

Programming assignment 1: Arrays and Recursion

In order to simulate the use of pure arrays in Python we will apply *strict limitations* to our use of the Python *list*. The following limitations apply to both the **base implementation** of ArrayList and **sorting and searching**.

In short there are only two things we **may do**:

- Initialize the *array* in this way:
 - **arr = [0] * size**
 - Where **size** can be any integer (also hard-coded, if needed; **arr = [0] * 16**)
 - The variable also *doesn't* have to be called **size**
- Access the value at one specific location in the array:
 - **arr[3] = 7**
 - **arr[index] = "some_string"**
 - **some_number = arr[i+1]**
 - **arr1[i] = arr2[j]**
 - **arr[i] = arr[i+1]**
 - **print(arr[index])**
 - It is fine to send the value of an item into built-in functions
 - Just not the list itself

Many things are **not allowed**:

- Calling a built-in function on the list class
 - **lis.append("some_string")**
 - **lis.insert(i, 19)**
- Sending the list directly into a built-in python function
 - **len(lis)**
 - **print(lis)**
 - **str(lis)**
- Using ranges or negative integers in the bracket operator
 - **lis[1:]**
 - **lis[0:10]**
 - **lis[-1]**
 - **Lis[1:-1]**
- Using operators directly on the list
 - **lis3 = lis1 + lis2**
 - **lis += [3,4]**
 - **lis += "some_string"**
 - **lis *= 2**
 - although this is good for a quick-fix **resize** implementation
 - it is not "legal" in a final implementation
 - **lis2 = lis1 * 2**
- Using the *join* functionality in any way

Base implementation (60%)

Make a class called ArrayList that encapsulates an array. Implement the following functions in that class (these will be tested with integers, strings and custom classes):

- **`__str__(self)`**
 - Returns a string with all items from the array
 - Have a comma and a space between them
 - but no brackets ([]) around them
- **`prepend(self, value)`**
 - Inserts an item into the list before the first item
- **`insert(self, value, index)`**
 - Inserts an item into the list at a specific location, ***not overwriting*** other items
 - *If the index is not within the current list, **raise `IndexOutOfBoundsException()`***
 - *It should be possible to add to the front and back of the list, and anywhere in between*
- **`append(self, value)`**
 - Adds an item to the list after the last item
- **`set_at(self, value, index)`**
 - Sets the value at a specific location to a specific value
 - Overwrites the current value there
 - *If the index is not within the current list, **raise `IndexOutOfBoundsException()`***
- **`get_first(self)`**
 - Returns the first value in the list
 - *If there are no items in the list, **raise `Empty()`***
- **`get_at(self, index)`**
 - Returns the value at a specific location in the list
 - *If the index is not within the current list, **raise `IndexOutOfBoundsException()`***
- **`get_last(self)`**
 - Returns the last value in the list
 - *If there are no items in the list, **raise `Empty()`***
- **`resize(self)`**
 - Re-allocates memory for a larger array and populates it with the original array's items
- **`remove_at(self, index)`**
 - Removes from the list an item at a specific location
 - *If the index is not within the current list, **raise `IndexOutOfBoundsException()`***
- **`clear(self)`**
 - Removes all items from the list
- Test these operations well. You can implement a random number insertion, which generates random numbers and then calls the functions several times.
 - Test **edge cases** specifically
 - Insert into an *empty* list, or outside possible indices
 - Insert at the very *end* (or *exactly one* too far)
 - Remove from *empty* list
 - Add in all possible ways to a list that is *exactly full* (*size == capacity*)
 - *Add, remove and clear* often and unpredictably.
- *Bonus 5% on top of grade for solutions without any unnecessary repetition of code.*

Sorting and searching (20%)

Add the following functionality to your class (this will only be tested with integer values).

- **ArrayList instance knows if it is ordered or not**
 - When you have only ever inserted in an ordered fashion, it is ordered
 - You can only insert in an ordered fashion if it's already ordered
 - When you add to the list in any other way it will not be ordered anymore
- **insert_ordered(self, value)**
 - Insert a value so that the list retains ordering
 - If the ArrayList instance is not in an ordered state, **raise NotOrdered()**
- **find(self, value)**
 - Returns the index of a specific value
 - If the instance of ArrayList is in an ordered state, use recursive binary search
 - If the ArrayList instance is not ordered, use linear search
 - *If the value is not found in the list, ~~raise NotFound()~~*
- **remove_value(self, value)**
 - Removes from the list an item with a specific value
 - *Can you use only helper functions that have already been implemented?*
 - *If the value is not found in the list, raise NotFound()*
- **Bonus 5% on top of grade if all find operations are implemented recursively and without unnecessary copying of data**

In all of the implementations, students are free to add any helper functions, classes or instance variables or default variables that they deem helpful or necessary.

Recursion (20%)

This assignment is not directly related to the ArrayList assignment.

It should be implemented using recursive programming and restrictions on the use of lists do not apply.

modulus(a, b)

- Write the recursive operation **modulus** that calculates the modulus of two integers without using the mathematical operators *, / or %
 - e.g.
 - modulus(13, 4) == 1
 - modulus(12, 3) == 0
 - modulus(14, 3) == 2

how_many(lis1, lis2)

- Write the recursive operation **how_many** that takes two lists and returns an integer the value of which is how many of the items in **lis1** are also in **lis2**.
 - e.g.
 - how_many([a,f,d,t], [a,b,c,d,e]) == 2
 - If two items in **lis1** have the same value, they are each counted
 - E.g.
 - how_many([a,b,f,g,a,t,c], [a,b,c,d,e]) == 4