For questions 1-10, choose the letter that gives the output of the code fragment. (Four points each.)

**Question 1**

```
initial = 0

for i in range(7, 10):
    initial -= i
    print(initial, end = ' ')
```

a. 0 0 0  
b. 0 -1 -3  
c. 7 15 24  
d. -7 -15 -24  
e. none of the above

**Question 2**

```
refrain = 'HALLELUJAH'
countPattern = ''
for letter in refrain:
    count = refrain.count(letter)
    if str(count) in countPattern:
        countPattern += letter
    else:
        countPattern += str(count)
print(countPattern)
```

a. TypeError: operator += not defined for type str  
b. 2233131122  
c. 2A3L1LUJAH  
d. HALLELUJAH  
e. none of the above

**Question 3**

```
cypher = 'Kansas is going bye-bye'
print(cypher[-7:] + cypher[6:7] + cypher[:6])
```

a. IndexError: string index out of range  
b. SyntaxError: invalid syntax  
c. bye-byesKansas  
d. bye-byeKansas  
e. none of the above
Question 4
oldMacDonald = 'eieio'
equipment = 'radio receiver'
count = 0
for i in range(len(equipment)-1):
    substring = equipment[i:i+2]
    if substring in oldMacDonald:
        count += 1
print(count)
a. 0
b. 2
c. 3
d. 6
e. none of the above

Question 5
def asymmetry(s):
symmetric = ''
    for i in range(len(s)):
        if s[i] == s[-i-1]:
            symmetric += s[i]
        else:
            return s[i:-i]
    return symmetric
t = 'abcdba'
print(asymmetry(t))
a. abcdba
b. cdabba
c. abba
d. cd
e. none of the above

Question 6
lst = [{'a':0,'z':25}, [2.7, 3.14], ['e', ['pi']]]
print(lst[2][1])
a. 2.7
b. 'pi'
c. ['pi']
d. IndexError: list index out of range
e. none of the above
Question 7

def longWords(s, num):
    longList = []
    sList = s.lower().split()
    for word in sList:
        if len(word) >= num:
            longList.append(word)
    return longList

t = 'The ideas of the ruling class are the ruling ideas'
print(longWords(t, 5))

a. ['ideas', 'ruling', 'class']
b. ['ideas', 'ruling', 'class', 'ruling', 'ideas']
c. ['ruling']
d. ['ruling', 'ruling']
e. none of the above

Question 8

d = {}
s = 'banana'
for letter in s:
    if letter not in d:
        d[letter] = 'odd'
    elif d[letter] == 'even':
        d[letter] = 'odd'
    else:
        d[letter] = 'even'
print(d)

a. {}
b. {'n', 'b', 'a'}
c. ['n': 'even', 'b': 'odd', 'a': 'odd']
d. {'n': 'even', 'b': 'odd', 'a': 'odd'}
e. none of the above
Question 9
outF = open('digits.txt', 'w')
for i in range(10):
    outF.write(str(i))
outF.close()
inF = open('digits.txt', 'r')
print(inF.read())
inF.close()

a. 0123456789
b. 12345678910
c. NameError: name 'digits' is already defined
d. NameError: name 'digits' is not defined
e. none of the above

Question 10
print(yesNo(3, 5, '=='))
def yesNo(a, b, comparison):
    if comparison == "==" and a == b:
        return 'Yes'
    elif comparison == '<' and a < b:
        return 'Yes'
    elif comparison == '>' and a > b:
        return 'Yes'
    else:
        return 'No'

a. Yes
b. No
c. yesNo
d. <
e. none of the above
**Question 11A (14 points)**

Write a function named `outposts()` that uses turtle graphics to draw a circle (an outpost) at each of the four corners of a square. Do not draw the sides of the square. (Hint: there is a turtle method `circle` that draws a circle of a specified radius.) You should not make any assumption about the initial state of the turtle (up/down, position or orientation), and there is no requirement concerning the state of the turtle on return.

The function `outposts()` takes three parameters:

i. a turtle, `t`, that is used to draw the outposts
ii. an int, `radius`, that is the radius of each outpost
iii. an int, `side`, that is the distance between successive outposts.

**Question 11B (6 points)**

Write code that calls `outposts()` so that `outposts()` draws circles with a line width of 5 and a radius of 10, with successive outposts separated by a distance of 50. (Hint: you must create the turtle that will be used before calling `outposts()`.)

The following would be correct graphical output:
**Question 12 (20 points)**
The following lower case letters are written partly below the baseline: g, j, p, q, y. We say that these letters have a descender.

Write a function named hasDescender() that returns a list of those words in a string in which at least one letter has a descender. A word should appear in the return list at most once, no matter how many letters in it have descenders and no matter how many times the word occurs in the input string. You may assume that the input string consists of only lower case letters and spaces – no punctuation or upper case letters. The order of words in the returned list does not matter.

**Input:** a string s that consists of words, separated by spaces  
**Return:** a list of words in s that contain at least one descender

For example, the following would be correct output:

```python
>>> will = 'suspicion always haunts the guilty mind'
>>> print(hasDescender(will))
['suspicion', 'always', 'guilty']
```

**Question 13 (20 points)**
Write a function named wordLengths() that computes the frequency with which words of different lengths occur in an input file and then returns a dictionary that contains this information.

The function wordLengths() takes one parameter: a string that is the name of an input file that exists before wordLengths() is called.

You may assume that the input file is in the current working directory. Also assume that the input file contains only lower case letters and endlines (no capital letters or punctuation).

Each occurrence of a word in the input file should be counted. Duplicate words should be counted twice, triplicates three times, and so on.

For example, suppose this is the content of the file santa.txt:

```
merry christmas to all  
and to all a good night
```

Then the function call

wordLengths('santa.txt')

should return a dictionary with these key/value pairs

```
{1: 1, 2: 2, 3: 3, 4: 1, 5: 2, 9: 1}
```