Data Structures and Algorithms

How to reverse singly linear linked list?

```
void reverse() {
    Node oldhead = head;
    head = null;
    while(oldhead != null) {
        // delete first node from old list
        Node temp = oldhead;
        oldhead = oldhead.next;
        // add that node at the start of new list
        temp.next = head;
        head = temp;
    } // repeat until old list is empty
}
```

```
void reverse() {
    Node *oldhead = head;
    head = NULL;
    while(oldhead != NULL) {
        // delete first node from old list
        Node temp = oldhead;
        oldhead = oldhead->next;
        // add that node at the start of new list
        temp->next = head;
        head = temp;
    } // repeat until old list is empty
}
```

How to reverse singly linear linked list using recursion?

```
Node reverse(Node cur) {
    // if cur node is last node of list, then head to that node
    if(cur.next == null) {
        head = cur;
        return cur;
    }
    // reverse the rest of the list
    Node tail = reverse(cur.next);
    // add cur node to the next of tail
    tail.next = cur;
    // make cur next null
    cur.next = null;
    // cur node is last node for the prev call
    return cur;
}
Node reverse() {
    if(head != null)
        reverse(head);
}
```

How to reverse singly linear linked list using recursion -- with single pointers?

```
Node reverse(Node cur) {
    // if cur node is last node of list, then head to that node
    if(cur.next == null) {
        head = cur;
        return cur;
    }
    // reverse the rest of the list and add cur node to the next of tail
```

```
    reverse(cur.next).next = cur;
    // make cur next null
    cur.next = null;
    // cur node is last node for the prev call
    return cur;
}
Node reverse() {
    if(head != null)
        reverse(head);
}
```

```
Node reverse(Node cur) {
    if(cur.next == null)
        head = cur;
    else
        reverse(cur.next).next = cur;
    cur.next = null;
    return cur;
}
Node reverse() {
    if(head != null)
        reverse(head);
}
```

Display singly linear linked list in reverse order.

- Time: $O(n^2)$

```
int count = 0;
// count number of nodes
for(Node trav = head; trav != null; trav = trav.next)
```

```
    count++;
while(count > 0) {
    // traverse upto count position
    Node trav = head;
    for(int i=1; i<count; i++)
        trav = trav.next;
    // display that node
    System.out.println(trav.data);
    // decrement count
    count--;
} // repeat until count is 0
```

Display singly linear linked list in reverse order.

- Time: O(n)
- Aux Space: O(n)

```
// push all elements on stack
Stack<Integer> s = new Stack<>();
for(Node trav=head; trav!=null; trav=trav.next)
    s.push(trav.data);
// pop all elements from stack and print them -- one by one
while(!s.isEmpty()) {
    int ele = s.pop();
    System.out.println(ele);
}
```

Display singly linear linked list in reverse order -- using recursion.

```

```

```
void reverseDisplay(Node cur) {
    // if cur node is null, do nothing
    if(cur == null)
        return;
    // display rest of the list
    reverseDisplay(cur.next);
    // display the current node
    System.out.println(cur.data);
}
void reverseDisplay() {
    reverseDisplay(head);
}
```

Display a given range of numbers in reverse order -- using recursion.

```
void revDisplay(int s, int e) {
    // if all nums are done, do nothing
    if(s > e)
        return;
    // display next numbers
    revDisplay(s+1, e);
    // display current number
    System.out.println(s);
}
```

Find the middle node of singly linear linked list.

- Time: O(n) -- Needs n/2 iterations

```
Node findMid() {
    Node slow = head, fast = head;
    while(fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
```

Find the middle node of singly linear linked list using single pointer.

- Time: O(n) -- But needs n+n/2 iterations
- Traverse till last node and count the number of nodes.
- Traverse till count/2 and print it.

Find the middle node of singly linear linked list using recursion.

- Refer slides

Check if singly linear list contains a loop.

```
boolean hasLoop() {
    Node slow = head, fast = head;
    while(fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if(slow == fast)
            return true;    // loop
    }
    return false;   // no loop
}
```

Check if given singly linear list is palindrome or not.

- Time: O(n)
- Space: O(n)

```
boolean isPalindrome() {
    Stack<Integer> s = new Stack<>();
    for(Node trav=head; trav!=null; trav=trav.next)
        s.push(trav.data);
    for(Node trav=head; trav!=null; trav=trav.next)
        if(trav.data != s.pop())
            return false; // not palindrome
    return true;
}
```

Check if given singly linear list is palindrome or not.

- Find the middle of the list -- fast and slow pointer.
- Reverse the second half of the list.
- Compare first half of the list with second half of the list (reversed). If same, palindrome, otherwise not.

Create a stack (LIFO) using queue (FIFO).

- Stack operations: push(), pop(), peek(), isEmpty().

```
class Stack {
    private Queue<Integer> main = new LinkedList<>();
    private Queue<Integer> temp = new LinkedList<>();
    // time: O(n)
    public void push(int val) {
```

```
            while(!main.isEmpty()) {
                int ele = main.poll(); // pop
                temp.offer(ele); // push
            }
            main.offer(val); // push
            while(!temp.isEmpty()) {
                int ele = temp.poll(); // pop
                main.offer(ele); // push
            }
        }
        // time: O(1)
        public int pop() {
            return main.poll(); // pop
        }
        // time: O(1)
        public boolean isEmpty() {
            return main.isEmpty();
        }
    }
```

Create a queue using stack.

- Homework

Input a string from user and find the character repeated maximum number of times (irrespective of case).

- Hint: Use Hashing.

Print the length of highest continous number range in given array.

- Input: 8, 2, 9, 1, -3, 5, 4, 3, 7, -2
- Output: 5 (because highest continous range: 1, 2, 3, 4, 5 -- 5 elements)
- Hint: Use Hashing (HashSet).

```java
static int findLongestConseqSubseq(int arr[]) {
    HashSet<Integer> set = new HashSet<>();
    for(int ele: arr)
        set.add(ele);
    int maxseq = 0;
    for(int ele: set) {
        int cnt = 0;
        while(set.contains(ele)) {
            cnt++;
            ele++; // check next element
        }
        if(cnt > maxseq)
            maxseq = cnt;
    }
    return maxseq;
}
```