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

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

O(n^2 log_2 n)
```

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)

O(n^2)
```

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) - \( O(n) \)

def doSomething(n):
    for k in range(n):
        doMore(n) - \( O(n) \)

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

main(n)
```

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

Question 4. (8 points) Suppose a \( O(n^3) \) algorithm takes 10 second when \( n = 100 \). How long would the algorithm run when \( n = 1,000 \)?

\[
T(n) = c \cdot n^3
\]

\[
T(100) = c \cdot 100^3 = 10 \text{ sec}
\]

\[
T(1000) = c \cdot 1000^3
\]

\[
c = \frac{10 \text{ sec}}{100^3} = 10^{-5} \text{ sec}
\]

\[
\frac{1000^3}{10^5} = \frac{10^9}{10^5} = 10^4 \text{ sec}
\]

\[10,000 \text{ sec}\]

Question 5. (10 points) Why should any method/function having a "precondition" raise an exception if the precondition is violated?

To inform the programmer that an error in using the function has occurred. It aids in debugging of the program since error detected immediately.
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 \(\text{len(self._items) - 1 or -1}\)

![Deque Object Diagram](image)

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

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

b) (9 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 from the Deque and returned""
    if len(self._items) == 0:
        raise ValueError("Cannot removeFront from empty Deque")
    return self._items.pop(0)
```

c) (5 points) An alternate Deque implementation would swap the location of the front and rear items as in:

![Alternate Deque Object Diagram](image)

Why is this alternate implementation probably not very helpful with respect to the Deque’s performance?

- addRear and removeRear become \(O(n)\)
- addFront and removeFront become \(O(1)\)

so it just swaps the \(O(n)\)'s around.
Spring 2016. Data Structures - Test 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:

```
      4
   [1]  
 [2]  
  23
[4]  [5]  
 34  25
[8]  [9]  [10]  [11]  [12]  
 120  44  28  31  26
```

Python List actually used to store heap items

```
0  1  2  3  4  5  6  7  8  9  10  11  12
[4]  23  15  34  25  20  30  120  44  28  31  26
```

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: \( 2 \times i \)
   - its right child if it exists: \( 2 \times i + 1 \)
   - 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)

c) (6 points) What is the big-oh notation for the insert operation? (EXPLAIN YOUR ANSWER)

\( O(\log_2 n) \) The inserted item starts at index \( n \), where \( n \) is the # of items in the heap. Its index in the worst case is halved \((\text{\textit{up}})\) until it reaches 1.

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

```
      4
   [1]  
 [2]  
 23
 34  25  18  30
[8]  [9]  [10]  [11]  [12]  
 120  44  28  31  26
```

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

e) (7 points) What would the above heap look like after \texttt{delMin}? (show the changes on above tree)
Question 8. The `Node2Way` class (which inherits the `node.py` class) can be used to dynamically create storage for each new item added to a Deque using a doubly-linked implementation as in:

DoublyLinkedList Deque Object

Node2Way Objects

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>( O() )</th>
</tr>
</thead>
<tbody>
<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>size</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>str</td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>

b) (14 points) Complete the `addRear` method.

```python
class DoublyLinkedListDeque(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 """
        temp = Node2Way(newItem)
        if self.size == 0:
            self.front = temp
            self.rear = temp
        else:
            temp.setPrevious(self.rear)
            self.rear.setNext(temp)
            self.rear = 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 setNext(self, newnext):
        self.next = newnext

class Node2Way(Node):
    def __init__(self, initdata):
        Node.__init__(self, initdata)
        self.previous = None

    def getPrevious(self):
        return self.previous

    def setPrevious(self, newprevious):
        self.previous = newprevious
```

c) (5 points) Would using singly-linked nodes (i.e., `Node` objects instead of `Node2Way`) slow down any of the Deque operations? Justify your answer.

Yes, the `removeRear` would become \( O(n) \) because we need to reset the self.rear pointer by starting at the first node and traversing down the chain of nodes.