Tutorial 3

Problem 1.

Consider the Binary Search algorithm below:

```
PreCondition: data must be sorted

binarySearch(list, target)
  set position to -1   (TARGET_NOT_FOUND)
  set targetNotFound to true
  if list not empty
    while targetNotFound AND have not looked at or discarded
                                          every element of list
      find middle element of list
      if middle element == target
        set position to position of target in original list
        set targetNotFound to false
      else
        if target < middle element
          list = first half of list
        else
          list = last half of list
  return position
```

a) Hand trace the algorithm while we are searching for the **target 4** in the following data list:

-27 -14 -2 -1 2 3 4 6 8 9 13 24 35

Let's show our work, i.e., for each iteration of the algorithm, show the middle element, the partition (section of the list) we are discarding and the one we are keeping.

Which scenario does the above situation represent: best, average or worst case scenario?

b) Hand trace the algorithm while we are searching for the **target 13** in the following data list:

-7 -4 -2 -1 2 3 4 6 8 9 13 24 35

Again, let's show our work.

Which scenario does the above situation represent: best, average or worst case scenario?

c) Hand trace the algorithm while we are searching for the **target 5** in the following data list:

-1 -27 24 -2 3 35 -14 6 13 9 8 2 4

Do you have any observations?

Note: Download and execute the two binary search programs (BinarySearch_Iterative and BinarySearch_Recursive) posted on our course web site. Make sure you become familiar with the code of these two Python programs.

Problem 2.

Exercise 9.1. from our online textbook:

Write a Python program that reads `words.txt` (http://greenteapress.com/thinkpython2/code/words.txt) and prints only the words with more than 20 characters (not counting whitespace and newline character). Our Python program must print one word per line.

Problem 3.

In our Tutorial 2 Problem 8, we discovered that the function below only looks at the first letter in **`s`** and either returns **`True`** if this letter is a lowercase letter, or **`False`** otherwise.

In this problem, we are asked to rewrite the following function considering all the good programming style guidelines (GPS) we have seen this semester (have a look at the GPS web page on our course web site). Amongst other things, this signifies that we need to rewrite the function such that it has only 1 single `return` statement (at the end of the function body). We are also asked to fix it such that it actually does check whether a string contains any lowercase letters (this, after all, is its purpose).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False
```

What kind of pre-condition can we add to this function? A pre-condition is a condition that must be true before the function is called. Often, pre-conditions are about the state of a function's parameters. So, what must always be true about this function's parameter in order for this function to function ☺.

Problem 4.

From page 75 of our online textbook:

What does the following function do?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

This pattern of computation, i.e., traversing a sequence and returning when we find what we are looking for, is called a search. We have seen 2 searching algorithms this semester. Which searching algorithm does the above function `find( )` implement?

Why is `find( )` seen as the opposite of the **indexing [ ]** operator? To understand this question and hence to answer it, we may want to explain to ourselves what `find( )` does and what the **indexing [ ]** operator does. As we do so, the answer will become obvious!

Let's rewrite the above function such that it uses the more appropriate `for` loop. Why is the `for` loop more appropriate than the `while` loop in the above function?

Let's modify the `find( )` once again (the one with the `for` loop) such that it has only one `return` statement (the one at the end of the function body).

Let's modify the `find( )` one more time (the one with the `for` loop and 1 `return` statement) by adding a third parameter, namely the index in `word` where it should start searching. Let's make sure we use a descriptive name when we name this 3rd parameter.

Problem 5.

Problem 7.2 at page 69 of our online textbook:

Exercise 7.2. *The built-in function* `eval` *takes a string and evaluates it using the Python interpreter. For example:*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

Write a function called `eval_loop` *that iteratively prompts the user, takes the resulting input and evaluates it using* `eval`*, and prints the result.*

It should continue until the user enters `'done'`*, and then return the value of the last expression it evaluated.*

For example, if the user enters "`45 + 98`" at the prompt the `input(…)` function prints on the screen, this `input(…)` function returns "`45 + 98`" as the resulting string, which our function could assigns to a variable called `resultingString`. Then our function calls `eval(resultingString)` and prints its result, i.e., `143` on the computer monitor screen.

Let's make sure that our Python program, containing our function `eval_loop( )` is robust, i.e., it does not crash if the user enters a nonsensical equation or nothing at all.

Problem 6.

Build test cases need to completely test the function `eval_loop( )` we created in the Problem 5. Let's make sure our text cases have 2 sections: a "Test Data" section and an "Expected Results" section.

Remember: **To completely test** means that all statements of our Python code fragment have been executed, i.e., tested by all our test cases. This does not mean that each test case executes all the statements. That would be impossible. It means that once all the test cases have been used to test our Python code fragment, they have execute all its statements.

Problem 7.

From our online textbook:

8.7 Looping and counting

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a **counter**. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result—the total number of a's.

As an exercise, encapsulate this code in a function named count, and generalize it so that it accepts the string and the letter as arguments.

And returns count to the caller (the code that called our function).

As a second exercise, rewrite the function so that instead of traversing the string, it calls the "three parameter" version of `find( )` from our Problem 4.

Problem 8 - Challenge.

What is the time efficiency of each of the following Python code fragments. Express this time efficiency using one of the Big O notations seen in class (using one Big O notation per Python code fragment). The critical operation is "+", i.e., the addition.

a)
```
y = y + 1000
```

b)
```
for each in range(n)
    x = x + y
    y = y + 1000
```

c)
```
count = 0
for each in range(n)
    count = count + 10
for each in range(n)
    count = count + each
```

d)
```
count = 0
for each in range(n)
    for element in range(n)
        count = count + 10
```