1. **(Functions)** Turtles are capable of drawing equilateral polygons.

- Draw a straight line 4 times, turning 90 degrees between each line to create a square.
- Draw a straight line 6 times, turning 60 degrees between each line to create a hexagon.
- Turtles draw circles by drawing very short lines and turning a very small amount each time.

Ignoring the built-in circle function, write the code for your own `drawCircle` function to have a turtle draw a circle for a given radius by using line segments.

2. **(Recursion)** A number \(a\) is a power of \(b\) if it is divisible by \(b\) and \(a/b\) is a power of \(b\). Write a function, `isPower` that takes parameters \(a\) and \(b\) and returns `True` if \(a\) is a power of \(b\) and `False` otherwise. Assume it is a precondition of this function that \(a\) and \(b\) are both greater than 0.

   a.) Write the function using **recursion**.
   
   b.) Show a substitution trace for `isPower(27, 3)` and `isPower(40, 4)`.
   
   c.) Write the function again **iteratively** using a loop.

3. **(Recursion)** Horner's Scheme is a method for efficiently evaluating a polynomial. It takes a list of coefficients that represent the polynomial, and a value for \(x\). For example, `horner([1, 4, 3], 5)` would return 96 because we are evaluating \(3x^2 + 4x + 1\) for when \(x = 5\). Show the substitution trace for this example.

   ```python
   def horner(polyList, x):
       if len(polyList) == 0:
           return 0
       else:
           return polyList[0] + x * horner(polyList[1:], x)
   ```

4. **(Recursion)** Show the output when `factorial(3)` is invoked.

   ```python
   def factorial(N):
       asterisk = ''
       for i in range(N):
           asterisk += '*
       print(asterisk + "factorial " + str(N))

       if N == 0:
           print(asterisk + "returning " + str(1))
           return 1

       prev = factorial(N-1)
       cur = N * prev
       print(asterisk + "returning " + str(cur))
       return cur
   ```
5. **(Strings)** An anagram is a word formed by rearranging the letters of another word. For example, angel and glean are anagrams of each other. Design a function isAnagram that takes two strings and returns a boolean indicating whether or not the strings are anagrams of each other.

   a) [Strategy] Write an outline of your approach.

   b) [Code] Write your Python code.

   c) [Testing] Provide four test cases, using specific values of input parameters, and for each test case state the expected output.

6. **(Sorting)** What is the big-O time complexity for insertion sort? Provide two sample data sets of eight (8) values each that illustrate insertion sort running in best and worst case times.

7. **(Searching)** What are the logic errors in the following binarySearch function?

   ```python
def binary_search(data, target, start, end):
    if start >= end:
        return -1

    mid_index = (start + end) // 2
    mid_value = data[mid_index]

    if target == mid_value:
        return mid_index
    elif target < mid_value:
        return binary_search(data, target, start, mid_index-1)
    else:
        return binary_search(data, target, start+1, end)
```
8. **(Greedy Algorithms, Lists, Files, Classes, Loops)** You are a student who wants to take as many courses as possible without taking any course that overlaps with another. Assume the courses are in a file that contains one course per line. The information for each course is the course name, the start time and the stop time (using 24-hour time).

There are many ways we could approach solving this problem in a greedy manner, but only one is always optimal (chooses the most number of non-overlapping courses):

a.) **Earliest Start Time** - Consider courses in ascending order based on start time. Take the first course and add it to your schedule, then “cross out” any other course that conflicts with it. Repeat this until there are no courses left.

<table>
<thead>
<tr>
<th>Courses:</th>
<th>Result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 8 14</td>
<td>CS (8-14)</td>
</tr>
<tr>
<td>Calculus 9 11</td>
<td></td>
</tr>
<tr>
<td>Art 11 13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimal:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculus (9-11)</td>
</tr>
<tr>
<td>Art (11-13)</td>
</tr>
</tbody>
</table>

b.) **Shortest Interval** - Consider courses in ascending order based on length. Take the first course and add it to your schedule, then “cross out” any other course that conflicts with it. Repeat this until there are no courses left.

<table>
<thead>
<tr>
<th>Courses:</th>
<th>Result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art 11 14</td>
<td>Calculus (10-12)</td>
</tr>
<tr>
<td>CS 8 11</td>
<td></td>
</tr>
<tr>
<td>Calculus 10 12</td>
<td></td>
</tr>
<tr>
<td>Art 11 13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimal:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS (8-11)</td>
</tr>
<tr>
<td>Art (11-14)</td>
</tr>
</tbody>
</table>

c.) **Earliest Finish Time** - Consider courses in ascending order based on finish time. Take the first course and add it to your schedule, then “cross out” any other course that conflicts with it. Repeat this until there are no courses left.

<table>
<thead>
<tr>
<th>Courses:</th>
<th>Result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculus 9 11</td>
<td>Calculus (9-11)</td>
</tr>
<tr>
<td>CS 8 14</td>
<td>Art (11-13)</td>
</tr>
<tr>
<td>Art 11 13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimal:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculus (9-11)</td>
</tr>
<tr>
<td>Art (11-13)</td>
</tr>
</tbody>
</table>

In summary, choosing courses based on *earliest finish time* is always optimal. **For this problem we guarantee the file will be ordered based on finish time.**

a.) Design a class, `Course`, that can be used to represent a course.

b.) Write a function, `mkCourse`, that can build a course object.

c.) Write a function, `readCourses`, which takes the course file as an argument and returns a list of `Course` objects.

d.) Write a function, `greedySchedule`, which take a list of `Course` objects and returns the schedule as a list of `Course` objects using the optimal greedy algorithm.
9. **(Lists)** A *permutation* of a string $s$ is a string $s'$ such that $s'$ is a rearrangement of the letters of $s$. Note that the trivial rearrangement that leaves $s$ alone counts as a permutation. Given this notion, we can talk about the collection of all distinct permutations of a string $s$. For example, for '123' we have '123', '132', '213', '231', '312', '321'.

In part c), you will write a function to compute the list of all permutations of a string. But first in parts a) and b), you will write helper functions.

a) Write the function `insertEverywhere`, that takes a character $c$ and a string $s$, and returns a list of strings with $c$ inserted into $s$ in every possible position. For example, `insertEverywhere('a', '123')` should return ['a123', '1a23', '12a3', '123a'].

b) Write the function `insertEverywhereAll`, that takes a character $c$ and a list of strings $L$, and returns a list of strings that is constructed by concatenating calls to `insertEverywhere` on each of the elements of $L$. For example, `insertEverywhereAll('a', ['12', '34'])` should return ['a12', '1a2', '12a', 'a34', '3a4', '34a'].

Using `insertEverywhereAll`, write the function `perm` that takes a string $s$ and returns the list of all distinct permutations of $s$.

10. **(Trees)** Show the binary search tree that results from inserting these elements in the following order to an initially empty tree:
   6, 8, 4, 5, 2, 1, 3, 7, 9, 10.

11. **(Linked Lists)** This problem deals with implementations of linked lists.

   a) Write a function called `append(circList, value)` that inserts a new value at the end of a circularly linked list whose first node is referenced by the parameter `circList`. A circularly linked list has its last node linked back to the first node of the list. The `Node` class has two slots, `data` and `next`.

   b) What is the time complexity of `append` in terms of the list's length, $N$?

   c) Identify three tests for `append` in terms of the size of the list (before the append operation). The value appended is unimportant since it does not affect the behavior of the algorithm.

   d) Provide code for a function called `maxNode(circList)` that returns the largest value in a circularly linked list whose first node is referenced by `circList`.

   e) What is the time complexity of the `maxNode` operation in terms of list length, $N$?
f) Suggest three tests for maxNode in terms of the values of the list.

12. **(Trees)** Recall the definition of an expression tree:

* A *number tree*, which has one part: a number.

* A *sum tree*, which has the following parts.
  - a first sub-expression, which is an expression tree, and
  - a second sub-expression, which is an expression tree.

* A *product tree*, which has the following parts.
  - a first sub-expression, which is an expression tree, and
  - a second sub-expression, which is an expression tree.

Write the function `expToPostfix` that takes one argument, an expression tree, and returns a string that represents the expression in postfix. Note that in postfix notation, the addition or multiplication sign is written after the numbers. For example, `expToPostfix(mkSum(mkNum(2), mkProd(mkNum(3), mkNum(4))))` should return `'2 3 4 * +'`. 