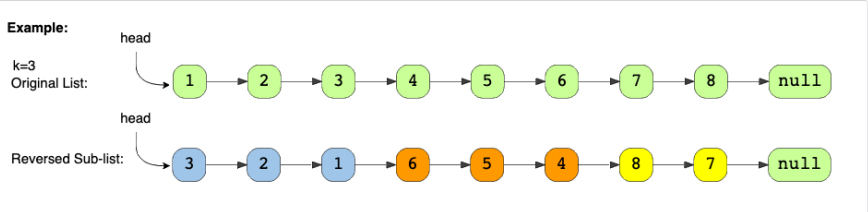
Reverse every K-element Sub-list (medium)

Problem Statement

Given the head of a LinkedList and a number 'k', **reverse every 'k' sized sub-list** starting from the head.

If, in the end, you are left with a sub-list with less than 'k' elements, reverse it too.



Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
class Node {
  constructor(value, next=null){
    this.value = value;
    this.next = next;
  }

  get_list() {
    result = "";
    temp = this;
    while (temp !== null) {
      result += temp.value + " ";
      temp = temp.next;
    }
    return result;
  }
};


const reverse_every_k_elements = function(head, k) {
  // TODO: Write your code here
  return head;
}


head = new Node(1)
head.next = new Node(2)
head.next.next = new Node(3)
```

RUN SAVE RESET

Solution

The problem follows the **In-place Reversal of a LinkedList** pattern and is quite similar to Reverse a Sub-list. The only difference is that we have to reverse all the sub-lists. We can use the same approach, starting with the first sub-list (i.e. `p=1, q=k`) and keep reversing all the sublists of size 'k'.

Code

Most of the code is the same as Reverse a Sub-list; only the highlighted lines have a majority of the changes:

Java | Python3 | C++ | JS

```
class Node {
  constructor(value, next = null) {
    this.value = value;
    this.next = next;
  }

  print_list() {
    let temp = this;
    while (temp !== null) {
```

```
      process.stdout.write(`${temp.value} `);
      temp = temp.next;
    }
    console.log();
  }
}


function reverse_every_k_elements(head, k) {
  if (k <= 1 || head === null) {
    return head;
  }

  let current = head,
    previous = null;
  while (true) {
    const last_node_of_previous_part = previous;
    // after reversing the LinkedList 'current' will become the last node of the sub-list
    const last_node_of_sub_list = current;
```

RUN SAVE RESET

Close

Output 2.443s

```
Nodes of original LinkedList are: 1 2 3 4 5 6 7 8
Nodes of reversed LinkedList are: 3 2 1 6 5 4 8 7
```

Time complexity

The time complexity of our algorithm will be $O(N)$ where 'N' is the total number of nodes in the LinkedList.

Space complexity

We only used constant space, therefore, the space complexity of our algorithm is $O(1)$.