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

```python
for i in range(n):
    for j in range(n*n):
        print (i, j)
```

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

$\Omega(n^2)$

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

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

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

$O(n \log n)$

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(n):
        print(k)

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

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

$O(n^2)$

Question 4. (8 points) Suppose a $O(n^3)$ algorithm takes 10 seconds when $n = 1,000$. How long would you expect the algorithm to run when $n = 10,000$?

$O(n^3) \Rightarrow T(n) = c \cdot n^3$

$T(1000) = c \cdot 1000^3 = 10^8$ sec

$c = \frac{10 \text{ sec}}{1000^3} = \frac{10}{10^9} \text{ sec} = 10^{-8} \text{ sec}$

$T(10,000) = c \cdot 10,000^3 = 10^{12}$ sec

$= 10^{-8} \cdot 10^{12} = 10^4$ sec

$= 1,000$ sec

Question 5. (5 points) Why should any method/function having a "precondition" raise an exception if the precondition is violated?

If the precondition is violated, then the method/function will not work correctly. Better to discover the error as soon as possible rather than later when tracking down the error is more difficult.
Question 6. Consider the following FIFO (First-In-First-Out) Queue implementation utilizing a Python list:
Recall that a queue is a linear data structure that allows adding new items at the rear and removing items from the front. One possible implementation of a queue would be to use a built-in Python list to store the items such that
- the **rear** item is always stored at index 0,
- the **front** item is always at index len(self._items)-1 or -1

![Diagram of Queue Object and Python List Object]

a) (8 points) Complete the big-oh $O()$, for each Queue operation, assuming the above implementation. Let $n$ be the number of items in the Queue.

<table>
<thead>
<tr>
<th></th>
<th>isEmpty</th>
<th>enqueue</th>
<th>dequeue</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

b) (10 points) Complete the method for the dequeue operation including the precondition check.

```python
def dequeue(self):
    """Removes and returns the front item of the queue
    Precondition:  the queue is not empty.
    Postcondition: front item is removed from the queue and returned"
    return self._items.pop(0)
```

c) (7 points) Suggest an alternate Queue implementation to speed up some of its operations.

Linked list implementation would have all operations $O(1)$ (or "circular")
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:

```
  9
 / \
[2] [12]
 / \  /  \
 / \ / \ / \ \
 / \  / \ / \  / \
300 125 [10] 117 50
```

Python List actually used to store heap items

```
  0  1  2  3  4  5  6  7  8  9  10
-------
  9  15 10 114 20 30 50 300 125 117
```

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: $i \div 2$

b) (7 points) What would the above heap look like after inserting 12 and then 25 (show the changes on above tree)

c) (3 points) What is the big-oh notation for inserting a new item in the heap? $O(\log n)$ since heaps

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

```
  9
 / \  \\  
[2] 15
 / \  /  \
 / \ / \ / \ \
```

Python List actually used to store heap items

```
  0  1  2  3  4  5  6  7  8  9  10
-------
  9  15 10 114 20 30 50 300 125 117
```

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

9

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

f) (3 points) What is the big-oh notation for delMin?

$O(\log n)$
Question 8. The textbook’s unordered list ADT uses a singly-linked list implementation. I added the _size and _tail attributes:

UnorderedList Object

0          1          2          3
_data  _next  _data  _next  _data  _next

_a_  _b_  _c_  _d_

4

_a_  _b_  _c_  _d_

1) (15 points) The `insert(position, item)` method adds the item to the list at the specified position. Unlike the textbook’s implementation, ASSUME that the list may contain duplicate items!!! The precondition is that `position` is a nonnegative integer. If `position` is 0, then add it to the head of the list. If `position` is _size or bigger_, then add it to the tail of the list. Complete the `insert(position, item)` method code including the precondition check.

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

    def insert(self, position, item):
        if position < 0:
            raise (IndexError, "position must be nonnegative")
        temp = Node(item)
        if position == 0:
            temp.setNext(self._head)
            self._head = temp
        elif position == self._size:
            self._tail = temp
        elif position > self._size:
            if self._size == 0:
                self._head = temp
            else:
                self._tail.setNext(temp)
        else:
            if self._size == 0:
                self._head = temp
            else:
                self._head = temp
```

b) (10 points) Assuming the unordered list ADT described above that 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>
<th>$O(n)$</th>
<th>$O(1)$</th>
<th>$O(1)$</th>
<th>$O(1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(position, item)</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pop()</code> removes and returns tail item</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>length()</code> returns number of items in the list</td>
<td></td>
<td></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>append(item)</code> adds item to the tail of list</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add(item)</code> adds item to the head of list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

`needs to reset _tail pointer.`