Implement factorial function

```
def factorials_sum_list(n)
```

Give a recursive implementation. When called , the function should create and return a list, with the squares of the first n positive integers.

```
def squared_list(n)
```

Insert value in a sorted way in a sorted doubly linked list

```
def sortedInsert(self, data):
```

Delete a node in a Doubly Linked List

```
class Node:
	def __init__(self,data):
		self.data = data
		self.next = None
		self.prev = None
class DoublyLinkedList:
	def __init__(self):
		self.head = None
	def deleteNode(self, dele):
```

Reverse a doubly linked list

```
def reverse(self):
```

Find pairs with given sum in doubly linked list

```
def pairSum(head, x):
```

Delete a Doubly Linked List node at a given position

```
def deleteNodeAtGivenPos(head_red, n):
```

Remove duplicates from a sorted doubly linked list

```
def removeDuplicates(head_ref):
```

Delete all occurrences of a given key in a doubly linked list

```
def deleteAllOccurOfX(head,x):
```

Sorted insert in a doubly linked list with head and tail pointers

```
def insert_sorted(srt_lnk_lst, elem):
```

Find the largest node in Doubly linked list

```
def LargestInDll(head_ref):
```

Convert a given Binary Tree to Doubly Linked List

Given the head of a singly linked list, return true if it is a palindrome

```
def isPalindrome(head):
```

Reverse Linked List II

```
def reverseBetween(self, head, m,n):
```

Remove Duplicates in a sorted linked list

```
def removeDuplicates(self):
```

Midterm #2/ 2018

Q1

```
def move_to_end(self, node)
```

Q2

In this question, we will suggest a way to compactly represent a string that has a lot of repetitions of successive characters. We will represent such a string using a Doubly Linked List object, where each maximal sequence of the same character in consecutive positions, will be stored as a single tuple containing the character and its (positive) count.

```
def get_char(c_str, i):
```

Q3

Implement the following function: This function gets c_str, a DoublyLinkedList that stores a non-empty string in a compact representation as described above. In addition, the function expects a character new_ch.

```
c_str = [('a', 5),('b', 3),('a', 3), ('c', 1)]
append_char(c_str, 'c')
c_str = [('a', 5),('b', 3),('a', 3), ('c', 2)]

def append_char(c_str, new_ch):
```

Infix to Postfix conversion using stack

```
def infixToPostfix(self, exp):
```

Reverse a stack using recursion

```
def reverse(stack):
```

Reverse a stack without using extra space

```
def reverse(self):
```

Move last element to front of a given Linked List

def moveToFront(self):

Reverse a string using a stack

```
def reverse(string):
```

Reversing a queue using recursion

```
def reverse_queue(queue):
```

Midterm #2/2018

Q4

Implement the following functions:

def alternating_parity

This function is called with a lst, containing 2n positive integers. Half of the numbers in lst are even and half are odd.
When called, it should reorder the elements in lst, so that at the end, the elements will be ordered in lst with alternating parity. Relative order of the even numbers and the relative orders of the odd numbers should remain the same as they were originally in lst.

```
def alternating_parity(lst):
```

Q5

A Flippable-stack is an abstract data type that is like a regular stack, but in addition, it allows to flip (reverse) the order of the elements that are currently in it, so that the element at the bottom would become the top element, the second from the bottom would become the second top, etc.

- A Flippable-stack has the following operations:
- FlippableStack(): creates new FlippableStack object, with no elements in it
- is_empty(): returns false if there are one or more items in the FlippableStack; true if there are no items in it
- push(item): inserts a new item at the top of the FlippableStack
- pop(): removes and returns the item that is at the top of the FlippableStack
- top(): returns (without removing) the item that is at the top of the FlippableStack
- flip(): flips the order of the items that are currently in the Flippable Stack
-
- Complete the implementation of the FlippableStack class
- runtime requirement: All FlippableStack operations should run in O(1) worst-case
- Notes:
- You may use data types we implemented in class (such as ArrayStack, ArrayQueue, DoublyLinkedList), as data members in your implementation
- Make sure to choose the most suitable data type, so you could satisfy the runtime requirement
- You can’t change the implementation of any of these data types. You may only use them
- Make sure that your implementation for the flip method runs in constant worst-case time. As a friendly advice, you shouldn’t change the actual order of all items, as that would take too much time.
- If you need more space than what is provided, you are probably over complicating the implementation. However, in any case, do not write on the back of any page

```
class FlippableStack:
      def __init(self):
```

```
      def __len__(self):
```

```
      def is_empty(self):
            return (len(self) ==0)

      def push(self, item):
```

```
      def pop(self):
            if (self.is_empty()):
                  raise Exception(“FlippableStack is empty”)
```

```
      def top(self):
            if (self.is_empty()):
                  raise Exception(“FlippableStack is emptyI ”)
```

```
    def flip(self):
```

Give a Python implementation for the MaxStack ADT. The MaxStack ADT supports the following operations:

- MaxStack(): initializes an empty MaxStack object
- maxS.is_empty(): returns True if maxS does not contain any elements, or False otherwise
- len(maxS): Returns the number of elements in maxS
- maxS.top(): Returns a reference to the top element of maxS, without removing it; an exception is raised if maxS is empty
- maxS.pop(): removes and return the top element from maxS; an exception is raised if maxS is empty
- maxS.max(): returns the element in maxS with the largest value, without removing it; an exception is raised if maxS is empty

Note: Assume that the user inserts only integers to this stack (so they could be compared to one another and a maximum date is well defined).

For example, your implementation should follow the behavior below:

```
>>> maxS.MaxStack()
>>> maxS.push(3)
>>> max.push(1)
>>> max.push(6)
>>> max.push(4)
>>> max.max()
6
>>> max.pop()
4
>>> max.max()
3
```

```
def __init__(self):
```

```
def __len__(self):
```

```
def is_empty
def push(self):
def top(self):
```

Midterm #2/ 2019

Q4

implement the following function:
def move_val_to_front(q,val)

This function is called with:
1. q - a Queue object containing integers
2. val - an integer

When called, it should reorder the numbers in q, so that when the function is done, all occurrences of val will be in the front of the queue.

```
def move_val_to_front(q,val):
```

Q5

A Parties Queue is a variation of a Queue. It is used to apply a first-in-first-out order, but instead of storing individual items as elements, it stored parties (collection of items) as elements. It also supports an additional operation that allows to add items to the part that is last in line.

A Parties-Queue has the following interface:

- pq = PartiesQueue(): created a new PartiesQueue object, with no parties in it
- pq.enq_party(party_size)
- pq.first_party(): returns the size of the party that is first in line
- pq.deq_first_party(): removes and returns the size of the party that is first in line
- pq.add_to_last_part(size_to_add): mutates the object to reflect an addition of size_to_add guests to the party that is last in line
- pq.is_empty(): returns true if and only if there are no parties in line

```
class PartiesQueue:
      def __init__(self):
```

```
      def enq_party(self, party_size)L
```

```
def add_to_last_party(self, size_to_add):
```

```
def first_party(self):
      if(self.is_empty()):
            raise Exception(“Parties is empty”)
```

```
def is_empty(self):
```