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

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

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

\[ O(n \log_2 n) \]

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

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

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

\[ O(n^2) \]

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*n):
        doMore(n)

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

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

\[ O(n^5) \]

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

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

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

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

\[ T(10000) = c \cdot 10000^4 = (\frac{1 \text{ sec}}{10^{11}}) \cdot 10^{16} \]

\[ = 10^5 \text{ sec} = 100,000 \text{ sec} \]

Question 5. (8 points) Why should a method/function having a "precondition" raise an exception if the precondition is violated?

To help the programmer immediately know that an error occurs.
Question 6. A FIFO 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 front item is always stored at index 0,
- the rear item is always at index len(self._items) -1 or -1

Queue Object

Python List Object

\text{items: [ ]}

\text{0 \ 1 \ 2 \ 3}

\text{'a' \ 'b' \ 'c' \ 'd'}

\text{front}

\text{rear}

\text{expected}

---

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>_str_</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

b) (6 points) Complete the method for the \texttt{dequeue} operation, \textbf{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 Exception("cannot dequeue from empty queue")

    return self._items.pop(0)
```

c) (6 points) Complete the method for the \texttt{\_str\_} operation,

```python
def \_str\_(self):
    """Returns a string representation of items from front to rear."

    strResult = "(head) "
    for item in self._items:
        strResult += str(item) + "\""

    return strResult + "(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:

![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>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>23</td>
<td>17</td>
<td>34</td>
<td>25</td>
<td>60</td>
<td>90</td>
<td>120</td>
<td>44</td>
<td>28</td>
<td>31</td>
<td>96</td>
<td>84</td>
<td>98</td>
<td>90</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: \( i \times 2 \)
- its right child if it exists: \( i \times 2 + 1 \)
- its parent if it exists: \( i / 2 \)

b) (7 points) What would the above heap look like after inserting 40 and then 20 (show the changes on above tree)

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

![Tree Diagram]

<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>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>23</td>
<td>17</td>
<td>34</td>
<td>25</td>
<td>60</td>
<td>90</td>
<td>120</td>
<td>44</td>
<td>28</td>
<td>31</td>
<td>96</td>
<td>84</td>
<td>98</td>
<td>90</td>
</tr>
</tbody>
</table>

2 c) (2 point) What item would `delMin` remove and return from the above heap? 12

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

e) (6 points) Performing 20,000 `inserts` 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 only needs a single comparison with its parent to move up a level, and it typically only move partially up tree. The `delMin` moves a leaf up to the root so it is likely to percolate down to being a leaf, and at each level 2 comparisons (two child then compare to min child) are needed per level.
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:

DoublyLinkedListe Object

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Big-Oh</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) (16 points) Complete the addRear method for the above DoublyLinkedListe implementation.

```python
class DoublyLinkedListe(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)
    temp.setPrevious(self._rear)
    if self._size == 0:
        self._front = temp
    else:
        self._rear.setNext(temp)
    self._rear = temp
    self._size += 1
```

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. In a singly-linked Deq:

When removeRear is called, resetting the _rear to point to the Node to the "left" of the one being removed is $O(n)$.