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

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

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

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):
        doMore(n)
def doMore(n):
    k = n
    while k > 1:
        print(k)
        k = k // 2
main(n)
```

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

Question 4. (6 points) Suppose a $O(n^2)$ algorithm takes 1 second when $n = 100$. How long would the algorithm run when $n = 10,000$?

Question 5. (7 points) Why should medium/large size programs be written using function definitions instead of a single block of monolithic code written at the top-level (i.e., all statements written outside of any function)?
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`

```python
Deque Object
 ITEMS:
 front  'a' 'b' 'c' 'd'
 rear

Python List Object
 0  1  2  3
'a' 'b' 'c' 'd'
```

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average Big-oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addFront</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeFront</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><strong>str</strong></td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

b) (7 points) Complete the method for the `removeFront` operation, **including the precondition check to raise an exception if it is violated**.

```python
def removeFront(self):
    """Removes and returns the front item of the Deque
    Precondition: the Deque is not empty.
    Postcondition: Front item is removed and returned from the Deque"
    if not self.isEmpty():
        raise Exception("Deque is empty")
    result = self._items[0]
    self._items.pop(0)
    return result
```

c) (7 points) Complete the method for the `__str__` operation.

```python
def __str__(self):
    """Returns the string representation of the Deque.
    Precondition: none
    Postcondition: Returns a string representation of the Deque from the
    front item thru the rear item with a blank space between each item."
    resultStr = "(front) "
    return resultStr"""
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:

```
                      14
                     /   \
                    [2]  [3]
                   /   /   \
                 /   /     /   \
                [8] [9] [10] [11]
               /     /     /     \
              [110] 47  98  61
```

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 7 and then 24 (show the changes on above tree)

c) (6 points) Explain why the insert operation is $O(\log_2 n)$, where $n$ is the number of items in the heap.

Question 8. Now consider the heap’s delMin operation that removes and returns the minimum item.

```
                      14
                     /   \
                    [2]  [3]
                   /   /   \
                 /   /     /   \
                [8] [9] [10] [11]
               /     /     /     \
              [110] 47  98  61
```

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

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

c) (5 points) What would be the $O(\ )$ of a single delMin, where $n$ is the number of items in the heap? (no justification needed)
Question 9. The Node class can be used to dynamically create storage for each new item added to a Deque using a singly-linked implementation as in:

![Diagram of LinkedDeque Object and Node Objects]

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

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

b) (14 points) Complete the `addFront` method for the above LinkedDeque implementation.

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

def addFront(self, newItem):
    """ Adds the newItem to the front of the Deque.
    Precondition: none """
```

```
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
```

c) (5 points) Would using doubly-linked nodes (Node2Way with previous, data, and next) improve the above implementation (i.e., speed up some of the queue operations enough to change their big-oh notation)? Justify your answer.