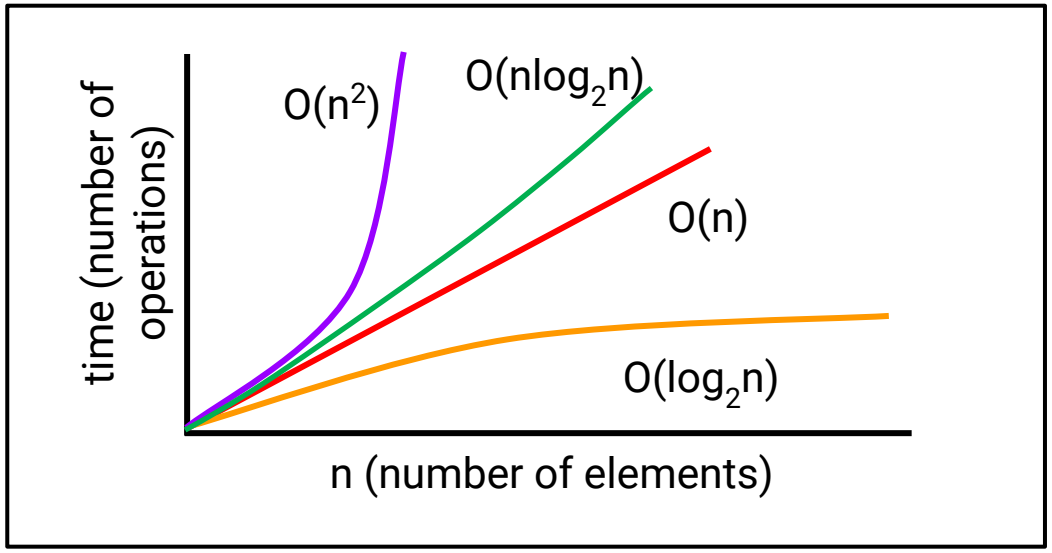
Problem 1

List the complexities for all the sorts we have written so far.

Use Big-O notation

Under what circumstances do each of the sorts experience their best/worst case. Why?



Insertion sort
Best: O(n)

Average: O(n^2)

Worst: O(n^2)

Merge sort
Best: O(nLogn)

Average: O(nLogn)

Worst: O(nLogn)

Quick sort
Best: O(nlogn)

Average: O(nlogn)

Worst: O(n^2)

Problem 2

Let A = [3, 2, 1, 5, 9, 4, 8]. You are going to sort it using the `partition` and `array_cat` functions you wrote in lecture.

Whenever you call `partition`, choose the pivot carefully that makes the accompanying quick sort code work.

```
Assume that A = [3, 2, 1, 5, 9, 4, 8]

less1, same1, more1 = partition(4, A)

less2, same2, more2 = partition(4, less1)
less_2_same2 = array_cat(less2, same2),
sorted_less1 = array_cat(less2_same2, more2)

less2, same2, more2 = partition(4, more1)
less2_same2 = array_cat(less2, same2),
sorted_more1 = array_cat(less2_same2, more2)

sorted_less1_same1 = array_cat(sorted_less1, same1)
sorted_A = array_cat(sorted_less1_same1 , sorted_more1)

print(sorted_A) // [1, 2, 3, 4, 5, 8, 9]
```

```
def quick_sort (an_array):
   if len (an_array) < 2:
      return an_array
   else:
      pivot = an_array[0]
      less, same, more = partition (pivot, an_array)
      new_array = array_cat(quick_sort(less), same)
      new_array = array_cat (new_array, quick_sort(more))
      return new_array


If you run quick_sort on sorted arrays of length 1000 or
more, it will probably crash I think…
```

Problem 3

We've seen some sorts are better under some circumstances than others. Insertion sort is not efficient in general, but it works very fast with arrays that have been already sorted or nearly sorted. Quick sort is efficient in general, but it behaves poorly on (nearly) sorted arrays.

Recall that the maximum recursion depth allowed is less than 1000. What would happen when you run `quick_sort` on sorted arrays of length 1000 or more?

Motivated with this observation, you are going to create a hybrid sort named `quick_insertion_sort` that enjoys best of each.

The new function is basically quick_sort since quick sort is usually better than insertion sort. But when it perceives that the `quick_sort` approach does not perform well (because the array is sorted), it will switch to `insertion_sort.` This will be another base case of the recursive hybrid sort.

How would you know if you've reached the right moment to change your strategy from quick sort to insertion sort? Implement your idea by modifying the `quick_sort` function to `quick_insertion_sort`.

```
def swap(an_array, a, b):
    temp = an_array[a]
    an_array[a] = an_array[b]
    an_array[b] = temp
    print(an_array)

def shift(an_array, index):
    while index > 0:
        swap(an_array, index, index-1)
        index -= 1

def insertion_sort(an_array):
    for index in range(len(an_array)):
        shift(an_array, index)

def quicksort(pivot, an_array):
    if len(an_array) < 2:
        return an_array
    else:
        less = arrays.Array(len(an_array))
        same = arrays.Array(len(an_array))
        more = arrays.Array(len(an_array))
        less_index = 0
        same_index = 0
        more_index = 0
        for i in range (len(an_array)):
            if an_array[i] < pivot:
                less [less_index] = an_array[i]
                less_index += 1
            elif an_array[i] > pivot:
                more [more_index] = an_array[i]
                more_index += 1
            else:
                an_array[i] == pivot
                same [same_index] = an_array[i]
                same_index += 1
        return array_utils.copy_array(less, less_index), array_utils.copy_array(same, same_index), array_utils.copy_array(more, more_index)
```