Question 1. (4 points) Consider the following Python code.

```python
for i in range(n):
    j = 1
    while j < n:
        print(i, j)
        j = j + 2
```

What is the big-oh notation $O(\ )$ for this code segment in terms of $n$?

Question 2. (4 points) Consider the following Python code.

```python
i = 1
while i < n:
    for j in range(n):
        print(j)
    for k in range(n):
        print(k)
    i = i * 2
```

What is the big-oh notation $O(\ )$ for this code segment in terms of $n$?

Question 3. (4 points) Consider the following Python code.

```python
def main(n):
    for i in range(n):
        doSomething(n)
        doMore(n)

def doSomething(n):
    for k in range($2^n$):
        print(k)

def doMore(n):
    for k in range(n):
        print(k)

main(n)
```

What is the big-oh notation $O(\ )$ for this code segment in terms of $n$?

Question 4. (8 points) Suppose a $O(\ n^4\ )$ algorithm takes 1 second when $n = 100$. How long would you expect the algorithm to run when $n = 1,000$?

Question 5. (5 points) In lab 2 (and on the Python Summary) the AdvancedDie class inherited from the Die class. How does inheritance aid a programmer in writing code?
Question 6. A **priority queue** has the same operations as a regular queue, except the items are NOT returned in the FIFO (first-in, first-out) order. Instead, each item has a priority that determines the order they are removed. One possible implementation of a priority queue would be to use a built-in Python list to store the items such that

- items in the Python list are **unordered** by their priorities,
- lowest number indicates the highest priority (i.e., dequeuing from the below priority queue would return 5)

\[
\begin{array}{cccc}
\text{PriorityQueue Object} & \text{Python List Object} \\
\_items: & \rightarrow \\
0 & 8 \\
1 & 5 \\
2 & 15 \\
3 & 10 \\
\end{array}
\]

a) (5 points) Complete the big-oh \( O(\_\_\_\_) \), for each PriorityQueue operation, assuming the above implementation. Let \( n \) be the number of items in the PriorityQueue.

<table>
<thead>
<tr>
<th>operation</th>
<th>big-oh ( O(____) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>enqueue(item)</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>dequeue</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>_str_</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>size</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

b) (15 points) Complete the method for the **dequeue** operation including the precondition check.

```python
class PriorityQueue(object):
    def __init__(self):
        self._items = []

    def dequeue(self):
        """Removes and returns the highest priority (lowest value) item in the PriorityQueue
        Precondition: the PriorityQueue is not empty.
        Postcondition: the highest priority (lowest value) item in the PriorityQueue is removed and returned"""
```

c) (5 points) Suggest an alternate PriorityQueue implementation to speed up some of its operations.
Question 7. Consider the binary heap approach to implement a priority queue. A Python list is used to store a complete binary tree (a full tree with any additional leaves as far left as possible) with the items being arranged by heap-order property, i.e., each node is ≤ either of its children. An example of a min heap “viewed” as a complete binary tree would be:

```
  4
 /   \
[2]  8
 / \
[4] 75
|   |
[8] 300
|   |
[9] 30
|   |
[10] 81
|   |
|   |
[12] 91
```

Python List actually used to store heap items

```
not used   4   8   13   75   30   20   50   300   81   57   91   25
```

a) (3 points) For the above heap, the list indexes are indicated in [ ]'s. For a node at index \( i \), what is the index of:
- its left child if it exists:
- its right child if it exists:
- its parent if it exists:

b) (7 points) What would the above heap look like after inserting 12 and then 2 (show the changes on above tree)

c) (3 points) What is the big-oh notation for inserting a new item in the heap?

Now consider the delMin operation that removes and returns the minimum item.

```
  4
 /   \
[2]  8
 / \
[4] 75
|   |
[8] 300
|   |
[9] 30
|   |
[10] 81
|   |
|   |
[12] 91
```

Python List actually used to store heap items

```
not used   4   8   13   75   30   20   50   300   81   57   91   25
```

d) (2 point) What item would delMin remove and return from the above heap?

e) (7 points) What would the above heap look like after delMin? (show the changes on above tree)

f) (3 points) Why does a delMin operation typically take longer than an insert operation?
Question 8. The textbook’s **Ordered list** ADT uses a singly-linked list implementation. I added the `_size` and `_tail` attributes:

OrderedList Object

- `_head`
- `_size`: 4
- `_tail`

a) (15 points) The `index(item)` method returns the position of the *item* in the list (e.g., ‘m’ is at position 2). Recall that the textbook’s implementation, assumes the *item* is in the list!!! Thus, the precondition is that *item* is in the list. Complete the `index(item)` method code including the precondition check.

```python
class OrderedList(object):
    def __init__(self):
        self._head = None
        self._size = 0
        self._tail = None

    def index(self, item):
        # Precondition: item is in the list
```

b) (10 points) Assuming the ordered list ADT described above does not allows duplicate items. Complete the big-oh `O()` for each operation. Let n be the number of items in the list.

<table>
<thead>
<tr>
<th>add(item)</th>
<th>pop()</th>
<th>length()</th>
<th>remove(item)</th>
<th>index(item)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adds the item into the list</td>
<td>removes and returns tail item</td>
<td>returns number of items in the list</td>
<td>removes the item from the list</td>
<td>returns the position of item in the list</td>
</tr>
</tbody>
</table>