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

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

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

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

Answer: $O(n^2)$

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

```python
for i in range(n):
    j = n
    while j > 0:
        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 $n$?

Answer: $O(n^2 \log_2 n)$

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

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

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

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

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

Answer: $O(n^4)$

Question 4. (8 points) Suppose a $O(n^4)$ algorithm takes 10 second when $n = 1000$. How long would the algorithm run when $n = 10,000$?

\[
T(n) = c \cdot n^4
\]

\[
T(1000) = c \cdot 1000^4 = 10 \text{ sec}
\]

\[
c = \frac{10 \text{ sec}}{1000^4} = \frac{10}{10^{12}} = \frac{1}{10^{11}} \text{ sec}
\]

\[
T(10000) = c \cdot 10000^4 = 10 \text{ sec}
\]

\[
= \frac{1}{10^{11}} \cdot 10^{16} = 10^{5} \text{ sec}
\]

\[
= 100,000 \text{ sec}
\]

Question 5. (10 points) Why should you design a program instead of “jumping in” and start by writing code?

Answer: Without designing first you will be writing code that you’ll need to modify or throw away later as you work further.
Question 6. A FIFO (First-In-First-Out) queue allows adding a new item at the rear using an enqueue operation, and removing an item from the front using a dequeue operation. One possible implementation of a queue would be to use a built-in Python list to store the queue items such that

- the **rear** item is *always stored at index 0*,
- the **front** item is always at index len(self._items) - 1, or -1

![Queue Object](image)

**Python List Object**

<table>
<thead>
<tr>
<th>rear</th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>'c'</td>
</tr>
<tr>
<td>'b'</td>
<td>'a'</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>isEmpty</th>
<th>enqueue(item)</th>
<th>dequeue</th>
<th>peek - returns front item without removing it</th>
<th><strong>str</strong></th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

b) (9 points) Complete the method for the dequeue operation, **including the precondition check to raise an exception if it is violated.**

```python
def dequeue(self):
    """Removes and returns the Front item of the Queue.
    Precondition: the Queue is not empty.
    Postcondition: Front item is removed from the Queue and returned""

    if len(self._items) == 0:
        raise ValueError("cannot dequeue from empty queue")

    return self._items.pop()
```

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

![Queue Object](image)

**Python List Object**

<table>
<thead>
<tr>
<th>front</th>
<th>rear</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>'d'</td>
</tr>
<tr>
<td>'b'</td>
<td>'c'</td>
</tr>
</tbody>
</table>

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

**Makes \( O(1) \) for enqueue, but dequeue goes from \( O(1) \) to \( O(n) \),**
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:

```
Python List actually used to store heap items
0 1 2 3 4 5 6 7 8
8 15 12 35 24 40 30 100
```

a) (7 points) What would the above heap look like after inserting 10 and then 7 (show the changes on above tree)

Now consider the delMin operation that removes and returns the minimum item.

b) (2 points) What item would delMin remove and return from the above heap?

b) (2 points) What item would delMin remove and return from the above heap?

c) (7 points) What would the above heap look like after a delMin operation? (show the changes on above tree)

d) (3 points) What is the big-oh notation for both of the insert and delMin operations, where n is the number of items in the heap? O(log n) + O(log n)

e) (6 points) Performing 20,000 insertions into an initially empty binary heap takes 0.23 seconds. Now, if we perform 20,000 delMin operations, it takes 0.39 seconds. Explain why 20,000 delMin operations take more time than the 20,000 insert operations?

An inserted item starts as a leaf and percolates up a branch of the tree by comparing with its parent. A delMin cause the "last" item to move to the root and percolates down each level by comparing its two children and then comparing with the smallest child. DelMin needs two comparison to move down a level while insert only needs one.
Question 8. The Node2Way and Node classes can be used to dynamically create storage for each new item added to a Deque using a doubly-linked implementation as in:

DoublyLinkedDeque Object

[Diagram of doubly-linked list with nodes 'a', 'b', 'c', 'd', and a temporary node]

Special case: removing last item

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>( O() )</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</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(1) )</td>
</tr>
<tr>
<td>removeFront</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>_str__</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

(b) (14 points) Complete the removeRear method for the above DoublyLinkedDeque implementation.

```python
class DoublyLinkedDeque(object):
    """ Doubly-linked list based deque implementation. """
    def __init__(self):
        self._front = None
        self._rear = None
        self._size = 0
    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 self._size == 0:
            raise ValueError("Cannot remove from empty Deque")
        temp = self._rear
        self._rear = self._rear._previous
        self._size -= 1
        return temp._data
```

```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
```

```python
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) Why would using singly-linked nodes (i.e., only Node objects with data and next) to implement the Deque lead to poor performance (i.e., cause some Deque operations to have worse big-oh notations)? Justify your answer.

Use singly-linked nodes would cause removeRear to have \( O(n) \).