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

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

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

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

$O(n^2 \log n)$

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

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

h = 1
while h < 2**(n*n):
    print(h)
    h = h + 1
```

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

$O(2^n)$

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)  # n^2 calls
        doMore(n)  # n^2
        doMore(n)  # n^2

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 $n$?

$O(n^3)$

Question 4. (6 points) Suppose a $O(n^4)$ algorithm takes 10 second when $n = 1000$. How long should the algorithm run when $n = 10,000$?

$T(n) = c \cdot n^4$

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

$T(10000) = c \cdot 10000^4 = c \cdot 10^{16}$

$= 10^{-11} \text{ sec}$

$10^{16} = 10^5 \text{ sec}$

$= 10^{1000000} \text{ sec}$

Question 5. (7 points) Why should a method/function having a "precondition" raise an exception if the precondition is violated?

The method/function is being used incorrectly. By raising an exception, it helps debugging, exactly where the error occurs.
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
_items: []
```
```
Python List Object
0 1 2 3
'd' 'c' 'b' 'a'
   rear    front
```

a) (6 points) Complete the average 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></th>
<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>$O(n)$</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 Exception("Cannot dequeue from an empty queue")
    return self._items.pop(0)
```

c) (5 points) The above `peekRear` code works correctly, but suggest an improvement to the code that makes it more efficient (you can modify the above code with your changes).
Question 7. (4 points) Suppose we want to implement a priority queue with integer priorities such that the smallest integer corresponds to the highest priority. One possible implementation would use a completely unorder Python list such as:

[Diagram of a list with priorities 30, 40, 10, 60, 25, 35]

What would be the big-oh notation for each of the following methods:
- enqueue: $O(1)$ just append to end
- dequeue: $O(n)$ need to check all items to find min, and shift about half of items to fill in the hole where popped

Question 8. Consider an alternate 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:

[Diagram of a binary heap]

a) (7 points) What would the above heap look like after inserting 17 and then 28 (show changes on above tree)

Now consider the binary heap’s delMin operation that removes and returns the minimum item.

b) (1 point) What item would delMin remove and return from the above heap? 14

c) (7 points) What would the heap look like after delMin? (show the changes on the tree that’s just above)

d) (2 points) What would be the $O()$ of a single insert or delMin, where n is the # of items in the heap? $O(\log n)$

e) (6 points). Explain why the average insert operation is faster than the average delMin operation.

- insert only requires one compare per level v.s. two compares for delMin
- inserted item probably percolates up less levels than left node moved to root and percolated down in delMin.
Question 9. The Node2Way and Node classes can be used to dynamically create storage for each new item added to a Stack using a doubly-linked implementation as in:

DoublyLinkedStack Object

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

<table>
<thead>
<tr>
<th>push(item)</th>
<th>pop()</th>
<th>peek()</th>
<th>size()</th>
<th>isEmpty()</th>
<th><strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

b) (19 points) Complete the push and __str__ methods for the above DoublyLinkedStack implementation.

c) (5 points) Explain how we could use singly-linked nodes (i.e., only Node objects with data and next) to implement the stack without degrading performance (i.e., causing some stack operations to have worse big-oh notations)? Justify your answer.

all methods except __str__ are $O(1)$ because push, pop, peek all can use __top__