Exercise 5 - Solutions

1. Input: list L

```
S = Stack()
for i in range(len(L)):
    S.push(L[i])
for i in range(len(L)):
    L[i] = S.pop()
```

2. $32 - (15 - 5) = 22$

3. It is useful to first look at a recursive version of an algorithm that generates all permutations of a list L. In Python-like pseudocode:

```
def permutations(L, level):
    if we have reached the final level = len(L) - 1:
        the current state of L is the only possibility so output it as one of the permutations
    else:
        while current element L[level] has not been treated yet:
            call permutations recursively on the tail of L from level+1
            cyclically rotate the tail of the list from level to the end
                (so that each of the elements in the tail eventually moves to position L[level])
```

A non-recursive implementation needs to keep track of the count of the cyclic rotations at the current level. So instead of calling recursively we push the current rotation count on the stack and start a new count at the next level. When the next level is exhausted, we pop the old count off the stack, increment and start over.

→ See complete Python codes for the recursive and equivalent non-recursive version on the class web page.

4. See Python code on class web page.

5. We need to find the predecessors of x and y. In Python-like pseudo-code:

```
Let v be the starting sentinel node of L
x_pred = y_pred = None
while x_pred is None or y_pred is None:
    if v.next == x:
        x_pred = v
    if v.next == y:
        y_pred = v
    v = v.next
```

} In worst case, runs in $\Omega(n)$

```
x_pred.next, y_pred.next, x.next, y.next = y, x, y.next, x.next
```

If list is doubly-linked, the predecessors can be found by following a pointer at $O(1)$ run-time.

6. For simplicity, assume that L is not empty.

```
v = L.start()
c = 1
while not (v.next() is L.start()):
    c += 1
    v = v.next()
```

7. Accesses all the elements in the original order.

8. Accessing each of the elements once before accessing again is taking $\Omega(n^2)$ time.

Repeating this access pattern n times over will thus incur $\Omega(n^3)$ time

9. For simplicity assume that L is not empty.

```
prev = None
while L.start.next is not None:
    L.start, L.start.next, prev = L.start.next, prev, L.start
L.start.next = prev
```