Thomas Landi

Question 1-
a)

Question 2-
Part 1:

1 - 100^100, 1 (Both of these are constant time, so they have the same growth rate)
2 - log(n)
3 - n + 5
4 - n log(n)
5 - n log(n^2)
6 - n^100
7 - 2^n, 2^2n (The 2 in the exponent is just a constant, so it will not affect the growth rate)
8 - n!

Part 2:

The function fun returns the index of the first character that is less than or equal to the character before it in the string, e.g. a <= b (returning the size of the string if none are found).

The best time complexity for this function is constant. This would be the case where the string is one character followed by a less than or equal character, e.g. “ba…” as the function would return on the first iteration.

The worst time complexity would be linear. This would be where every character is followed by a character greater than it, e.g. “abcdefghijklmnopqrstuvwxyz.” as it has to iterate through the entire array.

Question 3-

Part 1:

Two queues are needed to implement the operations, as retrieving the top element (either for removal or peeking) requires emptying the queue first into another for storage.

```
def push(x):
        internalQueue.enqueue(x)

def pop():
        if (internalQueue.empty()):
                return null:
        tempQueue = Queue()
        for i in range(1, internalQueue.size() - 1):
                tempQueue.enqueue(internalQueue.dequeue())
        returnVal = internalQueue.dequeue()
        while (tempQueue.peek() is not null):
                internalQueue.enqueue(tempQueue.dequeue())
        return returnVal

def peek():
        if (internalQueue.empty()):
                return null:
        tempQueue = Queue()
        for i in range(1, internalQueue.size() - 1):
                tempQueue.enqueue(internalQueue.dequeue())
        returnVal = internalQueue.peek()
        tempQueue.enqueue(internalQueue.dequeue())
        while (tempQueue.peek() is not null):
                internalQueue.enqueue(tempQueue.dequeue())
        return returnVal

def empty():
        If (internalQueue.size() == 0):
                return true
        return false
```

push(1):
Internal Queue: | 1 |

push(2):
Internal Queue: | 2 || 1 |

push(3):
Internal Queue: | 3 || 2 || 1 |

pop():
Internal Queue: | 3 | -> Temp Queue | 2 || 1 |
Remove 3 from internal Queue and store it
Temp Queue: -> Internal Queue | 2 || 1 |
Return 3

peek():
Internal Queue: | 2 | -> Temp Queue | 1 |
Store value (2) currently in the Internal Queue
Internal Queue | 2 | -> Temp Queue | 2 || 1 |
Temp Queue -> Internal Queue | 2 || 1 |
Return 2

Part II:

Two stacks will be necessary here as well, for similar reasons (one will be for storage, the other will allow for the destacking/restacking needed by the peek/dequeue functions)

```
def enqueue(x):
        internalStack.push(x)

def dequeue():
        if (internalStack.empty()):
                return null
        tempStack = Stack()
        for i in range(1, internalStack.size() - 1):
                tempStack.push(internalStack.pop())
        returnVal = internalStack.pop()
        while (tempStack.peek() is not null):
                internalStack.push(tempQueue.pop())
        return returnVal
```

```
def peek():
        if (internalStack.empty()):
                return null
        tempStack = Stack()
        for i in range(1, internalStack.size() - 1):
                tempStack.push(internalStack.pop())
        returnVal = internalStack.peek()
        tempStack.push(internalStack.pop())
        while (tempStack.peek() is not null):
                internalStack.push(tempQueue.pop())
        return returnVal

def empty():
        if (internalStack.size() == 0):
                return true
        return false
```

enqueue(1):
Internal Stack: | 1 |

enqueue(2):
Internal Stack: | 1 || 2 |

enqueue(3):
Internal Stack: | 1 || 2 || 3 |

dequeue():
Internal Stack: | 1 | -> Temp Stack | 3 || 2 |
Remove 1 from the Internal Stack and store it
Temp Stack: -> Internal Stack | 2 || 3|
Return 1

enqueue(4):
Internal Stack: | 2 || 3 || 4 |

Question 4-

```
def isBST(root):
        left = root.getLeftNode()
        right = root.getRightNode()
        if left.val > root.val  or root.val > right.val:
                return 0
        leftResult = isBST(left)
        rightResult = isBST(right)
        if leftResult == 1 and rightResult == 1:
                return 1
        return 0
```

The worst case time complexity for the algorithm is linear, as at it's worst case (it being a BST), it has to traverse the entire tree once.

Question 5-

Pseudocode:

Variables for keeping track of our progress in the male list, the female list, the count of couples so far, and our list of 'checkpoints' (males we haven't fully evaluated)

Loop while we don't have enough couples:
    Evaluate the loyalty at our current male/female marker
    If we have any previous males in our checkpoints:
        Loop through checkpoints:
            If any are more loyal than the current loyalty, add them
            Add the next couple in line from that checkpoint to the heap of checkpoints
    If we are still not at our count limit:
        Evaluate the loyalty for the next male with the first female
        If the current loyalty is better:
            add that couple to the list and increment the female counter
        Else:
            add the next couple and increment the male count:
            Add the current male and female counts to the checkpoint heap

This algorithm will run at most k log k time, as in the best case scenario, as it will never have to visit a male female couple more than once, and the couple list will be appended exactly k times. The only process being ran within the loop that will increase the time complexity is the sorting of the 'checkpoints,' however this is being performed with a heap sort, which will run at log k times, so in the worst case scenario, where every male gets a checkpoint, it will run k log k times.