Problem Solving Session

- The remainder of today's class will comprise the ***problem solving session*** (***PSS***).
- Your instructor will divide you into ***teams of 3 or 4 students***.
- Each team will ***work together*** to solve the following problems over the course of ***20-30 minutes***.
  - You may work on paper, a white board, or digitally as determined by your instructor.
  - You will submit your solution by pushing it to GitHub before the end of class.
- Your instructor will go over the solution before the end of class.
- If there is any time remaining, you will begin work on your homework assignment.

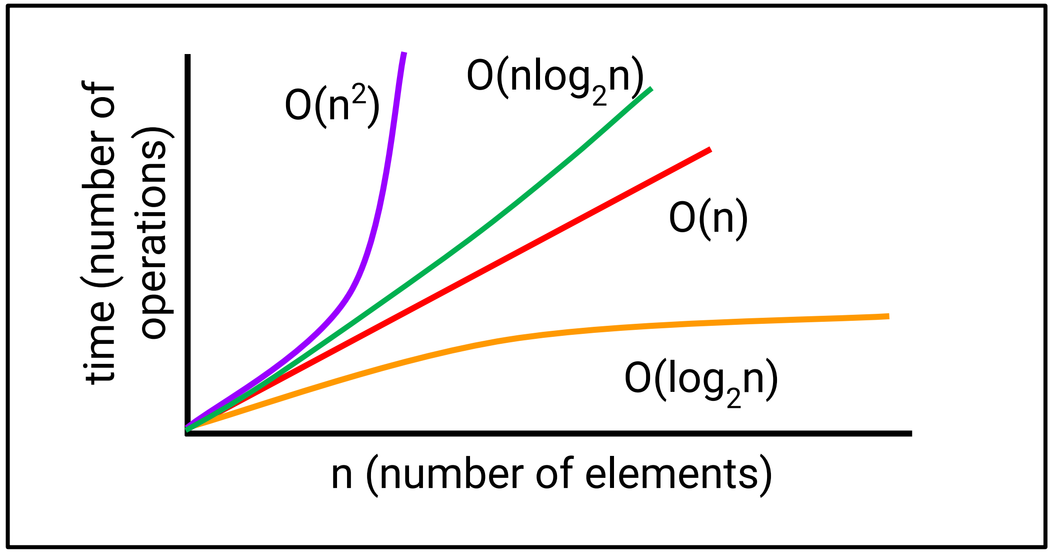
Class participation is a significant part of your grade (20%). This includes in class activities and the problem solving session.

Your Course Assistants will grade your participation by verifying that you pushed your solutions before the end of the class period each day.

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: **TOLD TO SKIP**

Average:

Worst:

Merge sort

Best:

Average:

Worst:

Quick sort

Best:

Average:

Worst:

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)
3 2 1   4  5 9 8
less2, same2, more2 = partition(2, less1)
less_2_same2 = array_cat(less2, same2),
sorted_less1 = array_cat(less2_same2, more2)
1 2 3  4  5 9 8
less2, same2, more2 = partition(8, more1)
less2_same2 = array_cat(less2, same2),
sorted_more1 = array_cat(less2_same2, more2)
1 2 3  4  5 8 9
sorted_less1_same1 = array_cat(sorted_less1, same1)
sorted_A = array_cat(sorted_less1_same1 , sorted_more1)
1 2 3 4 5 8 9
print(sorted_A) // [1, 2, 3, 4, 5, 8, 9]
```

Problem 3

```
def quick_sort (an_array):
   if len (an_array) < 2:
      return an_array
    elif len(an_array) > 990:
         for i in range(1,len(an_array)):
             shift(an_array,i)

   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
```

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`.