Python Basics

1. Although most in the industry will be reasonably forgiving when it comes to coding by hand, you should still be comfortable with the syntax of your language of choice.

   (a) Identify and fix the line(s) that have invalid syntax in the following code:
   If you are unsure about a specific line, just write a brief explanation of the error. Assume that incorrect lines do not affect the validity of other lines which may depend on them.

```
1  include math
2
3  int a = 0
4  b = 0
5
6  def function(arg):
7      return (arg + 2)
8
9  c = "hello"
10 sum = b + math.sqrt(c)
11 total = a + (b ** 0.5)
```

   1 This line should read "`import math`", or, alternatively, "`from math import *`".
   3 Python is a dynamically-typed language, so you cannot specify a variable's type in this manner.
   10 This line will cause a runtime error because `sqrt` tries to perform mathematical operations on a string, which is invalid.

   (b) Which of the following *are* keywords or standard functions in Python?

```
import     in     from     class     define     int     str     String
True     false     where     while     or     not     isinstance     new
print     range     float     char     bool     xor     sum
```

   All of these are valid, except `define`, `String`, `false`, `where`, `new`, `xor`, `char`.

General Recursion

2. Sally is being plagued by an army of lookalike suiters, each of which presents her with an enticing but unbearable meal upon arrival. The dishes all smell amazing, however, so she can't help but try each one. A dish will never fail to disappoint her, but fortunately, some of the suiters shared recipes and created identical concoctions. Once Sally tastes a meal once, she can immediately smell out instances of the same dish and send their bearers away.

   (a) Write a Python function that, given a list of "dishes," returns a list of the rejected dishes in the order they failed to fool her. (e.g. $[1, 2, 2, 3, 3, 3]$ would return [2,3,3])

```
def find_dupes(lst):
    result=[]
    seen=[]
    for member in lst:
        if member not in seen:
            seen.append(member)
        else:
            result.append(member)
    return result
```

(b) If your solution to 2a was iterative, write it recursively; if it was recursive, write it iteratively.

```
def find_recur(lst, seen=[]):
    if len(lst)==0:
        return []
    if lst[0] not in seen:
        seen.append(lst[0])
        return find_recur(lst[1:], seen)
    return [lst[0]]+find_recur(lst[1:], seen)
```

or

```
def find_tail(lst, result=[], seen=[]):
    if len(lst)==0:
        return result
    if lst[0] not in seen:
        seen.append(lst[0])
    else:
        result.append(lst[0])
    return find_tail(lst[1:], result, seen)
```

(c) What is the time complexity of your approach? Why?

$O(N^2)$, since the `seen` list must be searched linearly

3. Write a **recursive function** that reverses a string (e.g. "Racecar" yields "racecaR").

```
def reverse( string ):
    if string == '' :
        return string
    else:
        return reverse( string[1:] ) + string[0]
```

Other solutions are possible.

4. Perform a substitution trace on `reverse('Doge')`.

```
reverse('Doge') =   reverse('oge') + 'D'
                =   (reverse('ge') + 'o') + 'D'
                =   ((reverse('e') + 'g') + 'o') + 'D'
                =   (((reverse('') + 'e') + 'g') + 'o') + 'D'
                =   ((('' + 'e') + 'g') + 'o') + 'D'
                =   'egoD'
```

Files/IO

5. Write a function that takes in a file name, and returns the average size of a word in that file. Assume the files will only have 1 word per line, for example:

No
soup
for
you!

which has an average length of: 3.25
Assume a function `len( str )` *which returns the length of a string is provided.*

```
def average_wordlength(filename):
    characters = 0
    words = 0
    for line in open(filename):
        words += 1
        characters += len(line)
    return characters/words
```

Tail Recursion

6. Below is a function that, given a lower bound i, upper bound n, and function from integers to integers f, computes $\sum_{k=i}^{n} f(k)$.

```
1    def series_sum(i, n, f):
2        if(i == n):
3            return f(n)
4        return f(i) + series_sum(i + 1, n, f)
```

   (a) Invoke the function to compute $\sum_{k=1}^{5} k^2$.

```
series_sum(1, 5, lambda x: x**2)
```

OR

```
def square(x): return x**2
series_sum(1, 5, square)
```

   (b) Rewrite the `series_sum` function to be tail recursive.

```
1    def series_sum(i, n, f, s=0):
2        if(i == n):
3            return s + f(n)
4        return series_sum(i + 1, n, f, s + f(i))
```

   (c) What is the advantage of the new implementation?

   Tail recursive functions make no additional calculations after the execution of the recursive call completes; thus, they do not necessitate the preservation of stack frames belonging to intermediate recursive invocations. Provided the language implementation supports sufficient optimization (Note: Python does not), there is no more runtime memory overhead for calling a tail recursive function that makes 100 recursive calls as compared to one that makes none at all. The true benefit becomes apparent when the recursion depth becomes much deeper: a tail recursive function can handle a recursion depth in the millions, which would cause a non–tail recursive one to overflow the stack.

Greedy Algorithms

7. Given that an algorithm is *greedy*, is it guaranteed to return an *optimal* solution?

   NO. Greedy algorithms always choose the *current* best solution, which is not necessarily the *overall* best solution!

Structures

8. For the sake of this question, you find yourself to be the head programmer under Kim Jong Un's glorious reign. It also just so happens that a nation-wide track meet is being held today. Thus, the glorious leader has demanded that you write a program to keep track of information relating to all track runners present at the event.

   (a) Write a struct called `TrackRunner` to keep track of each competing runner. You will need to store each runner's `name` (a string), `age` (an int), and `fastestTime` (an int).

```
from rit_lib import *
TrackRunner = struct_type("TrackRunner",
    (str,'name'), (int,'age'), (int,'fastestTime'))
```

   (b) The glorious leader has decided that, on this day, no runner named "Joe" may win gold. Given a list of `TrackRunner` structs, write the function `aWinnerIsYou(runners)` that returns the runner in the list `runners` with the fastest time whose name is not "Joe". Use this function to find the runner's name, age, and fastest time and print those results.

```
def aWinnerIsYou(runners):

    best = None
    for i in range(len(runners)):
        curr = runners[i]
        if curr.name != 'Joe':
            if best == None:
                best = curr
            elif best.fastestTime > curr.fastestTime:
                best = curr
    return best

winner = aWinnerIsYou(runnersLst)
print(winner.name, winner.age, winner.fastestTime)
```

Stacks and Queues

9. How can you use a stack to check if an input string of exclusively parentheses, brackets, and braces is properly balanced? (e.g. "[{}]" is accepted, but "(()" and "[(])" are not.)
Assume that you are provided with a `Stack` structure that has `push()`, `pop()`, and `peek()` functions.

```
def closing_match(char):
    if char == '(':
        return ')'
    elif char == '[':
        return ']'
    elif char == '{':
        return '}'
    elif char == '<':
        return '>'
    else:
        return None

def delims_are_balanced(inString):
    '''
    String -> boolean

    Determines if the input string has balanced delimiters.
    Delimiters are () [] {} and <>.

    '''
    stack = Stack()
    for char in inString:
        if closing_match(char) != None: # a starting grouping symbol
            stack.push(char)
        else:
            if stack.is_empty():
                return False
            elif char == closing_match(stack.peek()):
                stack.pop()
            else: # mismatched grouping symbol
                return False
    return stack.is_empty()
```

Searching

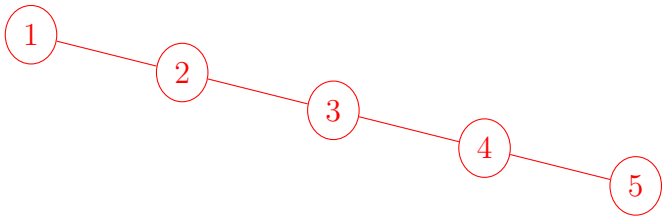
10. Given the sorted list [1, 4, 9, 16, 25, 69, 420, 1337], write out the steps that a binary search would take to find the number 69.

```
[1,4,9,16,|25|,69,420,1337]
[1,4,9,16,25,69,|420|,1337]
[1,4,9,16,25,|69|,420,1337]
```

Trees

11. (a) If we have a **balanced** binary search tree containing n nodes:
    i. What is its height?
    $\log_2(n)$
    ii. How much time would it take to traverse to any of the leaf nodes of this tree?
    $\log_2(n)$
    iii. What's the worst case search time for an unbalanced search tree?
    n

    (b) In terms of runtime efficiency, what is the worst possible configuration (e.g. arrangement of nodes and their children) of a binary search tree? Sketch a small example of what such a configuration looks like.
    A perfectly *imbalanced* tree, in which all nodes have one or 0 children.



    (c) Based on your answer above, what data structure is such a BST reminiscent of?
    A perfectly imbalanced tree as described above operates like a **linked list**.

12. (a) Write a binary tree program that defines the following:

- a `TreeNode` struct with the given properties:
  - `data` (an object) : the data in this node
  - `left` : the `TreeNode` head of the left subtree (or `None`)
  - `right` : the `TreeNode` head of the right subtree (or `None`)
- a `Tree` struct with the given properties:
  - `size` (an int) : the size of the binary tree
  - `head` (a `TreeNode`) : the head node
- a recursive function `is_in_tree(head, element)` that determines whether the binary tree with the given head contains the specified element

```
1 from rit_lib import *
2
3 TreeNode = struct_type("TreeNode",
4     (('TreeNode',NoneType),'left'),
5     (('TreeNode',NoneType),'right'))
6
7 Tree = struct_type("Tree", (int,'size'), (TreeNode,'head'))
8
9 def is_in_tree(head, element):
10     if head == None:
11         return False
12     elif head.data == element:
13         return True
14     elif head.data < element:
15         return is_in_tree(head.right, element)
16     elif head.data > element:
17         return is_in_tree(head.left, element)
```

(b) Give an example of how you could test your code.

```
newTree = Tree(3, TreeNode(3, TreeNode(1, None, None),
        TreeNode(4, None, None)))
print(is_in_tree(newTree.head, 1)) # -> True
print(is_in_tree(newTree.head, 3)) # -> True
print(is_in_tree(newTree.head, 5)) # -> False
```

Sorting

13. Fill in the table for the asymptotic running time of each sorting algorithm.

	Best	Worst	Average
Merge sort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$
Quicksort	$O(n\log(n))$	$O(n^2)$	$O(n\log(n))$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$

14. Below is Python code for a function that performs an insertion sort and prints `data` after each iteration of the `for` loop.

```
1 def insertion_sort( data ):
2     for marker in range( 1, len( data ) ):
3         val = data[marker]
4         i = marker
5         while i > 0 and data[i-1] > val:
6             data[i] = data[i-1]
7             i -= 1
8         data[i] = val
9         print( data )
```

(a) Write out what the function will print for the input list: [3,2,7,1].

```
[2,3,7,1]
[2,3,7,1]
[1,2,3,7]
```

(b) What is the sort algorithm's time complexity?
$O(N^2)$

15. In what scenario does Quicksort experience its worst-case time complexity? You may assume that we always pick the first element as the pivot.

Data that is (nearly) sorted or is (nearly) sorted in reverse order.

16. What causes Quicksort to run so slowly on the input you describe in the last question?

Quicksort splits its input into two lists based on the value of the pivot. If the pivot is either the smallest or the largest element, then one list will only have no elements, while the others will have all of the elements except the pivot.

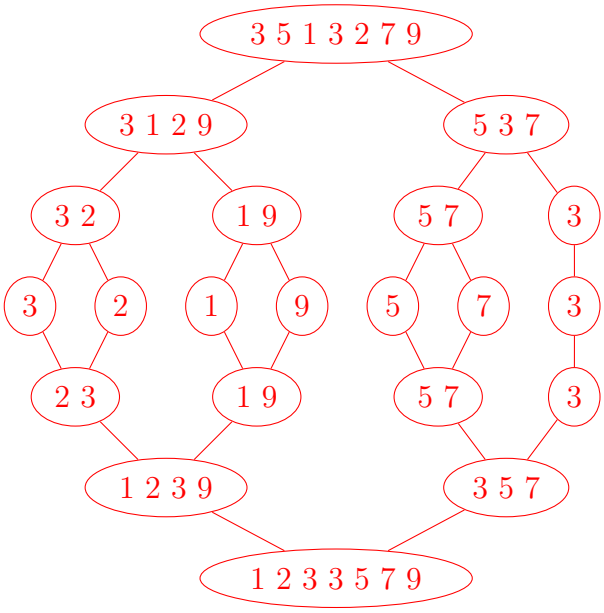
17. In Quicksort, why should we select a random pivot value, rather than always pivoting on, for example, the first or last element?

With real-world data, we're more likely to encounter ordered or semi-ordered data than randomized data. This makes it more likely for us run into Quicksort's worst-case time complexity. We run into this bad time complexity if we select pivots which are near the lowest or highest values.

Selecting a random value to pivot on helps us encounter the average case evens out the distribution of ordered and unordered data. Even if we're getting in sorted data, if we select pivots randomly, we should be able to end up with average time complexity.

18. Show the stages of a merge sort and a quicksort on the following list: [3,5,1,3,2,7,9]. Be sure to identify your pivot.

Merge sort:



Quicksort (using the first element in the list as a pivot):