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

O(n^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

2n \log_2 n \Rightarrow O(n \log_2 n)
```

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) - 2^n
        doMore(n) - 2^n

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

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

O(n 2^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 $n = 1,000$?

\[ T(n) = C n^4 \]

\[ T(1000) = C 1000^4 = C 10^{12} \]

\[ C = \frac{1}{100^4} = \frac{1}{10^8} \text{ sec} \]

\[ C = \frac{1}{10^{12}} \text{ sec} \]

\[ = 10^4 \text{ sec} \]

\[ = 10,000 \text{ sec} \]

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?

The subclass inherits working/correct code which it does not need to duplicate.
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., dequeueing from the below priority queue would return 5)

<table>
<thead>
<tr>
<th>PriorityQueue Object</th>
<th>Python List Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>items:</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>8 5 15 10</td>
</tr>
</tbody>
</table>

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>is_empty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>enqueue(item)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>str</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>size</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

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

c) (5 points) Suggest an alternate PriorityQueue implementation to speed up some of its operations.

Use a binary heap
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 \( \leq \) either of its children. An example of a min heap "viewed" as a complete binary tree would be:

![Binary Heap Diagram]

Python List actually used to store heap items

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>13</td>
<td>75</td>
<td>30</td>
<td>20</td>
<td>50</td>
<td>300</td>
<td>81</td>
<td>57</td>
<td>91</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

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: \( i \times 2 \)
- its right child if it exists: \( i \times 2 + 1 \)
- its parent if it exists: \( \lfloor i / 2 \rfloor \)

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? \( O(\log n) \)

Now consider the \texttt{delMin} operation that removes and returns the minimum item.

![Binary Heap Diagram after delMin]

Python List actually used to store heap items

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>13</td>
<td>75</td>
<td>30</td>
<td>20</td>
<td>50</td>
<td>300</td>
<td>81</td>
<td>57</td>
<td>91</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

d) (2 point) What item would \texttt{delMin} remove and return from the above heap? 4

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

f) (3 points) Why does a \texttt{delMin} operation typically take longer than an \texttt{insert} operation?

1. Grabbing last item in list to put at root is typically large and needs to percolate down further than inserted item goes up.
2. To move down one level requires 2 comparisons vs. 1 compare to move up
Question 8. The textbook’s **Ordered list** ADT uses a singly-linked list implementation. I added the `size` and `tail` attributes:

```
OrderedList Object

<table>
<thead>
<tr>
<th>_head</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'d'</td>
<td></td>
</tr>
<tr>
<td>_size</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>_tail</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 1 2 3

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):
        position = 0
        current = self._head
        while True:
            if current is None or current.getData() > item:
                raise ValueError("item not in list has not index")
            elif current.getData() == item:
                return position
            else:
                current = current.getNext()
                position += 1
```

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>Operation</th>
<th>( O(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(item)</code></td>
<td></td>
</tr>
<tr>
<td><code>pop()</code></td>
<td></td>
</tr>
<tr>
<td><code>length()</code></td>
<td></td>
</tr>
<tr>
<td><code>remove(item)</code></td>
<td></td>
</tr>
<tr>
<td><code>index(item)</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>( O(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(item)</code></td>
<td></td>
</tr>
<tr>
<td><code>pop()</code></td>
<td></td>
</tr>
<tr>
<td><code>length()</code></td>
<td></td>
</tr>
<tr>
<td><code>remove(item)</code></td>
<td></td>
</tr>
<tr>
<td><code>index(item)</code></td>
<td></td>
</tr>
</tbody>
</table>