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

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

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):
        k = 1
        while k < n:
            print(i, j, k)
            k = k + 2
```

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

$O(n^3)$

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

```python
def main(n):
    i = n
    while i > 0:
        for j in range(n):
            doSomething(n)
        i = i // 2

def doSomething(n):
    for k in range(n):
        doMore(n*n)

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

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

$O(n^4 \log n)$

Question 4. (5 points) Suppose a $O(n^2)$ algorithm takes 1 second when $n = 1000$. How long would the algorithm run when $n = 10,000$?

$T(n) = c \cdot n^5$

$T(1000) = c \cdot 1000^5 = 1 \text{ sec}$

$T(10000) = c \cdot 10000^5$

$C = \frac{1}{1000^5} = \frac{1}{10^{15}} \text{ sec}$

$T(10000) = c \cdot 10000^5 = c \cdot 10^{20}$

$\frac{1}{10^{15}} \cdot 10^{20} = 10^5 \text{ sec}$

$= 100,000 \text{ sec}$

Question 5. (8 points) Why should medium/large size programs be written using function definitions instead of a single block of monolithic code written at the top-level (i.e., all statements written outside of any function)?

Medium/large size programs are hard to understand without decomposing into small subtasks. Coding the small subtasks as a function allows it to be tested separately, programmed by another team member, and possibly reused in a later program.
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 rear item is always stored at index 0,
- the front item is always at index len(self._items) - 1 or -1

Deque Object

Python List Object

_aitems:

0 1 2 3
rear 'd' 'c' 'b' 'a'
front

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.

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

b) (9 points) Complete the code for the addRear method, including any precondition check needed by raising an exception if it is violated.

```python
def addRear(self, newItem):
    """Adds the newItem to the rear of the Deque
    Precondition: none
    Postcondition: newItem has been added to the rear of the Deque"
    self._items.insert(0, newItem)
```

c) (10 points) Complete the method for the __str__ operation.

```python
def __str__(self):
    """Returns the string representation of the Deque.
    Precondition: none
    Postcondition: Returns a string representation of the Deque from the
      front item thru the rear item with a blank space between each item."
    
    resultStr = "(front) "
    for index in range(len(self._items) - 1, -1, -1):
        resultStr += str(self._items[index]) + " \\
    resultStr += "(rear)"
    return resultStr
```
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](image)

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></td>
<td>4</td>
<td>13</td>
<td>17</td>
<td>34</td>
<td>25</td>
<td>80</td>
<td>90</td>
<td>120</td>
<td>44</td>
<td>28</td>
<td>31</td>
<td>96</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: 2 * i + 1
- its right child if it exists: 2 * i + 2
- its parent if it exists: \( \lfloor i/2 \rfloor \)

b) (7 points) What would the above heap look like after inserting 10 and then 20 (show the changes on above tree)

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

![Deletion Diagram](image)

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

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) What is the big-oh notation for the delMin operation? (EXPLAIN YOUR ANSWER)

The last item in the list is moved to index 1. At each iteration of the loop, it is swapped with its smaller child at \( 2 * i \) or \( 2 * i + 1 \), so its index at least doubles. Since we can only double \( \log_2 n \) times before reaching \( n \), we only loop \( O(\log_2 n) \) times.
Question 8. The Node class can be used to dynamically create storage for each new item added to a Stack using a singly-linked implementation as in:

```
<table>
<thead>
<tr>
<th>LinkedStack Object</th>
<th>Node Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>_size: A</td>
<td>data next</td>
</tr>
<tr>
<td>_bottom:</td>
<td>d</td>
</tr>
<tr>
<td>_top:</td>
<td>data next</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>data next</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>
```

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

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

b) (12 points) Complete the `push` method for the above LinkedStack implementation.

```python
class LinkedStack(object):
    """Singly-linked list based Stack implementation."""
    def __init__(self):
        self._size = 0
        self._bottom = None
        self._top = None
    def push(self, item):
        """Adds the item to the top of the Stack."
        Precondition: none
        temp = Node(item)
        if self._size == 0:
            self._bottom = temp
            self._top = temp
            self._size += 1
        else:
            self._top.setNext(temp)
            self._top = temp
            self._size += 1
```

c) (7 points) Suggest an improvement to the above implementation to speed up some of the stack operations enough to change their big-oh notation? (Justify your answer)

1. Switch "direction" of nodes:
   - Both pop + push are $O(1)$

2. Use doubly-linked (Node2Way) Nodes:
   - Both pop + push are $O(1)$

3. Use Python list with bottom at index 0.