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-o notation $O()$ for this code segment in terms of $n$?

$O(n^3 \log_2 n)$

$O(n^4) + 2$

$O(n \log n) + 2$

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-o notation $O()$ for this code segment in terms of $n$?

$O(n^2)$

$O(n \log n)^+ 2$

$O(n^2 \log n^+ 2)$

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

```python
def main(n):
    for i in range(n):
        doSomething(n)
        doMore(n^2 + n^3)

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

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

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

$O(n^4)$

$O(n^4)^+ 2$

$O(n^4)^+ 2$

Question 4. (8 points) Suppose a $O(n^3)$ algorithm takes 10 seconds when $n = 100$. How long would the algorithm run when $n = 1000$?

\[
T(n) = C \cdot n^5 \quad T(100) = C \cdot 100^5 = 10 \text{ sec} \\
\begin{align*}
C &= \frac{10 \text{ sec}}{100^5} = \frac{10 \text{ sec}}{10^{10}} \\
C &= \frac{1 \text{ sec}}{10^9}
\end{align*}
\]

\[
T(1000) = C \cdot 1000^5 = \frac{1 \text{ sec}}{10^9} \cdot 10^{15} = 10^6 \text{ sec}
\]

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

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

To insert an item at index 0, all $n$ items in the list must be shifted right to create a hole for the new item.

b) Why does `myList.append("item")` have an average big-oh of $O(1)$, where $n$ is the # of list items?

The Python list is assumed to have extra/unused space on its right end, so the new item can use the next free spot.
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

![Diagram of Queue Object](image1)

![Diagram of Python List Object](image2)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time 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 (returns front item without removing it)</td>
<td>O(1)</td>
</tr>
<tr>
<td><em>str</em></td>
<td>O(n)</td>
</tr>
<tr>
<td>size</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

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>Time 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 (returns front item without removing it)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><em>str</em></td>
<td>$O(n)$</td>
</tr>
<tr>
<td>size</td>
<td>$O(1)$</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""
    if self.isEmpty():
        raise ValueError("Cannot dequeue from empty queue.")
    return self._items.pop(0)
```

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

![Diagram of Queue Object](image3)

![Diagram of Python List Object](image4)

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

"Now enqueue becomes $O(n)$ and dequeue becomes $O(1)$"
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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>23</td>
<td>15</td>
<td>25</td>
<td>34</td>
<td>40</td>
<td>30</td>
<td>120</td>
<td>44</td>
<td>42</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: \( 2 \times i + 1 \)
- its right child if it exists: \( 2 \times i + 1 \)
- its parent if it exists: \( \lfloor i / 2 \rfloor \)

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 delMin operation that removes and returns the minimum item.

![Diagram](image)

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

2d) (7 points) What would the above heap look like after a delMin operation? (show the changes on above tree)

d) (6 points) Performing 20,000 inserts into an initially empty binary heap takes 0.23 seconds. Now, if we perform 20,000 delMin operations, it takes 0.39 seconds. Explain why these 20,000 delMin operations take more time than the 20,000 insert operations? On an insert the new item is repeatedly compared with its parent until it finds the right spot. On a delMin the last item is moved to the root and percolates down by comparing its two children to find the min. Plus, item moved in delMin is large and will likely percolate down more levels than a new item percolates up.
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

\[ \text{size} \rightarrow \text{front} \rightarrow \text{data} \rightarrow \text{next} \rightarrow \text{data} \rightarrow \text{next} \rightarrow \text{data} \rightarrow \text{next} \rightarrow \text{data} \rightarrow \text{next} \rightarrow \text{size} \]

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>isEmpty</th>
<th>enqueue(item)</th>
<th>dequeue</th>
<th>peek - returns front item without removing it</th>
<th><strong>str</strong></th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(1) )</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 """
        temp = Node(newItem)
        temp.setNext(self.rear)
        self.rear = temp
        self.size += 1.
        if self.size == 1:
            self.front = temp
```

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.

Two ways to get dequeue \( O(1) \):

1) (best way) -- have front point to first Node in the linked list e.g., \( \text{front} \rightarrow \ldots \rightarrow \ldots \rightarrow \text{front} \)

2) Use doubly-linked list of Node 2-way nodes.