting [1], [2], [3, 4] together yields [1, 2, 3, 4]
 - Partition [9, 9, 8] into [8], [9, 9], []
 - Putting [8], [9, 9], [] together yields [8, 9, 9]
- Putting [1, 2, 3, 4], [5, 5], and [8, 9, 9] yields [1, 2, 3, 4, 5, 5, 8, 9, 9]

Diagramming Quicksort

Key

Less Array

Pivot Array

More Array

6	8	2	5	4	9	7	3
---	---	---	---	---	---	---	---

Pivot on index 0

2	5	4	3	6	8	9	7
---	---	---	---	---	---	---	---

Pivot on index 0 of the new more/less arrays

2	5	4	3	6	7	8	9
---	---	---	---	---	---	---	---

Pivot on index 0 of the new more/less arrays

2	4	3	5	6	7	8	9
---	---	---	---	---	---	---	---

Pivot on index 0 of the new more/less arrays

2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---

Pivot on index 0 of the new more/less arrays

2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---

Once everything is a pivot it is sorted

7.24 Your Turn

Just as with some of the other sorts that we have looked at, you may gain a better understanding of how Quicksort works if you practice diagramming it "on paper" (or in a text file).



10	3	5	6	1	4	8	7
----	---	---	---	---	---	---	---

Use the `activities.txt` file to enter your solution.

7.24 Your Turn

Just as with some of the other sorts that we have looked at, you may gain a better understanding of how Quicksort works if you practice diagramming it "on paper" (or in a text file).



Use the `activities.txt` file to enter your solution.

10	3	5	6	1	4	8	7
3	5	6	1	4	8	7	10
1	3	5	6	4	8	7	10
1	3	4	5	6	8	7	10
1	3	4	5	6	8	7	10
1	3	4	5	6	7	8	10
1	3	4	5	6	7	8	10

7.25 Starting Quicksort

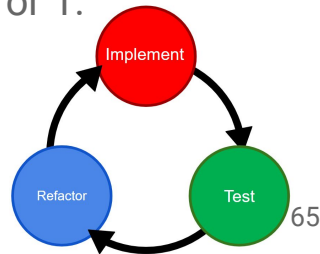
We will implement Quicksort by following a pattern similar to the other complex algorithms that we have implemented; we will break it down into smaller increments, use tests to verify that they work, and then combine the increments into the whole solution. Using the pseudocode to the left, let's start now!

```
def quick_sort(A)
    if A has at most 1 element
        return A
    otherwise
        pivot = A[0]
        less, same, more = partition(pivot, A)
        sorted_less = quick_sort(less)
        sorted_more = quick_sort(more)
        return sorted_less + same + sorted_more
```

Just as we did with Merge Sort, we will focus on incremental development and testing as we go.

- Create a `quicksort.py` file and define a `quicksort` function that declares a parameter for `an_array`.
 - Write only the code necessary for the easiest case: *if the length of the array is less than 2, return the array.*
- Create `quicksort_test` module and write at least one test that verifies that your nascent `quicksort` function works on arrays of length 0 or 1.

Incremental development and testing is an excellent way to tackle large, complex algorithms a little bit at a time.



Partition Algorithm

- Since Quicksort is a divide and conquer algorithm, we need to write the dividing part, **partition**
- For this sort we are going to break it into three parts based on a **pivot**
- The pivot can be any value in the array, but for our examples we will use the value at **index 0**
- Once we know the pivot value, iterate over the array and add all items to one of three new arrays based on their relation to the pivot
- The **less** array will contain all values less than the pivot
- The **same** array will contain all values equal to the pivot
- The **more** array will contain all values greater than the pivot

```
def partition(pivot, A):  
    less_count = number of values less than pivot  
    same_count = number of values equal to pivot  
    more_count = number of values greater than pivot  
  
    less = Array(less_count)  
    same = Array(same_count)  
    more = Array(more_count)  
  
    for every index i in A  
        if A[i] < pivot  
            less.append(A[i])  
        otherwise if A[i] > pivot  
            more.append(A[i])  
        otherwise  
            same.append(A[i])  
    return less, same, more
```

7.26 Handling Duplicate Values

Sometimes the array will contain the same value(s) more than once. Practice diagramming Quicksort on such an array.

The quicksort algorithm works with duplicate values

For the same array, add all pivot value to it and mark them all as pivot

For all other values, partition works the same as normal

Use the `activities.txt` file to enter your solution

3	1	4	2	3	5	1	4
---	---	---	---	---	---	---	---

7.26 Handling Duplicate Values

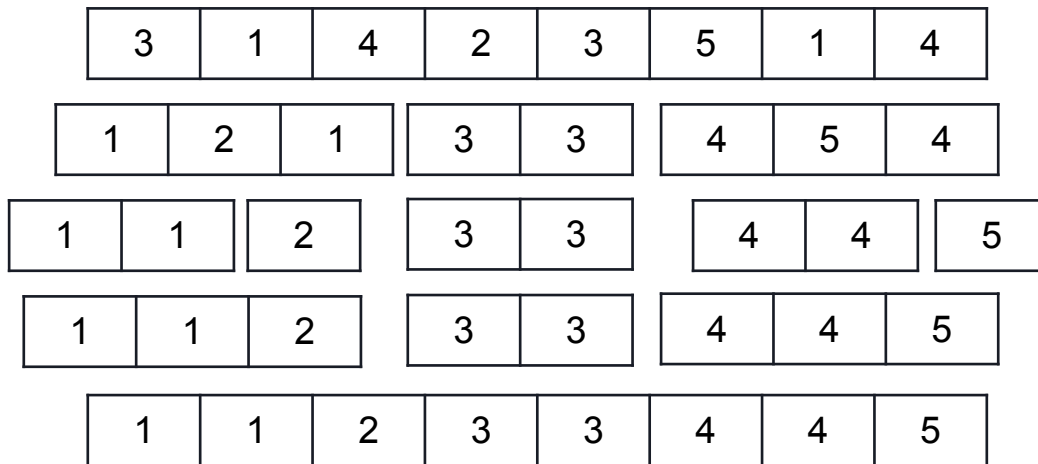
Sometimes the array will contain the same value(s) more than once. Practice diagramming Quicksort on such an array.

The quicksort algorithm works with duplicate values

For the same array, add all pivot value to it and mark them all as pivot

For all other values, partition works the same as normal

Use the `activities.txt` file to enter your solution

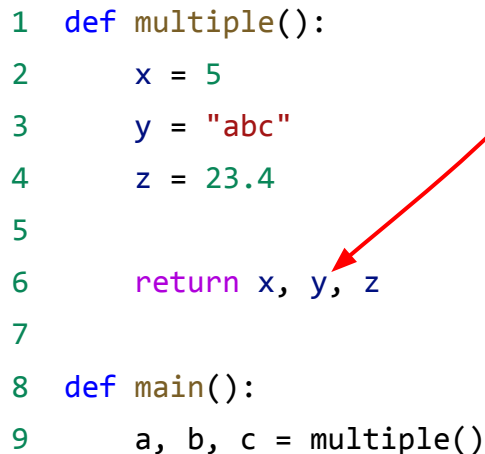


- Up until this point in the course the Python functions that we have written have only returned a single value.
 - The value that we **explicitly** return using a `return` statement, e.g. `return an_array`
 - In the event that we do not use a `return` statement, Python **implicitly** returns `None`
- Python functions may return **more than one value** as well.
- This is accomplished using a `return` statement followed by a comma-separated list of values.
 - e.g. `return x, y, z` will return all 3 values.
- When calling a function that returns multiple values, the values must be assigned to the same number of variables, again using a comma-separated list.
 - e.g. `a, b, c = my_function()`
- Assigning the return values to the wrong number of variables will result in a `ValueError` (wrong number of values to unpack).

Review: Multiple Return Values

A Python function may return multiple values using a `return` statement and a comma-separated list.

```
1 def multiple():
2     x = 5
3     y = "abc"
4     z = 23.4
5
6     return x, y, z
7
8 def main():
9     a, b, c = multiple()
```



When a function returns more than one value, the individual values may be assigned to the same number of variables using a comma-separated list.

7.27 The Partition Function

The first step in implementing the Quicksort algorithm is to partition the original array into three sub-arrays: one containing all of the values less than the pivot, one containing all of the values equal to the pivot, and one containing all of the values greater than the pivot. Let's write a helper function to do just that!

```
def partition(pivot, A):
    less_count = number of values less than pivot
    same_count = number of values equal to pivot
    more_count = number of values greater than pivot

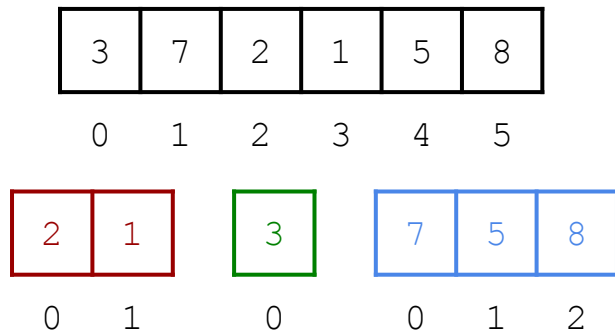
    less = Array(less_count)
    same = Array(same_count)
    more = Array(more_count)

    for every index i in A
        if A[i] < pivot
            less.append(A[i])
        otherwise if A[i] > pivot
            more.append(A[i])
        otherwise
            same.append(A[i])
    return less, same, more
```

- Open the `quicksort` module and define a new function named "partition" that declares a parameter for `an_array`.
- Use the pseudocode to the left as a guide to implement the partition function.
 - Sketch the algorithm out using comments.
 - Implement the parts that you know how to do.
 - Ask for help on the other steps!
 - **Hint:** you will need to use two loops: one to count the number of values that are less than, equal to, and greater than the pivot and a second one to copy the values into the three partitions.

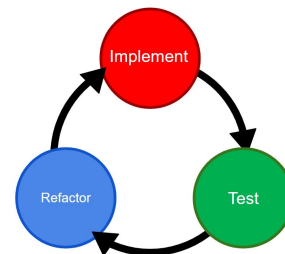
7.28 Testing Partition

The partition step is by far the most complex part of the quicksort algorithm, and so we will need to implement it carefully and make sure that it works under lots of different conditions. Let's write a few tests for the `partition` function to make sure that it works as intended.



- Open `quicksort_test` and create at least one test for each of the following `partition` scenarios:
 - $[1, 2] \Rightarrow [], [1], [2]$
 - $[3, 2, 4] \Rightarrow [2], [3], [4]$
 - $[3, 7, 2, 1, 5, 8] \Rightarrow [2, 1], [3], [7, 5, 8]$
- In each case, verify that the partitions contain the correct values in the correct order.

The partition function doesn't attempt to sort the values - it copies them in the same order that they originally appeared.



7.29 Array Concatenation

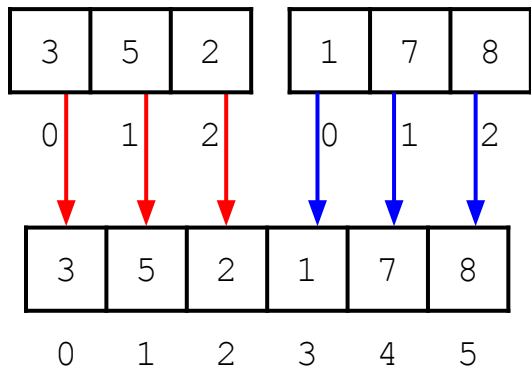
Our pseudocode algorithm uses the "+" operator to concatenate the `less`, `same`, and `more` arrays together, but that doesn't actually work with arrays! Before we can continue our `quicksort` implementation, we need a way to concatenate two arrays together.

```
define cat(A, B):  
    lengthA = len(A)  
    lengthB = len(B)  
    array = Array(lengthA + lengthB)  
  
    for every index i in A:  
        add A[i] to array  
  
    for every index i in B:  
        add B[i] to array  
  
    return array
```

- Open the `quicksort` module and define a new function named "cat" that declares parameters for `array1` and `array2`.
- We can't just add the two arrays together, so we will need to make a new array and copy the elements from the other two arrays. How big do we make the new array?
- The elements in `array1` should be copied into new array starting at index 0
- Where do the elements in `array2` get copied?
- Once all of the elements have been copied, return the new array.

7.30 Testing Concatenation

We'll need to verify that the concatenation function is working properly before we can implement the core Quicksort algorithm. Let's write a few tests to make sure that it works!



The `cat` function doesn't do anything fancy (like comparing the elements or trying to sort). It just copies all of the elements in the same order.

- Open `quicksort_test` and create at least one test for each of the following `cat` scenarios:
 - $[] + [1] \Rightarrow [1]$
 - $[1] + [] \Rightarrow [1]$
 - $[1] + [2] \Rightarrow [1, 2]$
 - $[1, 3] + [2, 5] \Rightarrow [1, 3, 2, 5]$
- In each case, verify that the array returned has the correct number of elements in the correct order.

7.31 Finishing Quicksort

Just as with Merge Sort, the most complicated parts of the Quicksort algorithm are handled by its helper functions. Once those are out of the way, implementing the core algorithm is fairly straightforward. Let's finish it now!

```
def quick_sort(A)
    if A has at most 1 element
        return A
    otherwise
        pivot = A[0]
        less, same, more = partition(pivot, A)
        sorted_less = quick_sort(less)
        sorted_more = quick_sort(more)
        return sorted_less + same + sorted_more
```

- Open the `quicksort` module and navigate to the `quicksort` function.
 - Use the pseudocode to the left to finish the core quicksort algorithm so that it works on arrays of length 2 or more.

Note that, because our `cat` function only works on two arrays at a time, you will need to call it twice to concatenate all three partitions together.

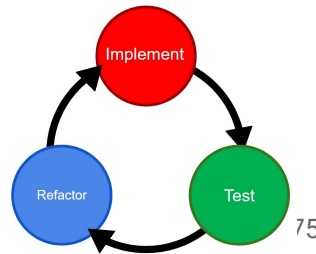
7.32 Testing Quicksort

We'll need to add a few more tests to the `quicksort_test` module to make sure that Quicksort is working for arrays with two or more elements.

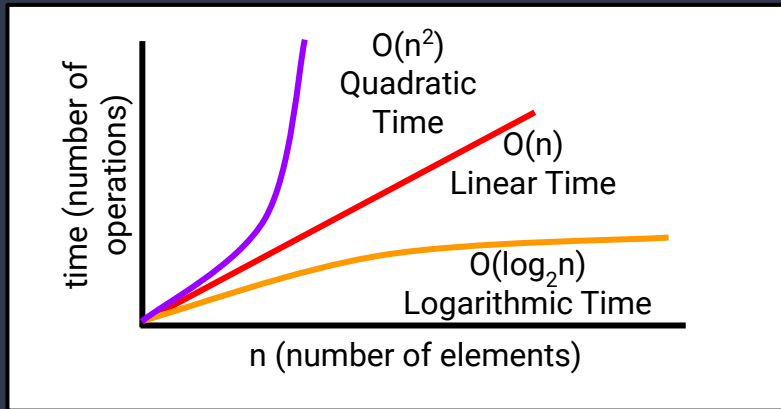
```
def quick_sort(A)
    if A has at most 1 element
        return A
    otherwise
        pivot = A[0]
        less, same, more = partition(pivot, A)
        sorted_less = quick_sort(less)
        sorted_more = quick_sort(more)
        return sorted_less + same + sorted_more
```

Once we have enough tests to verify that Quicksort works as intended, we've finished the algorithm!

- Open `quicksort_test` and create at least one test for each of the following `quicksort` scenarios:
 - An unsorted array with 2 elements.
 - An unsorted array with 5 or more elements.
 - A sorted array with 5 or more elements.
- Each test should verify that the array returned contains all of the elements in sorted order.
- Verify that all of your tests are passing before pushing your working code to GitHub.



Complexity Analysis



- Let n denote the original number of elements in A ,
- The time complexity, $T(n)$, of the quick sort is comprised of two phases:
 - The `partition` function
 - The `cat` function

Partition Complexity Analysis

- let n denote the original number of elements in A
- The **partition** function
 - uses one loop to count all of the values in A: **$O(n)$**
 - uses another loop to copy the elements from A into the less, same, and more arrays: **$O(n)$**
 - the number of operations within the loop is **constant**
 - therefore the partition part is **$O(2n)$** or just **$O(n)$**

```
def partition(pivot, A):  
    less_count = number of values less than pivot  
    same_count = number of values equal to pivot  
    more_count = number of values greater than pivot  
  
    less = Array(less_count)  
    same = Array(same_count)  
    more = Array(more_count)  
  
    for every index i in A  
        if A[i] < pivot  
            less.append(A[i])  
        otherwise if A[i] > pivot  
            more.append(A[i])  
        otherwise  
            same.append(A[i])  
    return less, same, more
```

Concatenation Complexity Analysis

- The `cat` function declares parameters for two arrays (A and B).
 - Let `lengthA` be the number of elements in A and `lengthB` be the number of elements in B.
- Let ***n*** denote the total number of elements in A and B: $n = \text{lengthA} + \text{lengthB}$
- The `cat` function must:
 - Allocate an array of size `n`: **$O(n)$**
 - Use loops to copy all of the values from A and B into the new array:
 $O(\text{lengthA}) + O(\text{lengthB}) = O(n)$
- The total complexity of the `cat` function is therefore $O(n) + O(n) = O(2n) = \mathbf{O(n)}$

```
define cat(A, B):  
    lengthA = len(A)  
    lengthB = len(B)  
    array = Array(lengthA + lengthB)  
  
    for every index i in A:  
        add A[i] to array  
  
    for every index i in B:  
        add B[i] to array  
  
    return array
```

Running Time Analysis

- Quicksort has at least three cases to consider.
- (Best Case) after partitioning, the `less` and `more` lists each have **about half** the elements from the original list.
 - We put **about half** of the elements in each list, each of which needs to be sorted.
 - Q: How many times can we cut the list in half before we reach the point where all lists have 1 element?
 - A: m times where $2^m = n$, or $\log_2 n$ times.
 - So we have to perform an $O(n)$ operation (partition) $\log_2 n$ times to get to lists of length 1.
 - Therefore $T(n) = O(n * \log_2 n)$

```
define quicksort(A)
    if A has at most 1 element
        return A
    otherwise
        pivot = A[0]
        less, same, more = partition(pivot, A)
        sorted_less = quick_sort(less)
        sorted_more = quick_sort(more)
        return sorted_less + same + sorted_more
```

The `partition` and `cat` operations both have a time complexity of **$O(n)$** .

In the **best case** each operation will need to be executed **$\log_2 n$** times.

7.33 Quicksorting Already-Sorted Data?

We know that some sorting algorithms perform better or worse depending on the configuration of the data. How does Quicksort perform if the data is already sorted? Let's diagram it to find out!

2	4	6	8	9	11	13	15
---	---	---	---	---	----	----	----

An already sorted list is a common edge case for sorting algorithms

Let's see how well quicksort handles it

Use the `activities.txt` file to enter your solution

7.33 Quicksorting Already-Sorted Data?

We know that some sorting algorithms perform better or worse depending on the configuration of the data. How does Quicksort perform if the data is already sorted? Let's diagram it to find out!

An already sorted list is a common edge case for sorting algorithms

Let's see how well quicksort handles it

Use the `activities.txt` file to enter your solution

2	4	6	8	9	11	13	15
2	4	6	8	9	11	13	15
2	4	6	8	9	11	13	15
2	4	6	8	9	11	13	15
2	4	6	8	9	11	13	15
2	4	6	8	9	11	13	15
2	4	6	8	9	11	13	15
2	4	6	8	9	11	13	15

Running Time Analysis

- Quicksort has at least three cases to consider.
- (*Worst case*) a list that is **already sorted** (or **reverse sorted**)
 - Partitioning is still **$O(n)$** (iterate over the list).
 - But this time one of the lists is empty and the other list contains $n-1$ elements.
 - Q: How many times will we need to partition if only one of the n elements is removed each time?
 - A: n times.
 - So we have to perform an $O(n)$ operation (partitioning) n times.
 - Therefore $T(n) = O(n) * n = O(n^2)$
- (*Typical case*) a list that is in some **random order**
 - Partitioning is still an $O(n)$ operation.
 - About half the elements will end up in less and more.
 - Therefore $T(n) = O(n \log_2 n)$ (same as Best Case).

```
define quicksort(A)
    if A has at most 1 element
        return A
    otherwise
        pivot = A[0]
        less, same, more = partition(pivot, A)
        sorted_less = quick_sort(less)
        sorted_more = quick_sort(more)
        return sorted_less + same + sorted_more
```

Quicksort *is* a divide and conquer algorithm, which usually suggests it will have good performance most of the time.

But the performance varies dramatically based on how well the **divide** step works (partition), i.e. how close each partition is to *half*.

Pivot Selection



Where we choose the pivot can make a big difference in performance for some arrays

- As we've seen, selecting the first value as the pivot can lead to problems with a sorted array
- Needing to sort an **already sorted sequence** is not as rare as you might think
 - Imagine a function that creates an array from a file and always sorts it
 - If the file was generated in order, then it will sort a sorted array
- We **arbitrarily** chose index 0 as are pivot
- Are there better choices?
- Some common alternatives are:
 - **last** index – $\text{len}(\text{array}) - 1$
 - **middle** index – $\text{len}(\text{array}) // 2$
 - **random** index

7.34 Pivot Race

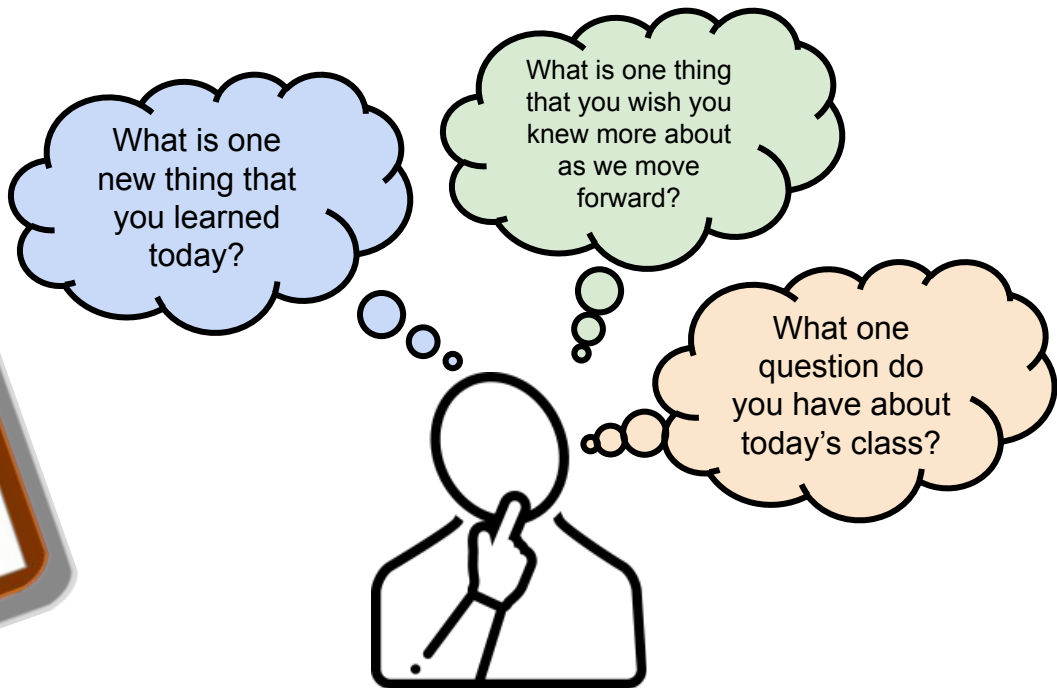
Quicksort performs better if roughly half of the elements end up in each of the less and more lists. Whether or not this happens depends on the configuration of the data *and* the selection of the pivot value. We've already tried using the value at index 0, now let's try some alternatives and see how they compare against each other.



We already know that choosing the first (or last) value in a sorted list performs poorly. What other ways might we choose a pivot?

- In the `quicksort` module, make two copies of the `quicksort` function.
 - Name one `quicksort_mid`
 - Name the other `quicksort_random`
- Change the index of the pivot to be the middle index and a random index, respectively
- In `main`, time (and optionally plot) how long it takes to sort a random and sorted array using all three pivot variants
- Which one did the best overall?

Summary & Reflection



Please answer the questions above in your notes for today.