* This lab will review dynamic arrays, run-time analysis of list methods, and sorting.
* It is assumed that you have reviewed chapter 5 and chapter 12 of the textbook. You may want to refer to the text and your lecture notes during the lab as you solve the problems.
* When approaching the problems, think before you code. Doing so is good practice and can help you lay out possible solutions.
* Think of any possible test cases that can potentially cause your solution to fail!
* Your TAs are available to answer questions in the lab, and during office hours.

**Part A**

Dynamic Arrays and List Methods

(2:00 PM - 4:10 PM)

**Vitamins (20 minutes)**

1. Give the **worst case** (not amortized) asymptotic run-time for each of the following list methods. Write your answer in terms of n, the length of the list. Provide an appropriate summation for multiple calls. (10 minutes)

Given: lst = [ 1, 2, 3, 4, … ,n], len(lst) is n, and val is an integer.

What will be the run-time when calling the following for lst?

| Method | Single (1) Call | Multiple (n) Calls  for i in range(n): ... |
| --- | --- | --- |
| append(val) |  | What will be the total cost if lst = [ ] instead?  Will the overall run-time change? |
| Insert(0, val) |  |  |
| pop(0) |  |  |

1. Given the following mystery functions: (10 minutes)
2. Replace mystery with an appropriate name
3. Determine the function’s worst-case runtime and extra space usage with respect to the input size.
4. **def** mystery(n):

lst = []

for i in range(n):

lst.insert(i, i)

1. **def** mystery(n):

for i in range(1, n+1):

total = sum([num for num in range(i)])

print(total)

1. **def** mystery(lst):

lst2 = lst.copy()

lst2.reverse()

if (lst == lst2):

return True

return False

**Coding (70 mins)**

In this section, it is strongly recommended that you solve the problem on paper before writing code.

Download the **Lab3\_ArrayList.py** file found under Resources/Lectures on Brightspace

**1.** Extend the ArrayList class implemented during lecture with the following methods:

* 1. Implement the\_\_iadd\_\_ method for the ArrayList class. (10 minutes)

The expression arr1 += arr2 **mutates** arr1 to contain the concatenation of these two lists. You may remember that this operation produces the same result as the **extend method**.

**Your implementation should return *self*, which is the object being mutated**

ex) arr1 is an ArrayList with [1, 2, 3]

arr2 is an ArrayList with [4, 5, 6]

→ arr1 += arr2

arr1 is mutated and now has [1, 2, 3, 4, 5, 6].

Note: If *n1* is the size of arr1, and *n2* is the size of arr2, then \_\_iadd\_\_ should run in . It’s not *n2* because we have to take array resizing into account.

* 1. Modify the \_\_getitem\_\_ and \_\_setitem\_\_ methods to also support **negative** indices. (10 minutes)

The position a negative index refers to is the same as in the Python list class. That is -1 is the index of the last element, -2 is the index of the second last, and so on.

ex) arr is an ArrayList with [1, 2, 3]

→ print(arr[-1]) outputs 3

→ print(arr[-2]) outputs 2

→ arr[-1] = 5

print(arr[-1]) outputs 5 now

Note: Your method should also raise an IndexError in case the index (positive or negative) is out of range.

* 1. Implement the\_\_mul\_\_method for the ArrayList class. (10 minutes)

The expression arr1 \* k (where k is a positive integer) creates and returns a **new** ArrayList object, which contains k copies of the elements in arr1.

ex) arr1 is an ArrayList with [1, 2, 3]

→ arr2 = arr1 \* 2

arr2 is a new ArrayList with [1, 2, 3, 1, 2, 3].

Note: If *n* is the size of arr1 and k is the int, then \_\_mul\_\_ should run in .

* 1. Implement the\_\_rmul\_\_method to also allow the expression n \* arr1. The behavior of n \* arr1 should be equivalent to the behaviour of arr1 \* n. (5 minutes)

(You’ve done this before for the Vector problem in homework 1)

* 1. Modify the constructor \_\_init\_\_ to include an option to pass in an iterable collection such as a string and return an ArrayList object containing each element of the collection. Do not account for dictionaries. (10 minutes)

ex) arr = ArrayList("Python")

→ print(arr) outputs ['P','y','t',h','o','n']

→ arr2 = ArrayList(range(10))

→ print(arr2) outputs [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

* 1. Implement a remove method. (10 minutes)

remove(val) will remove the **first** instance of val in the ArrayList. It must be done in run-time. **You do not need to call resize.**

ex) arr is an ArrayList with [1, 2, 3, 2, 3, 4, 2, 2]

→ arr.remove(2)

→ print(arr2) outputs [1, 3, 2, 3, 4, 2, 2]

* 1. Implement a removeAll( ) method. (15 minutes)

removeAll(val) will remove all instances of val in the ArrayList. The implementation should be in-place and maintain the relative order of the other values. It must also be done in run-time. **You do not need to call resize.**

ex) arr is an ArrayList with [1, 2, 3, 2, 3, 4, 2, 2]

→ arr.removeAll(2)

→ print(arr2) outputs [1, 3, 3, 4]

**Part B**

Sorting

(4:20 PM - 5:50 PM)

**Vitamins (40 minutes)**

**1.** For each function, identify the sorting algorithm being used and analyze its run-time and extra space usage. (20 minutes)

Hint: You may want to draw your list after each iteration to see what’s happening.

Use a small list input such as [1, 3, 6, 5, 2, 4] to test each function.

Choices: Merge Sort, Insertion Sort, Selection Sort

1. def sort1(lst):

for i in range(1, len(lst)):

curr = lst[i]

j = i - 1

while j >= 0 and curr < lst[j]:

lst[j+1] = lst[j]

j -= 1

lst[j+1] = curr

1. def sort2(lst):

for i in range(len(lst)):

min\_ind = i

for j in range(i+1, len(lst)):

if lst[min\_ind] > lst[j]:

min\_ind = j

lst[i], lst[min\_ind] = lst[min\_ind], lst[i]

1. def sort3(lst):

if len(lst) > 1:

mid = len(lst)//2

l = lst[:mid]

r = lst[mid:]

sort3(l)

sort3(r)

i = j = k = 0

while i < len(l) and j < len(r):

if l[i] < r[j]:

lst[k] = l[i]

i += 1

else:

lst[k] = r[j]

j += 1

k += 1

while i < len(l):

lst[k] = l[i]

i += 1

k += 1

while j < len(r):

lst[k] = r[j]

j += 1

k += 1

**2.** Different sorting algorithms have different runtimes and extra space usage. Provide the appropriate value for the following cells. For the runtimes, use Big-O notation. (20 minutes)

Note: Average case is different from the amortized case. With amortization, we take the entire run-time cost into account but with the average case, we make an assumption on probability. That is, we assume that the worst-case list input does not happen often.

The best case runtime is the theoretical fastest runtime you can get from a function when the input is perfectly optimized. For sorting algorithms, this could be when the input is already sorted.

| **Sorting Algorithm** | **Worst Case Run-time** | **Average Case Run-time** | **Best Case Run-time** | **Extra Space** |
| --- | --- | --- | --- | --- |
| Insertion Sort |  |  |  |  |
| Selection Sort |  |  |  |  |
| Merge Sort |  |  |  |  |

**Coding (25 minutes)**

In this section, it is strongly recommended that you solve the problem on paper before writing code.

1. Implement a function, sort\_ints\_and\_floats(lst), that is given a list of integers and floats, lst. (25 minutes)

When called, this function should change the order of elements in lst so that all integers appear first, and all floats appear last. Moreover, all the integers should be sorted in an ascending order, and all the floats should be in ascending order. This function should be in-place.

DO NOT split the integers and floats first, and then sort them separately. Rather, you should sort your elements as you simultaneously split integers and floats.

Note: This function can be written in O(nlogn) worst case run-time. Although you are encouraged to write your function in O(nlogn), you are allowed to implement a function that runs in O(n^2) worst case run-time.

→ lst = [3, 4.1, 1, 4, -1, 1.5, 5.6, 10, 4]

→ sort\_ints\_and\_floats(lst)

→ print(lst) outputs [-1, 1, 3, 4, 4, 10, 1.5, 4.1, 5.6]