Problem 1

Give the time cost of each line of code or snippet, using big-Oh notation. Assume that \( n \) represents the length of the list, or other data structure.

Some of these require some thought about what Python has to do. If you are uncertain about your answer, then write down your thoughts - then ask! We'll discuss common questions in class.

\[ x = 10 \]
\[ O(1) - \text{assignment is a simple operation} \]

\[ a = b \quad \# \text{do we need to know what type } b \text{ is???} \]
\[ O(1) - \text{assignment is a simple operation. It doesn't matter what type } b \text{ is.} \]

\[ \text{some_list.append(100)} \]
\[ O(1) \]

\[ \text{foo = other_list[1:]} \]
\[ O(n) \quad \textbf{(CORRECTED)} - \text{slicing is } O(n), \text{ because it has to copy } O(n) \text{ many elements. (Of course, if the slice is a constant size, then the cost of the slice is } O(1).) \]
for i in some_list:
    print(i)
O(n) - we iterate over the list, and print each element. (The real cost of print() varies based on the type of the object. But we'll often pretend that it is O(1).)

Problem 2

For each function below, give the time cost of each individual line of code (for one iteration of the line); write it next to the line (in big-Oh notation). Then, in a second column, write how many times that line of code runs (also in big-Oh). Assume that n is the size of the data passed to the function - such as the length of a list.

```python
def pretty_print_list(vals):
    print("{")
    O(1)  1
    first = True
    O(1)  1
    for v in vals:
        O(1)  n
        if not first:
            O(1)  n
        print(",")
        O(1)  0
        first = False
        O(1)  0
    print(v)
    O(1)  n
print(""}")
O(1)  1
```

NOTE: I got the test backwards in the if() statement above. As it is, the if() condition is never true, so the body never runs. If we fix the bug (removing the ‘not’), then it would only run once - since we would change ‘first’ to False.

def find_dups(vals):
    for i in range(len(vals)):
        O(1)  n
        for j in range(len(vals)):
            O(1)  n^2
            if i != j:
                O(1)  n^2
            if vals[i] == vals[j]:
                O(1)  n^2
print(i, j)  O(1)  \( n^2 \)
def bubble_sort_is_terrible(vals):
    for loop_count in range(len(vals)):
        for i in range(len(vals)-1):
            if vals[i] > vals[i+1]:
                vals[i],vals[i+1] = vals[i+1],vals[i]

Problem 3

For each function below, give the time cost of each individual line of code in big-Oh notation. This time, though, give the total cost for that line, across all iterations. (That is, an O(1) line, which runs O(n) times, should be listed as O(n).)

Make sure to account for the cost of other functions that these functions call!

def max_min(vals):
    assert len(vals) > 0
    max = vals[0]
    min = vals[0]
    other_vals = vals[1:]
    for v in other_vals:
        if v < min:
            min = v
        if v > max:
            max = v
    return (min,max)
def wrapper(vals):
    bigOh_1_func(vals)  # O(1)
    for v in vals:  # O(n)
        bigOh_n_func(vals)  # O(n^2)

def list_to_str(vals):
    retval = ""  # O(1)
    for v in vals:  # O(n)
        new_retval = retval + str(vals)  # O(n^2)
        retval = new_retval  # O(n)
    return retval  # O(1)

NOTE: Calling \texttt{str()} on a list has cost \(O(n)\) (at least), since there are at least \(O(n)\) things that have to be printed out.
Problem 4

Both functions below run in O(n) time. However, from a casual inspection, you might think that they run in O(n^2). Explain why they run in O(n).

If you don’t know the answer, that’s OK - ask for help! (And write down as much as you know.)

def func1(vals, val1, val2):
    pos1 = -1
    pos2 = -2
    for i in range(len(list)):
        if vals[i] == val1:
            pos1 = i
        if vals[i] == val2:
            pos2 = i
    if pos1 < pos2:
        slice = vals[pos1:pos2]
        for v in slice:
            print(v)

In this function, we have two for() loops, but they are not nested inside each other. Thus, what we have is a loop which takes O(n) time, followed by a second loop, which also takes O(n) time.
def func2(vals):
    for i in range(len(vals)):
        if vals[i] > 0:
            for v in vals[:i+1]:
                print(v)
            break

This looks like it ought to be $O(n^2)$, because we have a for loop inside another for loop. However, the ‘break’ changes things - if the second loop runs, then we will (after the second loop completes) break out of the first loop. There are a couple of ways to think about this.

One way to think about it is that there are, really, two loops: a “search” loop that runs first, followed by a “print” loop, which runs later. You can rewrite the function into this equivalent form:

def func2(vals):
    pos = -1
    for i in range(len(vals)):
        if vals[i] > 0:
            pos = i
            break
    if pos != -1:
        for v in vals[:pos+1]:
            print(v)

Another way to think about it is to count the number of times that the inner loop can run. When after the inner loop runs once, we break out of the outer loop - and thus, it is impossible for the inner loop to run more than once. Thus, the total cost for the inner loop is $O(n)$ - **even though** it’s inside another loop.