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

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

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

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

```
i = 1
while i < n:

    for j in range(n):
        print(j)

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

    i = i * 2
```

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

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

```
def main(n):
    for i in range(n):
        doSomething(n)
        doMore(n)

def doSomething(n):
    for k in range(2**n):
        print(k)

def doMore(n):
    for k in range(n):
        print(k)

main(n)
```

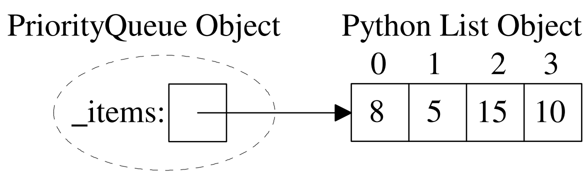
What is the big-oh notation $O(\ )$ for this code segment in terms of n?

Question 4. (8 points) Suppose a $O(n^4)$ algorithm takes 1 second when n = 100. How long would you expect the algorithm to run when n = 1,000?

Question 5. (5 points) In lab 2 (and on the Python Summary) the AdvancedDie class inherited from the Die class. How does inheritance aid a programmer in writing code?

Question 6. A ***priority queue*** has the same operations as a regular queue, except the items are NOT returned in the FIFO (first-in, first-out) order. Instead, each item has a proirity that determines the order they are removed.
One possible implementation of a priority queue would be to use a built-in Python list to store the items such that

- items in the Python list are **unordered** by their priorities,
- lowest number indicates the highest priority (i.e., dequeuing from the below priority queue would return 5)



a) (5 points) Complete the big-oh *O* ( ), for each PriorityQueue operation, assuming the above implementation.
Let n be the number of items in the PriorityQueue.

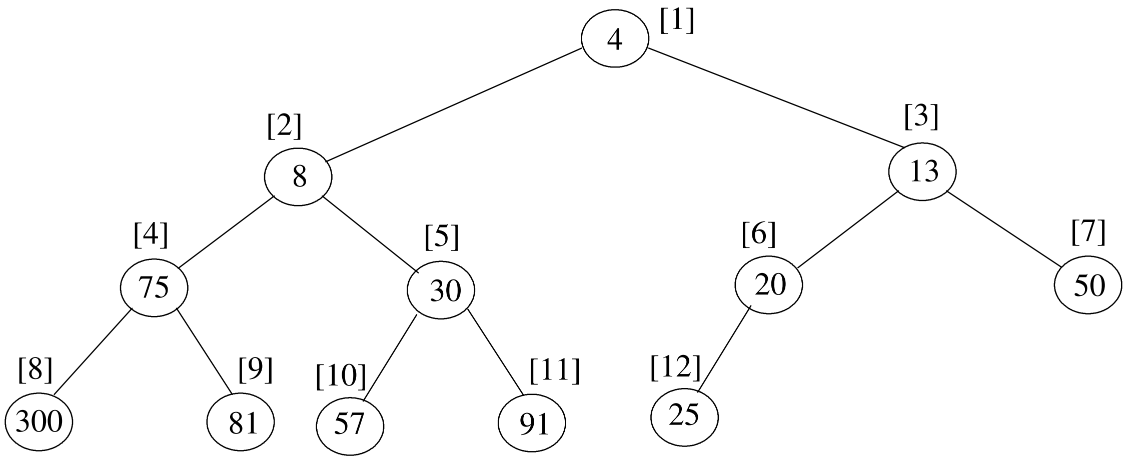
isEmpty	enqueue(item)	dequeue	__str__	size

b) (15 points) Complete the method for the `dequeue` operation including the precondition check.

```
class PriorityQueue(object):

   def __init__(self):
      self._items = []

   def dequeue(self):
       """Removes and returns the highest priority (lowest value) item in the
          PriorityQueue

          Precondition:  the PriorityQueue is not empty.
          Postcondition: the highest priority (lowest value) item in the PriorityQueue is
                         removed and returned"""
```

c) (5 points) Suggest an alternate PriorityQueue implementation to speed up some of its operations.

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 arranges 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

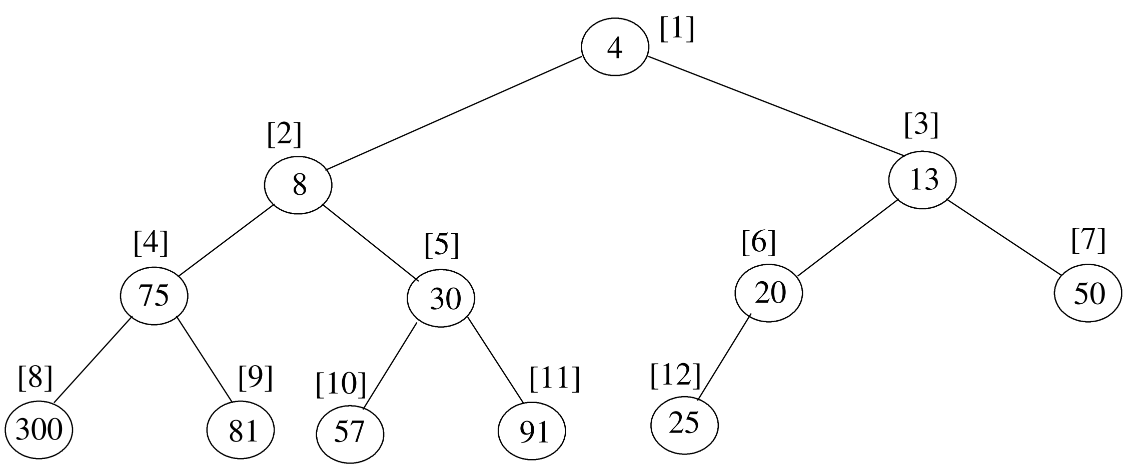
0	1	2	3	4	5	6	7	8	9	10	11	12
not used	4	8	13	75	30	20	50	300	81	57	91	25

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:
- its right child if it exists:
- its parent if it exists:

b) (7 points) What would the above heap look like after inserting 12 and then 2 (show the changes on above tree)

c) (3 points) What is the big-oh notation for inserting a new item in the heap?

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



Python List actually used to store heap items

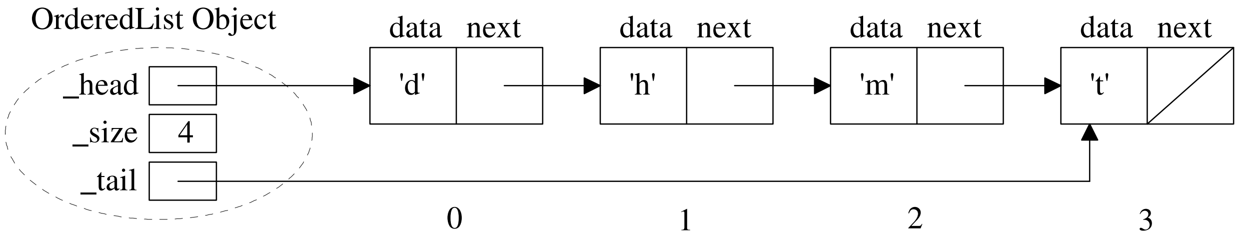
0	1	2	3	4	5	6	7	8	9	10	11	12
not used	4	8	13	75	30	20	50	300	81	57	91	25

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

e) (7 points) What would the above heap look like after `delMin`? (show the changes on above tree)

f) (3 points) Why does a `delMin` operation typically take longer than an `insert` operation?

Question 8. The textbook's **Ordered list** ADT uses a singly-linked list implementation. I added the `_size` and `_tail` attributes:



a) (15 points) The `index(item)` method returns the position of the `item` in the list (e.g., 'm' is at position 2). Recall that the textbook's implementation, assumes the `item` is in the list!!! Thus, the precondition is that `item` is in the list. Complete the `index(item)` method code including the precondition check.

```
class OrderedList(object):

   def __init__(self):
      self._head = None
      self._size = 0
      self._tail = None

   def index(self, item):
```

```
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
```

b) (10 points) Assuming the ordered list ADT described above **does not allows duplicate items**. Complete the big-oh O ( ) for each operation. Let n be the number of items in the list.

`add(item)` adds the item into the list	`pop()` removes and returns tail item	`length()` returns number of items in the list	`remove(item)` removes the item from the list	`index(item)` returns the position of item in the list