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

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

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

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

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

Question 4. (9 points) Suppose a $O(n^3)$ algorithm takes 10 seconds when $n = 1000$. How long would you expect the algorithm to run when $n = 10,000$?

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

$$T(1000) = c \cdot (1000)^3 = 10 \text{ sec}$$

$$c = \frac{10 \text{ sec}}{(1000)^3} = \frac{1 \text{ sec}}{10^8}$$

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

$$= \left(\frac{1 \text{ sec}}{10^8}\right) \cdot 10^{12}$$

$$= 10^4 \text{ sec}$$

$$= 10,000 \text{ sec}$$

Question 5. (9 points) Why should any method/function having a "precondition" raise an exception if the precondition is violated?

The method probably does not work correctly if the precondition is violated, so it is better to detect the error immediately. Thus, making it easier to find the error.
Question 6. A Deque (pronounced "Deck") is a linear data structure which behaves like a double-ended queue, i.e., it allows adding or removing items from either the front or the rear of the Deque. One possible implementation of a Deque would be to use a built-in Python list to store the Deque 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

![Deque Object](image)

![Python List Object](image)

<table>
<thead>
<tr>
<th>Method</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>isNone</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addRear</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeRear</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addFront</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>removeFront</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>size</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

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

```
6  isEmpty  addRear  removeRear  addFront  removeFront  size
    O(1)     O(1)     O(1)      O(n)      O(n)         O(1)
```

b) (9 points) Complete the method for the `removeRear` operation including the precondition check.

```python
def removeRear(self):
    
    """Removes and returns the rear item of the Deque
    Precondition: the Deque is not empty.
    Postcondition: Rear item is removed from the Deque and returned"
    
    if len(self._items) == 0:
        raise ValueError("cannot remove from an empty Deque")
    return self._items.pop()
```

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

```
We could use a doubly-linked list of nodes with
- front and rear pointers.
```
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:

```
          2
       /   \
      4     1
     /     / \
    1      3   6
   / \
 / \ / \   / \   /  \
75 [5] 30 26  50
 / \ / \   / \   / \
 / \ / \   / \   / \
300 [10] 5  30
```

Python List actually used to store heap items

```
0 1 2 3 4 5 6 7 8 9 10
  4 8 13 75 30 20 50 300 81 57
```

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: \( \lfloor i / 2 \rfloor \)

b) (7 points) What would the above heap look like after inserting 6 and then 2 (show the changes on above tree)

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

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

```
          13
       /   \
      50    [3]
     / \   /  \
   20  50
 / \ / \
[4] 30
 / \ / \
75 [5] 30
 / \ / \
81 [10] 57
 / \ / \
```

Python List actually used to store heap items

```
0 1 2 3 4 5 6 7 8 9 10 11 12
  4 8 13 75 30 20 50 300 81 57 91 85
```

d) (2 points) What item would \texttt{delMin} remove and return from the above heap? 4

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

f) (3 points) Why does a \texttt{delMin} operation typically take longer than an \texttt{insert} operation?

Insert only has to compare new item with parent while \texttt{delMin} compares twice per level (i.e., two children to find min. and then min. child with item perculiating down). Plus, inserted item generally twists depths of tree levels.
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 LinkedStack class using this Node class. Conceptually, a LinkedStack object would look like:

```
class LinkedStack(object):
    """ Link-based stack implementation. """
    def __init__(self):
        self._bottom = None
        self._size = 0
        self._top = None

    def __str__(self):
        """ Returns a string with items strung from bottom to top. Each item should be separated by a space. """
        resultStr = """"
        temp = self._bottom
        while temp != None:
            resultStr += str(temp.getData()) + " \n"
            temp = temp.getNext()
        return resultStr
```

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

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

a) (11 points) Complete the __str__ method above.

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>$O(\cdot)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(item)</td>
<td>$O(1)$</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><strong>str</strong>()</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

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

Reverse the "direction" of the next pointers:

```
<top>  <c>  <b>  <a>
```

```