Exercise 1: Pancake Sort

You have a plate filled with pancakes of different sizes, and you want to order them in descending order of size. E.g., let $A = [3, 2, 1, 5]$ represent the pancakes of respective sizes, where the lowest pancake has size 3, the one on top of it has size 2 etc. However, you can only reorder the pancakes by inserting a spatula between two pancakes (or below the lowest pancake) and flipping the pancakes above the spatula. We represent this as operation $\textsc{Flip}(A[i \dots n])$, where n is the length of the array A. In our example, $\textsc{Flip}(A[2 \dots 4]) = [3, 5, 1, 2]$.

Come up with an algorithm that sorts the pancakes in descending order, using as few Flip operations as you can find.[1] Start with the ideas behind the "selection sort" algorithm.

What is the worst-case "flip complexity", i.e. number of Flip calls needed, of your algorithm in terms of n? State the actual number, do not use big-O notation.

Exercise 2: Linked List

Write a `LinkedList` class, and the required `Node` class, as `R6` classes. It should be a doubly linked list, which should implement the following methods and have the corresponding asymptotic worst-case time complexity costs:

- `append(x)` ($O(1)$): Append an element x to the end of the list.
- `insert(x, i)` ($O(i)$): Insert an element x at index i in the list. Items that were at index i or higher are shifted to the right (their index is incremented by 1). i must be between 1 and `length() + 1`; throw an error if it is not.
- `pop()` ($O(1)$): Remove, and return, the last element from the list. Throw an error if the list is empty.
- `remove(i)` ($O(i)$): Remove, and return, the element at index i from the list. Items that were at index $i + 1$ or higher are shifted to the left (their index is decremented by 1). Throw an error if i is out of bounds.
- `get(i)` ($O(i)$): Get the element at index i in the list, or throw an error if i is out of bounds.
- `find(x)` ($O(n)$, where n is the length of the list): Return the index i of the first occurrence of x, i.e., where `identical(x, get(i))` is `TRUE`, or 0 if x is not in the list.
- `length()` ($O(1)$): Get the length of the list.
- `asList()` ($O(n)$): Return a `list` of the elements in the list, in order.
- `reverse()` ($O(n)$): Return a copy of the list with items in reverse order. Do not worry about "deep copies" for this exercise.
- `subset(i, j)` ($O(j)$, assuming $i \leq j$): Return a copy of the list with items in the range from index i to j.

Indexing should be 1-based, i.e., the first element has index 1, the last element has index `length()`. Make sure the following code runs and gives the expected output:

```
ll <- LinkedList$new()
ll$pop()
#> Error in ll$pop(): The list is empty.

ll$append(1)
ll$append("b")
ll$insert("a", 2)

ll$length()
```

[1] Try to find a good simple algorithm, finding the true optimum is in fact an NP-hard problem.

```r
#> [1] 3

ll$pop()
#> [1] "b"

ll$append("c")
ll$append(c(1, 2, 3))

ll$remove(2)
#> [1] "a"

ll$get(2)
#> [1] "c"

ll$find(c(1, 2, 3))
#> [1] 3

ll$asList() |> deparse() |> cat(sep = "\n")
#> list(1, "c", c(1, 2, 3))

ll$reverse()$asList() |> deparse() |> cat(sep = "\n")
#> list(c(1, 2, 3), "c", 1)

ll$subset(2, 3)$asList() |> deparse() |> cat(sep = "\n")
#> list("c", c(1, 2, 3))
```

Some practical notes:

(a) You should write a *doubly* linked list. Out of the methods asked of you for this exercise, only the `pop()` method strictly requires a doubly linked list to achieve the desired $O(1)$ time complexity.

(b) The methods required for this class do not expose the internal `Node` objects. However, it may be a good idea to write the following internal (possibly `private`) helper methods that do:

- `.getNodeAt(i)`: Get the `Node` object at index i.
- `.insertInPlaceOf(node.new, node.place)`: Insert `node.new` in place of `node.place`; `node.place` and all subsequent nodes are shifted to the right.
- `.remove(node)`: Remove `node` from the list; all nodes after `node` are shifted to the left.

(c) A linked list in R6 is *really slow*! It is a good practice exercise, but you should be aware that in almost all cases where you need "linked-list-like" behavior, you should instead either just use a normal `list` (even with thousands of elements), or use a specialized package that implements the data structure in C(++).