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 $O()$ for this code segment in terms of $n$? $O(n^2)$

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 $O()$ for this code segment in terms of $n$? $O(n^2 \log_2 n)$

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

```python
def main(n):
    for i in range(n):
        doSomething(n) = $\log_2 n$

    def doSomething(n):
        for j in range(n):
            doMore(n) = $\log_2 n$

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

main(n)
```

What is the big-oh notation $O()$ for this code segment in terms of $n$? $O(n^2 \log_2 n)$

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

$T(n) = c \cdot n^2$

$T(10000) = c \cdot 10000^2 = c \cdot 10^8$

$T(100) = c \cdot 100^2 = 1 \text{ sec}$

$c = \frac{1 \text{ sec}}{100^2} = 10^{-4} \text{ sec}$

$\frac{10^8}{10^{-4}} = 10^4 \text{ sec}$

$= 10,000 \text{ sec}$

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)?

By splitting the program into functions that do
- a smaller subtask, we have several advantages:
  - function is smaller and easier to write, test, and debug
  - you might be able to reuse the function in a similar program
  - work can be easily divided if working in a team
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

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

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>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</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>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 self.isEmpty():
        raise Exception("Cannot removeFront from empty Deque")
    return self._items.pop(0)
```

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)"
    for item in self._items:
        resultStr += str(item) + " \\
    return resultStr + "(rear)"
```
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:

```
+---+---+---+
| 7 | 19 |
+---+---+---+
| 34| 52 | 26 |
+---+---+---+
| 47| 98 | 61 |
+---+---+---+
```

Python List actually used to store heap items

<table>
<thead>
<tr>
<th>10</th>
<th>21</th>
<th>19</th>
<th>34</th>
<th>52</th>
<th>26</th>
<th>110</th>
<th>47</th>
<th>98</th>
<th>61</th>
<th>46</th>
<th>111</th>
<th>33</th>
</tr>
</thead>
</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 7 and then 24 (show the changes on above tree)

c) (6 points) Explain why the insert operation is O(log₂n), where n is the number of items in the heap.

Inserted item initially at index n. Repeatedly compared with parent and swapped if necessary. Since the parent's index is always at \( \lfloor i/2 \rfloor \) if the child is at i, the O(log₂n), we can only cut n in half log₂n times before

Question 8. Now consider the heap's delMin operation that removes and returns the minimum item. Get to 1.

```
+---+---+---+
| 21| 34 |
+---+---+---+
| 52 |
+---+---+---+
```

a) (2 point) What item would delMin remove and return from the above heap? 14

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( ), where n is the number of items in the heap? O(log₂n)
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:

```
LinkedDeque Object
\[\text{temp}^{\text{\downarrow}}\]

\(\text{size}: 4\)
\(\text{front}: \text{None}\)
\(\text{rear}: \text{None}\)

\(\text{Node Objects}\)
\(\text{data}^{\text{\downarrow}}\)
\(\text{next}^{\text{\downarrow}}\)
\(\text{data}^{\text{\downarrow}}\)
\(\text{next}^{\text{\downarrow}}\)
\(\text{data}^{\text{\downarrow}}\)
\(\text{next}^{\text{\downarrow}}\)
\(\text{data}^{\text{\downarrow}}\)
\(\text{next}^{\text{\downarrow}}\)

\(\text{data}^{\text{\downarrow}}\)
\(\text{next}^{\text{\downarrow}}\)

\(\text{a}^{\text{\downarrow}}\)
\(\text{b}^{\text{\downarrow}}\)
\(\text{c}^{\text{\downarrow}}\)
\(\text{d}^{\text{\downarrow}}\)
```

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>(O(n))</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(n))</td>
</tr>
<tr>
<td>_str</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

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

```
class LinkedDeque(object):
    """ Slightly-linked list based deque implementation. """
    def __init__(self):
        self._size = 0
        self._front = None
        self._rear = None
def addFront(self, newItem):
    """ Adds the new item to the front of the Deque. 
Precondition: none """
    temp = Node(newItem)
    if self._size == 0:
        self._front = temp
        self._rear = temp
    else:
        temp.next = self._front
        self._front = temp
        self._size += 1
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) improvement the above implementation (i.e., speed up some of the queue operations enough to change their big-oh notation)? Justify your answer.

Yes, the removeRear would go to \(O(1)\) if doubly-linked nodes were used since the node before the removed rear node can be found in \(O(1)\) time with doubly-linked nodes.