Exercise 1: Dynamic Array

The *amortized cost* is the average time per operation, evaluated over a sequence of operations, when individual operations may have varying costs. There are different ways to derive the amortized cost. Using *aggregate analysis*, the amortized cost is the average cost per operation over a sequence of operations. Given a sequence of n operations with total cost $T(n)$, the amortized cost per operation is $T(n)/n$. It is usually expressed using Landau notation, so e.g. $\Theta(f(n))$.

We are going to make use of the following operations, with their respective costs:

- $A \leftarrow$ **allocate**(n): Create an empty Array of size n. Allocation costs $\Theta(1)$. The array size n can then be queried with the **length**(A) function (which itself is $\Theta(1)$).
- **copy**(A, B): Copy the contents of Array A to Array B. Cost: $\Theta(\text{length}(A))$.
- $A[i] \leftarrow x$: Set the value of A at the ith position to x. Cost: $\Theta(1)$.
- $A \leftarrow B$: Replace the array that is referenced by A with the array that is referenced by B, and deallocate the original array. This operation does not involve any moving of data, it basically just renames the array. The cost is therefore $\Theta(1)$.

Consider the following algorithms for a dynamically growing array, i.e. an array that provides an `append` method that can add an unlimited number of elements. For each of these, determine the costs of the `append` operation as $\Theta(f(n))$ in terms of the current number of elements n in the array. You should find the asymptotic *best-case* and *worst-case* cost of adding an element to an array of size n, as well as the asymptotic amortized cost of adding elements up to size n.

(a)

Algorithm 1 GrowingArray1

```
A ← ALLOCATE(0),  n ← 0
function APPEND(x)
    n ← n + 1
    A′ ← ALLOCATE(n)
    COPY(A, A′)
    A ← A′
    A[n] ← x
end function
```

(b)

Algorithm 2 GrowingArray2

```
A ← ALLOCATE(100),  n ← 0
function APPEND(x)
    n ← n + 1
    if n > LENGTH(A) then
        A′ ← ALLOCATE(n + 100)
        COPY(A, A′)
        A ← A′
    end if
    A[n] ← x
end function
```

Algorithm 3 GrowingArray3

(c)

$A \leftarrow \text{ALLOCATE}(1),\ n \leftarrow 0$
function APPEND(x)
    $n \leftarrow n + 1$
    if $n > \text{LENGTH}(A)$ **then**
        $l \leftarrow \text{LENGTH}(A)$
        $A' \leftarrow \text{ALLOCATE}(2 \cdot l)$
        $\text{COPY}(A, A')$
        $A \leftarrow A'$
    end if
    $A[n] \leftarrow x$
end function

Algorithm 4 GrowingArray4

(d)

$A \leftarrow \text{ALLOCATE}(100),\ n \leftarrow 0$
function APPEND(x)
    $n \leftarrow n + 1$
    if $n > \text{LENGTH}(A)$ **then**
        $l \leftarrow \text{LENGTH}(A)$
        $A' \leftarrow \text{ALLOCATE}(\lceil 1.1 \cdot l \rceil)$
        $\text{COPY}(A, A')$
        $A \leftarrow A'$
    end if
    $A[n] \leftarrow x$
end function

Here, $\lceil x \rceil$ denotes rounding up x to the next integer.

Exercise 2: Queue

Write an R6 class that implements a queue with a maximum capacity of 10 elements. The class should have the following methods:

- `enqueue(x)`: add an element (of any type) to the queue
- `dequeue()`: remove and return the oldest element from the queue
- `head()`: return the oldest element from the queue
- `tail()`: return the newest element from the queue
- `size()`: return the number of elements in the queue

`enqueue` should throw an error if the queue is full. `dequeue`, `head`, and `tail` should throw an error if the queue is empty. Internally, the class should use a list to store the elements. It should be efficient in terms of adding and removing elements from the queue, i.e. it should not re-allocate the list for every operation.