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

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

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

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):
        doSomething(n)

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

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

main(n)
```

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

Question 4. (6 points) Suppose a $O(n^4)$ algorithm takes 10 second when $n = 1000$. How long would the algorithm run when $n = 10,000$?

Question 5. (8 points) Why should a method/function having a "precondition" raise an exception if the precondition is violated?
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 \( \text{len(self._items)} - 1 \) or -1

![Queue Object Diagram]

\[ \text{Queue Object} \quad \text{Python List Object} \]

\[ \_\text{items:} \quad \begin{array}{cccc} \text{'a'} & \text{'b'} & \text{'c'} & \text{'d'} \\ \text{front} & \text{rear} \end{array} \]

a) (6 points) Complete the expected 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>( 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><strong>str</strong></td>
<td>( O(n) )</td>
</tr>
<tr>
<td>size</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

b) (8 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) (8 points) Complete the method for the __str__ operation,

```python
def __str__(self):
    """Returns a string representation of items from front to rear. ""
    strResult = "(front) "
```
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:

```
      12
     /\  
    23 17
   /\ /\  /\  
  34 25 60 90
 /\ /\ /\ /\ /\ /\ /\  
120 44 28 31 96 84 98
```

Python List actually used to store heap items

```
not used 12 23 17 34 25 60 90 120 44 28 31 84 96 98
```

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 40 and then 20 (show the changes on above tree)

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

```
      12
     /\  
    23 17
   /\ /\  /\  
  34 25 60 90
 /\ /\ /\ /\ /\ /\ /\  
120 44 28 31 96 84 98
```

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

d) (7 points) What would the heap look like after deleteMin? (show the changes on tree in the middle of the page)

e) (6 points) Performing 20,000 inserts into an initially empty binary heap takes 0.23 seconds. Now, if we perform 20,000 deleteMin operations, it takes 0.39 seconds. Explain why 20,000 deleteMin operations take more time than the 20,000 insert operations?
Question 8. The `Node2Way` and `Node` classes can be used to dynamically create storage for each new item added to a Deque using a doubly-linked implementation as in:

DoublyLinkedDeque Object

```
    _front: previous data next previous data next previous data next previous data next
    _rear:  
    _size: 4
```

a) (6 points) Complete the big-oh expected $O(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)$, for each DoublyLinkedDeque operation, assuming the above implementation. Let $n$ be the number of items in the DoublyLinkedDeque.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addRear</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>removeRear</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>addFront</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>removeFront</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

b) (16 points) Complete the `addRear` method for the above DoublyLinkedDeque implementation.

```python
class DoublyLinkedDeque(object):
    """ Doubly-linked list based deque implementation."""
    def __init__(self):
        self._size = 0
        self._front = None
        self._rear = None
    def addRear(self, newItem):
        """ Adds the newItem to the rear of the Deque.
        Precondition: none """
        if self._size == 0:
            self._front = Node2Way(newItem)
            self._rear = self._front
        else:
            self._rear.next = Node2Way(newItem)
            self._rear = self._rear.next
        self._size += 1
class Node2Way(Node):
    def __init__(self, initdata):
        Node.__init__(self, initdata)
        self.previous = None

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
def setNext(self, newnext):
    self.next = newnext
```

c) (5 points) Why would using singly-linked nodes (i.e., only Node objects with data and next) to implement the Deque lead to poor performance (i.e., cause some Deque operations to have worse big-oh notations)? Justify your answer.