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.

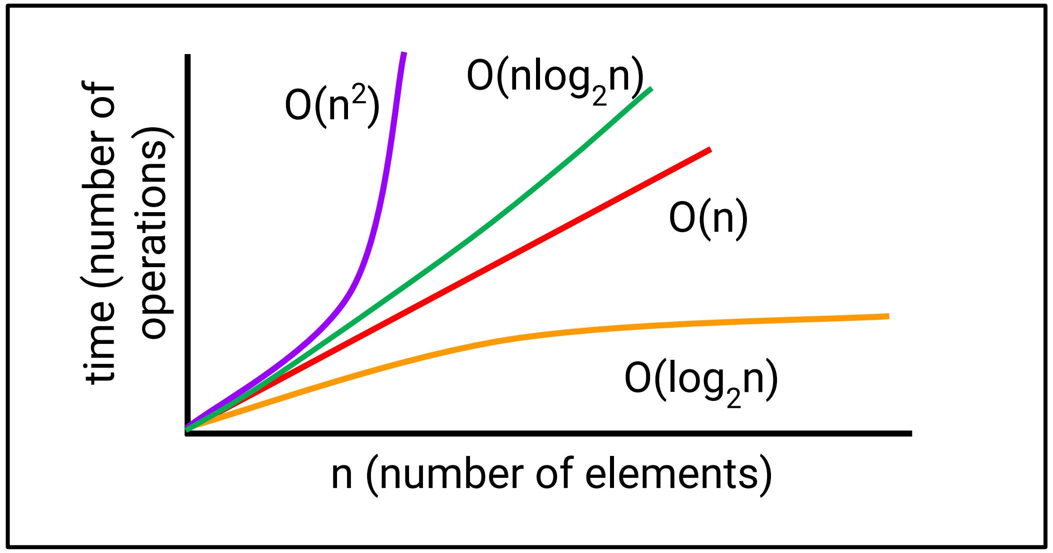
Class participation is a significant part of your grade (20%). This includes in class activities and the problem solving session.

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: $O(n)$ sorted

Worst: $O(n^2)$ reverse sorted

Merge sort

Every: $O(n*\log_2(n))$

Quick sort

Best useful case: $O(n*\log_2(n))$ less and greater partition are equal

Worst: $O(n\hat{}2)$ sorted or reverse sorted

Very best: $O(n)$ list is comprised of only the pivot value

Problem 2

Recall that the Quicksort partition function uses a pivot to divide a list into three sublists:

- less - less than the pivot
- same - equal to the pivot
- more - greater than the pivot

If an optimal pivot is chosen each time a list is partitioned, the unequal elements will be divided about equally between the less and more sublists.

Given the list L shown to the right, choose a pivot at each step so that the list is completely sorted by the time the code finishes executing.

Hint: you will not necessarily always choose the pivot at index 0.

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


less1, same1, more1 = partition(4, L)
less2, same2, more2 = partition(2, less1)
sorted_less1 = less2 + same2 + more2

less2, same2, more2 = partition(8, more1)
sorted_more1 = less2 + same2 + more2


sorted_A = sorted_less1 + same1 + sorted_more1


print(sorted_A)
```

```
def count_up(n): #The code will crash because 1000 is over
the recursion limit
  if n < 0:
    return 0
  else:
    rest = count_up(n-1)
    print(n)
    return n + rest
#modified version below
def count_up(n, stack = 1):
    if stack > 990: #990 instead of 1000 to be safe
        print(n)
        return n
    elif n < 0:
        return 0
    else:
        rest = count_up(n-1, stack = stack + 1)
        print(n)
        return n + rest
```

Problem 3

Consider the `count_up` function to the left. What will happen if `n>1000`? Write your answer as a comment.

Write a modified version of the function that stops if/when it is close to the maximum recursion depth in Python.

To be clear, the function should still print and sum as many values as possible without crashing the program. One possible output might look like this:

```
19049
19050
19051
19052
19053
… (many removed)
19996
19997
19998
19999
20000
sum: 18587324
```

Problem 4

When our implementation of Quicksort is used with (mostly) sorted lists and a pivot index of 0, the partition step places nearly all of the elements in the `more` list at each iteration. This causes the performance of Quicksort to devolve from the expected case of $O(n\log_2 n)$ to $O(n^2)$.

Python's maximum recursion depth is ~1000, meaning that trying to use Quicksort on (mostly) sorted lists with more than 1000 elements will cause a recursion error.

Insertion Sort is generally a slow sorting algorithm, but it excels when sorting (mostly) sorted data, running in O(n) time.

You will be creating a hybrid sorting algorithm that combines both Quicksort and Insertion Sort to attempt to leverage the best performance of both algorithms.

The new algorithm will be called `quick_insertion_sort`and it will start by trying to use Quicksort to sort a list. When the algorithm detects that Quicksort is not performing well (perhaps because the list is (mostly) sorted), it will switch to using Insertion Sort instead.

How will you determine that Quicksort is approaching its worst case performance and choose to switch to Insertion Sort? ***Hint***: if Quicksort is running near-optimally, the recursion depth will not far exceed $\log_2 n$ (where n is the length of the list).

```
import math
def quick_insertion_sort(a_list, n, stack = 1):
    “””where n = len(a_list)”””
    if len(a_list) < 2:
        return a_list
    elif stack > 2 * (math.log2(n)):
        return linear_sort(a_list)
    else:
        less, equal, greater = partition(a_list,
        a_list[0])
        return quick_insertion_sort(less,n,stack =
        stack + 1) + equal +
        quick_insertion_sort(greater,n, stack =
        stack + 1)
```