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

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

O \left( n^2 \log n \right)
```

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

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

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

O \left( n^2 \right)
```

What is the big-oh notation $O(\cdot)$ 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):
        for j in range(n):
            print(j)

def doSomething(n):
    doMore(n)

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

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

main(n)
```

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

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

$$T(n) = cn^4$$

$T(100) = c \cdot 100^4 = 10 \text{ sec}$

Therefore, $c = \frac{10 \text{ sec}}{100^4}$.

$T(1000) = c \cdot 1000^4 = 10^5 \text{ sec}$

Question 5. (10 points) Why should you design a program instead of “jumping in” and start by writing code?

By designing the program first, you are able to avoid mistakes and reworking of code, so overall time is saved.

Mark F.
Question 6. Consider the following Stack implementation utilizing a Python list:

```
"Abstract" Stack

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bottom</td>
</tr>
</tbody>
</table>

Stack Object

items: [ ]

Python list Object

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>c</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>top</td>
<td>bottom</td>
<td></td>
</tr>
</tbody>
</table>
```

a) (6 points) Complete the big-oh notation for the Stack methods assuming the above implementation: ("n" is the # items)

<table>
<thead>
<tr>
<th>Method</th>
<th>Big-oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(item)</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>pop()</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>peek()</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>size()</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>( O(1) )</td>
</tr>
<tr>
<td><strong>init</strong></td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

b) (9 points) Complete the code for the pop method including the precondition check.

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

def pop(self):
    """Removes and returns the top item of the stack
    Precondition: the stack is not empty.
    Postcondition: the top item is removed from the stack and returned"""

    if len(self._items) == 0:
        raise ValueError("Cannot pop empty stack!")

    return self._items.pop(0)
```

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

Flip the order of the stack so the top is at the largest index, so all operations (except __str__) are \( O(1) \).
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 $\leq$ 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>7</td>
<td>12</td>
<td>31</td>
<td>30</td>
<td>75</td>
<td>40</td>
<td>100</td>
<td>100</td>
<td>94</td>
<td>88</td>
<td>91</td>
<td>45</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: $2i + 1$
- its right child if it exists: $2i + 2$
- its parent if it exists: $\lfloor i / 2 \rfloor$

b) (7 points) What would the above heap look like after inserting 18 and then 9 (show the changes on above tree)

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

![After DelMin Operation 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>7</td>
<td>12</td>
<td>31</td>
<td>30</td>
<td>75</td>
<td>40</td>
<td>100</td>
<td>100</td>
<td>94</td>
<td>88</td>
<td>91</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

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

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

e) (6 points) What is the big-oh notation for the delMin operation? (EXPUN YOUR ANSWER)

$O(\log n)$ Moving last item to index 0 takes $O(1)$, percolating down from index 1 cause the index to at least double. You can only double the index 1 $\log_2(n)$ times before it reaches $n$, so $\log_2 n$
Question 8. The Node class (in node.py) is used to dynamically create storage for a new item added to the stack. Consider the following LinkedQueue class using this Node class. Conceptually, a LinkedQueue object would look like:

"Abstract Queue"

```
| 'w' | 'x' | 'y' |
front   rear
```

LinkedQueue Object

```
_data: ['w', 'x', 'y']
_size: 3
_rear: None
```

Node Objects

```
<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>'w'</td>
<td></td>
</tr>
<tr>
<td>'x'</td>
<td></td>
</tr>
<tr>
<td>'y'</td>
<td></td>
</tr>
</tbody>
</table>
```

(a) (13 points) Complete the dequeue method including the precondition check.

```
class LinkedQueue(object):
    """ Linked-list based queue implementation."""
    
def __init__(self):
        self._front = None
        self._size = 0
        self._rear = None

def dequeue(self):
    """ Removes and returns the front item in the queue. 
    Precondition: the queue is not empty. """
    if self._size == 0:
        raise ValueError("Cannot dequeue")
    temp = self._front
    self._front = self._front._next
    self._size -= 1
    if self._size == 0:
        self._rear = None
    return temp._data
```

class Node:
    def __init__(self, initdata):
        self.data = initdata
        self.next = None

def getData(self):
    return self.data

def getNext(self):
    return self.next

def setData(self, newdata):
    self.data = newdata

```
b) (7 points) Assuming the queue ADT described above. Complete the big-oh O() for each queue operation. Let n be the number of items in the queue.

<table>
<thead>
<tr>
<th></th>
<th>enqueue(Item)</th>
<th>dequeue()</th>
<th>size()</th>
<th>str()</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>init</em></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

(5 points) Would using doubly-linked nodes (i.e., Node2Way) speed up some of queue operations? Justify your answer. No, slow down operations if "previous" links must be maintained.