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

```python
def main(n):
    for i in range(n*n*n*n):
        for j in range(n):
            for k in range(n):
                print(i, j, k)
```

What is the big-oh notation \( O() \) for this code segment in terms of \( n \)? \( O(n^6) \)

Question 2. (4 points) Consider the following Python code.

```python
k = 1
while k < n:
    print(k)
    k = k * 2
```

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

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

```python
def main(n):
    for i in range(n):
        doSomething(i)
        doMore(n)

def doSomething(n):
    for j in range(n):
        doMore(n)  # \( O(n^2) \)

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

What is the big-oh notation \( O() \) for this code segment in terms of \( n \)? \( O(n^4) \)

Question 4. (6 points) Suppose a \( O(n^3) \) algorithm takes 1 second when \( n = 100 \). How long would the algorithm run when \( n = 1,000 \)?

\[
T(n^3) = c \cdot n^3
T(1000) = c \cdot 1000^3 = 1 \text{ sec}
\]

\[
c = \frac{1 \text{ sec}}{100^3}
\]

Question 5. (7 points) Why should you design a program first instead of “jumping in” and start by writing code?

To avoid mistakes mainly. If you start by writing code, then you are more likely to go down the wrong path and need to rewrite code. Overall, starting by writing code will take more time.
Question 6. Consider the following Stack implementation utilizing a Python list: **Note: top is at index 0.**

<table>
<thead>
<tr>
<th>Abstract Stack</th>
<th>Stack Object</th>
<th>Python list Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) (6 points) Complete the big-oh notation for the Stack methods assuming the above implementation: ("n" is the # items)

<table>
<thead>
<tr>
<th>Method</th>
<th>Big-oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(newItem)</td>
<td>$O(n)$</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>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

b) (14 points) Complete the code for the `peek` and `push` methods, **including the precondition check to raise an exception if it is violated.**

```python
class Stack:
    def __init__(self):
        self._items = []

    def peek(self):
        """Returns the top item of the stack without removing it.
        Precondition: the stack is not empty.
        Postcondition: the top item is returned without removed it."""
        if len(self._items) == 0:
            raise Exception("cannot peek at top of empty stack")
        return self._items[0]

    def push(self, newItem):
        """Adds the newItem to the top of the stack.
        Precondition: none.
        Postcondition: the newItem has been added to the top of the stack."""
        self._items.insert(0, newItem)
```

c) (5 points) Suggest an improvement to the above Stack implementation to speed up some of its operations.

5. **Have the bottom at index 0, so push and pop can be $O(1)$** (by doubly-linked)
Question 8. The Node class can be used to dynamically create storage for each new item added to a queue using a singly-linked implementation as in:

LinkedQueue Object

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>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(1) )</td>
</tr>
</tbody>
</table>

b) (19 points) Complete the `enqueue` method for the above LinkedQueue implementation.

```python
class LinkedQueue(object):
    """Singly-linked list based queue implementation."""

    def __init__(self):
        self._front = None
        self._rear = None
        self._size = 0

    def enqueue(self, newItem):
        """Adds the newItem to the rear of the queue.
        Precondition: none """

        temp = Node(newItem)
        if self._size == 0:
            self._front = temp
            self._size += 1
        else:
            self._rear.setNext(temp)
            self._rear = temp
            self._size += 1
```

```python
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
```

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:

![Binary Heap Diagram]

Python List actually used to store heap items

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>13</td>
<td>14</td>
<td>[6]</td>
<td>9</td>
<td>30</td>
<td>90</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>25</td>
<td>19</td>
<td>34</td>
<td>25</td>
<td>30</td>
<td>90</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>91</td>
<td>46</td>
<td>46</td>
<td>88</td>
<td>91</td>
<td>46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>44</td>
<td>88</td>
<td>120</td>
<td>44</td>
<td>88</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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: \( 2i \)
- its right child if it exists: \( 2i + 1 \)
- its parent if it exists: \( \lfloor i/2 \rfloor \)

b) (7 points) What would the above heap look like after inserting 14 and then 18 (show the changes on above tree) Now consider the \texttt{delMin} operation that removes and returns the minimum item.

![Updated Heap Diagram after inserting 14 and 18]

2. c) (2 point) What item would \texttt{delMin} remove and return from the above heap? 10

d) (7 points) What would the heap look like after \texttt{delMin}?(show the changes on tree in the middle of the page)

e) (6 points) Explain why the \texttt{delMin} operation is \( O(\log_2 n) \), where \( n \) is the number of items in the heap.

The item moved to root at index 1 percolates down the heap by being swapped with a child at either index \( 2i \) or \( 2i+1 \). Since the index is more than doubling with each swap, then \( O(\log_2 n) \).