Solution 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.

The strategy is a variation of selection sort. We iterate from the bottom of the stack up, placing the correct pancake at each position. For each position i (from 1 to $n-1$), we find the largest pancake in the unsorted part of the stack ($A[i \dots n]$) and move it to position i. This requires two flips in the worst case: one to bring the desired pancake to the top of the stack, and a second to move it to its correct position.

Algorithm 1 *PancakeSort*(A)

```
1: n ← LENGTH(A)
2: for i ← 1 to n − 1 do
3:     Find index k such that A[k] is the maximum value in A[i ... n].
4:     if k ≠ i then
5:         if k ≠ n then
6:             A ← FLIP(A[k ... n])          ▷ Bring the maximum element to the top of the unsorted stack.
7:         end if
8:         A ← FLIP(A[i ... n])              ▷ Move the maximum element to its correct position i.
9:     end if
10: end for
11: return A
```

Worst-Case Flip Complexity We analyze the number of flips required in the worst case for an array of size n. The algorithm iterates from $i = 1$ to $n - 1$.

- For each i from 1 to $n - 2$: In the worst case, the maximum element in $A[i \dots n]$ is neither at the correct position (i) nor at the top of the unsorted stack (n). This requires two flips:
  (a) One flip to bring the element to the top of the unsorted part of the stack.
  (b) A second flip to move it into position i.

  This gives 2 flips for each of the $n - 2$ iterations.
- For the last iteration, $i = n - 1$: The unsorted portion is $A[n-1 \dots n]$. The maximum element is either at position $n - 1$ (0 flips) or at position n. If it is at position n, it takes one flip, $\textsc{Flip}(A[n-1 \dots n])$, to sort it. So, the worst case for this step is 1 flip.

The total worst-case number of flips is the sum of the worst-case flips for each step:

$$\text{Total Flips} = \underbrace{2 \times (n-2)}_{\text{for } i=1 \dots n-2} + \underbrace{1}_{\text{for } i=n-1} = 2n - 4 + 1 = 2n - 3$$

So, the worst-case flip complexity of this algorithm is $2n - 3$ flips.

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

Note on Optimality The problem of finding the absolute maximum number of flips for any given stack of pancakes is known as the "pancake problem", and finding an optimal solution is an NP-hard problem. Bill Gates, of microchips-in-covid-vaccines fame, has published on it: Gates, W. H., & Papadimitriou, C. H. (1979). Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27(1), 47–57. `https://doi.org/10.1016/0012-365X(79)90068-2`. (Although this result has since been improved upon).

Solution 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] 3

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

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

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

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

```r
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(++).

There are many ways to implement a linked list. The following is an example, but it makes various choices, for example having one `sentinel` node, which could be made differently.

```r
Node <- R6::R6Class(
  "Node",
  public = list(
    value = NULL,
    prv = NULL,
    nxt = NULL,
    initialize = function(value) {
      self$value <- value
    }
  )
)

LinkedList <- R6::R6Class(
  "LinkedList",
  public = list(
    initialize = function() {
      private$.sentinel <- Node$new(NULL)
      # the sentinel points to itself as prv and nxt, so when we insert a node,
      # into the empty list, the new node's prv and nxt pointers are set
      # correctly.
      private$.sentinel$prv <- private$.sentinel$nxt <- private$.sentinel
    },
    append = function(x) {
      node <- Node$new(x)
      private$.insertInPlaceOf(node, private$.sentinel)
      invisible(self)  # returning invisible(self) allows chaining of methods
```

```
  },
  insert = function(x, i) {
    checkmate::assertCount(i, tol = 0)
    # Do asserts, before doing anything else; otherwise the following gives
    # weird error messages, e.g. when `i` is NULL.
    if (i < 1 || i > private$.length + 1) {
      stop("Index out of bounds.")
    }

    if (i == private$.length + 1) {
      self$append(x)
    } else {
      node <- Node$new(x)
      private$.insertInPlaceOf(node, private$.getNodeAt(i))
    }
    invisible(self)  # returning invisible(self) allows chaining of methods
  },
  pop = function() {
    if (private$.length == 0) {
      stop("The list is empty.")
    }
    node <- private$.sentinel$prv
    private$.remove(node)
    node$value
  },
  remove = function(i) {
    checkmate::assertCount(i, tol = 0)
    if (i < 1 || i > private$.length) {
      stop("Index out of bounds.")
    }
    node <- private$.getNodeAt(i)
    private$.remove(node)
    node$value
  },
  get = function(i) {
    checkmate::assertCount(i, tol = 0)
    if (i < 1 || i > private$.length) {
      stop("Index out of bounds.")
    }
    private$.getNodeAt(i)$value
  },
  find = function(x) {
    node <- private$.sentinel$nxt
    for (i in seq_len(private$.length)) {
      if (identical(x, node$value)) {
        return(i)
      }
      node <- node$nxt
    }
    return(0)
  },
  length = function() {
    private$.length
  },
  asList = function() {
    node <- private$.sentinel$nxt
    result <- vector("list", private$.length)
    for (i in seq_len(private$.length)) {
```

```
      # The following is the canonical way to fill a list with values that
      # could, potentially, be NULL.
      # Doing result[[i]] <- ... would remove item `i` if `...` is NULL.
      result[i] <- list(node$value)
      node <- node$nxt
    }
    result
  },
  reverse = function() {
    result <- LinkedList$new()
    node <- private$.sentinel$prv
    for (i in seq_len(private$.length)) {
      result$append(node$value)
      node <- node$prv
    }
    result
  },
  subset = function(i, j) {
    checkmate::assertCount(i, tol = 0)
    checkmate::assertCount(j, tol = 0)
    if (i > j) {
      stop("`i` must be less than or equal to `j`.")
    }
    if (i < 1 || j > private$.length) {
      stop("Index out of bounds.")
    }
    result <- LinkedList$new()
    node <- private$.getNodeAt(i)
    # as civilized people, we really dislike writing things like `i:j`, but
    # here we have in fact verified before that `i <= j`, so we will allow it.
    for (k in i:j) {
      result$append(node$value)
      node <- node$nxt
    }
    result
  }
),
private = list(
  # we could have a £.head and £.tail pointer, but these mean we have to handle
  # many special cases when the list is empty. Instead we use an empty 'sentinel'
  # node, which is never removed from the list, and which points to the first
  # element with £nxt, and the last element with £prv.
  #
  # Since it is an R6 object, we have to set it in `initialize()`, not here!
  .sentinel = NULL,
  .length = 0,

  # Get the Node object at index `i`
  .getNodeAt = function(i) {
    checkmate::assertIntegerish(i, lower = 1, upper = private$.length, tol = 0)
    node <- private$.sentinel$nxt
    for (j in seq_len(i - 1)) {
      node <- node$nxt
    }
    node
  },

  # Insert `node.new` in place of `node.place`; `node.place` and all subsequent
```

```
    # nodes are shifted to the right.
    .insertInPlaceOf = function(node.new, node.place) {
      node.new$prv <- node.place$prv
      node.new$nxt <- node.place
      node.place$prv$nxt <- node.new
      node.place$prv <- node.new
      private$.length <- private$.length + 1
    },

    # Remove `node` from the list; all nodes after `node` are shifted to the left.
    .remove = function(node) {
      if (private$.length == 0) {
        stop("The list is empty.")
      }
      node$prv$nxt <- node$nxt
      node$nxt$prv <- node$prv
      private$.length <- private$.length - 1
    }
  )
)
```