15110 Summer 2018
Problem Set 4

Name: _____

Andrew ID: _____

Instructions

- Type or neatly write the answers to the following problems.
- Save or scan this file as a pdf and submit to Gradescope

Exercises

1. (2.5 points) For this problem you will work with two mathematical functions, $f(n) = n^2 + 2n + 350$ and $g(n) = n^3 - 1000$. **Note: these are not Python functions and you should not be concerned with how to compute them.** You can think of them as functions that describe the running times of two different algorithms.

   a. Give the order of complexity for each function in big O notation. Use the **simplest** and **least** possible order of complexity for your answer. For example, if both $O(n)$ and $O(n-2)$ were possible answers, you would give $O(n)$ because it is simpler. And if both $O(n^4)$ and $O(n^3)$ were possible answers, you would give $O(n^3)$ because it is smaller.

   b. Which function, $f(n)$ or $g(n)$ has the higher (faster) rate of growth?

   c. Give a positive integer value n_1 such that $f(n_1) > g(n_1)$ and another positive integer value n_2 such that $g(n_2) > f(n_2)$. (Hint: try graphing the two functions for small positive integer values of n.)

   d. Explain why your answers to parts (b) and (c) are not contradictory; that is, how can we say that one function grows faster than the other, when sometimes f is larger and sometimes g is larger?

   e. Suppose we increase both $f(n)$ and $g(n)$ by multiplying by 1000, getting $f(n) = 1000n^2 + 2000n + 350000$ and $g(n) = 1000n^3 - 1000000$. Does the order of complexity of either $f(n)$ or $g(n)$ change? Explain briefly.

2. (1.5 points) For this question you will work with linear search algorithms. Below is a version of linear search in Python using a *while* loop.

```
def linear_search(items, key):
    length = len(items)
    i = 0
    while i < length and items[i] != key:
        i = i + 1
    if i >= length:
        return None
    else:
        return i
```

a. Consider the *instrumented* version of the function `linear_search` above, called `linear_search2(items, key)`. (This is called "instrumented" because counting the number of iterations and printing the result is analagous to inserting measuring instruments into a machine in order to get information about its functioning.)

```
def linear_search_2(items, key):
    num_iterations = 0
    length = len(items)
    i = 0
    while i < length and items[i] != key:
        num_iterations = num_iterations + 1
        i = i + 1
    print("Number of iterations: ", num_iterations)
    if i >= length:
        return None
    else:
        return i
```

Given the same inputs, which linear search function will consume more time? Does this change alter the **asymptotic time complexity** in terms of big O? Why or why not?

b. Suppose that we know the additional fact that the list is sorted in ascending order. For example, if our list has the values:

```
[1, 8, 20, 23, 39, 45, 56, 80, 90]
```

then if we want to search for the key 40 using linear search, we can stop when we reach 45 and return `None` because 40 cannot occur after 45 in a sorted list.

Here is a revised version of `linear_search` that returns `None` immediately as soon as it can be determined that the key cannot be in the list, assuming that the list is sorted in ascending (increasing) order.

```
def sorted_linear_search(items, key):
    length = len(items)
    i = 0
    while i < length and items[i] < key:
        i = i + 1
    if i >= length or items[i] > key:
        return None
    else:
        return i
```

Suppose that you call the `sorted_linear_search` function on four different lists with lengths 4, 10, 14, and 18 where the key is larger than all the elements of the lists. For example:

```
>>> sorted_linear_search([2, 7, 10, 13], 100)
>>> sorted_linear_search([2, 7, 10, 13, 14, 15, 16, 23, 32, 35],
100)
>>> sorted_linear_search([2, 7, 10, 13, 14, 15, 16, 23, 32, 35,
40, 57, 61, 65], 100)
>>> sorted_linear_search([2, 7, 10, 13, 14, 15, 16, 23, 32, 35,
40, 57, 61, 65, 70, 71, 75, 89], 100)
```

Plot a graph so that the *x* axis of your graph shows the number of items in the list and the *y* axis shows the number of iterations that your function makes. That is, you need to have four points in your graph whose *x* coordinates are 4, 10, 14, and 18 respectively. Do you observe a straight line or a curve?

c. In general, if the list has *n* elements, what is the number of iterations that would be made **in the worst case** for `sorted_linear_search`? Express your answer using big O notation and explain your reasoning.

3. (4 points) We can apply a powerful design idea called *Divide-and-Conquer* to the problem of searching a sorted list, using an algorithm called binary search. The basic idea is to find the middle element. Then, if that is not the key, you search either the first half of the list or the second half of the list, depending on the half that could contain the key. The process is repeated until we either find the key or we run out of elements to examine.

   Here is an implementation of binary search in Python using iteration (later in the class you'll work with another version using recursion):

```
def bsearch(items, key):
    min = 0
    max = len(items) - 1
    while min <= max:
        mid = (min + max) // 2
        if items[mid] == key:
            return mid
        if key > items[mid]:
            min = mid + 1
        else:
            max = mid - 1
    return None
```

   Let `items = ["Anna", "Dan", "Ella", "Finn", "Gina", "Ivan", "Karen", "Luke", "Mary", "Nadia", "Oliver", "Perry", "Russell", "Tom", "Ziv"]`.

   a. Trace the function above for each of the function calls shown below, showing the values of `min` and `max` after each iteration of the `while` loop is completed. Also write down the value returned by the function. We have started the trace with the initial values of `min` and `max`:

```
bsearch(items, "Nadia")

   min     max
   --------------
      0      14
```

```
bsearch(items, "Dan")

   min     max
   --------------
      0      14
```

```
bsearch(items, "Indira")

   min     max
   --------------
      0      14
```

b. Suppose that you call the `bsearch` function on four different sorted lists with lengths 4, 10, 14, and 18 where the key you are looking for is larger than any element in the lists. (Example: `bsearch(["Aung", "Ben", "Drew", "Eileen", "Felicia", "Niki"], "Norbert")`)

   Plot a graph so that the *x* axis of your graph shows the number of items in the list and the *y* axis shows the number of iterations that your function makes. That is, you need to have four points in your graph whose *x* coordinates are 4, 10, 14, and 18 respectively. Do you observe a straight line or a curve? Is the growth rate you see here faster or slower than the one you saw in question 2(b)?

c. Using linear_search_2 (above) as a model, instrument the `sorted_linear_search` and `bsearch` functions we gave you above and run them on lists of length 10,000 and 100,000 items, searching for keys that are larger than any key in the lists. You can easily generate a list of 100,000 numbers using the following:

```
>>> nums = list(range(100000))
```

   How many iterations did each function perform for each search?

d. Which of the two search algorithms, linear or binary search, do you think should be used to search a very large sorted list, for instance, a list of the names of all citizens of the US. Explain your choice. We don't expect you to analyze the binary search algorithm's big O complexity measure, but compare the experiments you made and extrapolate from what you see.

4. (2 points) Suppose the National Security Agency has obtained an **unsorted** list of phone calls made in the U.S. in the past year (so there are a lot of them.) Jane Analyst has the task of working with this data, but she is unethical, and would like to expunge the record of a particular phone call she made before doing the analysis and writing her reports. Let's call the list `call_records`.
    a. Which one of the search functions discussed in this homework should Jane use to look for her call in the list `call_records`? Why?

    b. Suppose that Jane has another unsorted list, called `ATT_call_records`, of calls originated by customers of ATT.. There are about two million calls on this list, and about eight million on the list `call_records`. If she sorts each list using insertion sort, and it takes 30 seconds to sort `ATT_call_records`, about how long would you expect it to take to sort `call_records`? Use whole numbers, we just want an approximation. Explain your answer.

    c. Would it make sense for Jane to sort these lists using insertion sort before searching them for her call? Why or why not?

d. What if Jane is really, really unethical, and wants to offer a service where she will expunge anyone's call for a fee? Should she sort the lists using insertion sort before searching them? Why or why not?