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

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

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*n*n)
def doSomething(n):
    for k in range(n):
        print(k)
def doMore(n):
    for j in range(n):
        print(j)
main(n)
```

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

Question 4. (8 points) Suppose a $O(\ n^5 \ )$ algorithm takes 10 second when $n = 100$. How long would the algorithm run when $n = 1,000$?

Question 5. (10 points) For a built-in Python list, explain each of the following average big-oh notations:

a) Why does `myList.insert(0, "item")` have an average big-oh of $O(n)$, where $n$ is the # of list items?

b) Why does `myList.append("item")` have an average big-oh of $O(1)$, where $n$ is the # of list items?
Question 6. A FIFO queue allows adding a new item at the rear using an enqueue operation, and removing an item from the front using a dequeue operation. One possible implementation of a queue would be to use a built-in Python list to store the queue items such that

- the front item is always stored at index 0,
- the rear item is always at index len(self._items) -1 or -1

![Queue Object](image1)

**Python List Object**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>'b'</td>
<td>'c'</td>
<td>'d'</td>
<td></td>
</tr>
</tbody>
</table>

**front**

**rear**

a) (6 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>Operation</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td></td>
</tr>
<tr>
<td>enqueue(item)</td>
<td></td>
</tr>
<tr>
<td>dequeue</td>
<td></td>
</tr>
<tr>
<td>peek (without removing it)</td>
<td></td>
</tr>
<tr>
<td><strong>str</strong></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
</tr>
</tbody>
</table>

b) (9 points) Complete the method for the dequeue operation, **including the precondition check to raise an exception if it is violated**.

```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"
```

c) (5 points) An alternate Queue implementation would swap the location of the front and rear items as in:

![Queue Object](image2)

**Python List Object**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>'c'</td>
<td>'b'</td>
<td>'a'</td>
<td></td>
</tr>
</tbody>
</table>

**front**

**rear**

Why is this alternate implementation probably not very helpful with respect to the Queue’s performance?
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:

```
Python List actually used to store heap items

not used   9   23   15   25   34   40   30   120   44   42
```

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 5 and then 3 (show the changes on above tree)

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

```
c) (2 point) What item would \texttt{delMin} remove and return from the above heap?
d) (7 points) What would the above heap look like after a \texttt{delMin} operation? (show the changes on above tree)
e) (6 points) Performing 20,000 \texttt{inserts} into an initially empty binary heap takes 0.23 seconds. Now, if we perform 20,000 \texttt{delMin} operations, it takes 0.39 seconds. Explain why these 20,000 \texttt{delMin} operations take more time than the 20,000 \texttt{insert} operations?
```
Question 8. The Node class can be used to dynamically create storage for each new item added to a Queue using a singly-linked implementation as in:

![LinkedQueue Object Diagram]

Node Objects

a) (6 points) Determine the big-oh, $O()$, for each LinkedQueue operation assuming the above singly-linked implementation. Let $n$ be the number of items in the queue.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Size Complexity</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>$O(n)$</td>
</tr>
<tr>
<td>peek</td>
<td>$O(n)$, without removing it</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

b) (14 points) Complete the enqueue method for the above LinkedQueue implementation.

```python
class LinkedQueue(object):
    """ Singly-linked list based Queue implementation."""
    def __init__(self):
        self._size = 0
        self._rear = None
        self._front = None

    def enqueue(self, newItem):
        """ Adds the newItem to the rear of the queue. 
        Precondition: none """
        # Implementation
```

c) (5 points) Suggest an improvement to the above implementation to speed up some of the queue operations enough to change their big-oh notation? Justify your answer.