**Vitamins (45 minutes)**

1. For each of the following , write out the summation results, and provide a tight bound , using the (5 minutes).

Given numbers, where *n* is a power of *2*:

=

  
 a\_1 = first term = 1

n = number of terms = log\_2(n)

r = common ratio BETWEEN elements = 2

*=*

Provide a tight bound , using the

= ()

Recall this sum:  
 = (sum of ARITHMETIC sequence)

= ()

Similar to the first question in lab

1. For each of the following code snippets, find for which the algorithm’s time complexity is in its **worst case** run and explain why. (10 minutes)  
     
   1. def func(lst):

i = 1

n = len(lst)

while (i < n):

print(lst[i])

i \*= 2

= ← this summation

# Each addend in the sum is 1 since indexing a list is a constant-time operation.  
There are terms in the summation. This is because in the while loop we multiply by two in each iteration. So we will have iterations relative to the length of the list (if the length is 80, we will have 7 iterations - where is 1 in the 1st iteration, 2 in the 2nd iteration, then 4, then 8, then 16, then, 32, finally 64 )

* 1. def func(lst):

i = 1

n = len(lst)

while (i < n):

print(lst)

i \*= 2

=

#Each addend in the sum is n since printing a list is a linear-time operation. (Think about it - how would the list be printed without some work being done that traverses the whole length of the list)  
  
There are terms in the summation. This is because in the while loop we multiply by two in each iteration

* 1. def func(lst):

i = 1

n = len(lst)

while (i < n): # i =n -1

print(lst[ : i])

i \*= 2

=

#Each addend represents the cost of slicing the list. If the list is of length 80, lst[:16] would represent 16 in the summation.   
  
There are terms in the summation. This is because in the while loop we multiply by two in each iteration

1. Give the **worst case** (not amortized) run-time O(n) 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. (25 minutes)

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

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

Two factors to consider: SHIFT + RESIZE. For append, SHIFT = 0

The actual runtime for insert is O(n - k), where k = insertion index, if append, k = n

| Method | 1 Call | Reason. |
| --- | --- | --- |
| append() | O(n)  O(1) amortized | Worst case, the array of n elements has reached maximum capacity. Therefore, we need to create a new array, copy n elements over, and then append the new element.  However, since the array is not always resized, we say its **amortized O(1)**. But **NOT worst-case** and **NOT the same as average theta(1).** |
| insert(0, val) | O(n) | Inserting to the front at index 0 will shift each of the n elements by 1 index over to the right. |

n^2

| Method | Multiple Calls | Reason |
| --- | --- | --- |
| append() | O(n) | 1. **If the list already has n elements:**   Worst case, we resize to create an array of capacity 2n and copy over n elements to the new array. Then, we add our n elements, so this is n + 1, which is O(n).   1. **If the list starts empty: (See folder for image reference)**   Worst case, n is a power of 2 (n = 2^k). So we add 1 element to the starting array of capacity 1, which costs 1.  Then we add our 2nd element, but we’d have to resize our array to size 2, so this results is 1 (copy over) + 1(new elem) = 2.  We add the 3rd and 4th elements before having to resize again, so that cost 2 but on the 5th element, we resize.  Thus, the cost of adding the 3rd and 4th elements is 2(new elem) + 2(copy over) = 4.  The cost of adding the 5th - 8th element will cause us to resize and copy over 8 elements when we are adding our 9th element, so this results in 4 (new elem) + 4(copy over) = 8.  Summation results in 1 + 2 + 4 + 8 … + n = 2n-1  O(n)  **NOTE: AMORTIZED COST for SINGLE call**  In both cases, since resizing does not happen on every append, the cost of each append varies. We can “redistribute” TOTAL costs 2n over n operations and say that each operation will cost 2n/n = 2 operations, which is O(1) constant. However, this is amortization and should not be used for the actual cost of n appends.  N appends do 1 + 1 + 1 + 1…  Does not make sense to prove the calculation of n appends using amortized cost, which was derived from the cost of n appends in the first place. |
| insert(0, val) | O(n^2) | Worst case, we insert elements from the front at index 0.   1. **If the list already has n elements:**   If we insert element 1 at 0, inserting is 1, and we shift n elements over, so cost is n + 1  If we insert element 2 at 0, inserting is 1 and we shift n + 1 elements over, so total is n + 2  Continuing this results in (n+1) + (n+2) + (n+3) … (n + (n))  We separate this into n(n) + (1 + 2 + 3 … n),  This results in n^2 + n(n+1)/2, which is O(n^2)   1. **If the list starts empty:**   If we insert element 1 at 0, we just insert, so cost is 1  If we insert element 2 at 0, we shift and insert, so cost is 1 + 1  Continuing this results in 1 + 2 + 3 … + n  This results in n(n+1)/2, which is O(n^2)  \* Array resizing cost is negligible (it just adds 2n-1 from resize). |

**Optional:** Derive the **amortized (worst case) cost** of a single append( ).

1. Given the following mystery functions: (5 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** appendN(n):

lst = []

for i in range(n):

lst.insert(i, i) #this is just append

1 + 2 + 4 + 8 + … + n = 2n-1 = O(n)

1. **def** triangle\_num(n):

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

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

print(total)

1 + 2 + 3 + 4 + 5 + 6 + … n = n(n+1)/2 = O(n^2)

Extra space is O(n) because a list is created each time for the sum but the lists do not exist simultaneously so it is not n^2.

1. **def** is\_palindrome(lst):

lst2 = lst.copy() #new list, O(n) Extra Space

lst2.reverse() #O(n)

if (lst == lst2): #O(n)

return True

return False

Extra space is another factor to consider for efficiency. We can solve is\_palindrome in O(n) run-time in addition to O(1) constant space as demonstrated in lab4. It is important to be conscious of run-time and extra space (do not create another list if the problem can be solved in-place).

**Coding (See CS1134 Lab5 Solutions.py)**

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

Download the **ArrayList.py** file found under Resources/Lectures on NYU Classes

Extend the ArrayList class implemented during lecture with the following methods:

* 1. Implement the \_\_repr\_\_ method for the ArrayList class, which will allow us to display our ArrayList object like the Python list when calling the print function. The output is a sequence of elements enclosed in [ ] with each element separated by a space and a comma. (10 minutes)

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

→ print(arr) outputs [1, 2, 3]

Note: Your implementation should create the string in , where n = len(arr).

* 1. Implement the\_\_add\_\_ method for the ArrayList class, so that the expression   
     arr1 + arr2 is evaluated to a **new** ArrayList object representing the concatenation of these two lists. (10 minutes)

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

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

→ arr3 = arr1 + arr2

arr3 is a new ArrayList with [1, 2, 3, 4, 5, 6].

Note: If *n1* is the size of arr1, and *n2* is the size of arr2, then \_\_add\_\_ should run in

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

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 implemented in class to also support **negative** indices. 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. (20 minutes)

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

→ print(arr[-1]) outputs 3

→ 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, so that the expression arr1 \* k (where k is a positive integer) creates a **new** ArrayList object, which contains k copies of the elements in arr1. (15 minutes)

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. (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 that will remove the first instance of val in the ArrayList.

Set None in place of the val that is removed. **You do not have to account for resizing the array for this question.** (20 minutes)

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 that 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. Set None in place of the val that is removed. **You do not have to account for resizing the array for this question.**

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

→ arr.removeAll(2)

→ print(arr2) outputs [1, 3, 3, 4]

2.

In class, you learned that finding an element in a **sorted list** can be done in run-time with *binary search*.

Suppose that, we take a **sorted list and shift it by some random k steps**:

ex) lst = [1, 3, 6, 7, 10, 12, 14, 15, 20, 21] → [15, 20, 21, 1, 3, 6, 7, 10, 12, 14]

*Food for thought:* Now, if we try to use binary search to search for 21, index: left = 0, right = 9, mid = 4, we see that lst[left] = 15, lst[right] = 14, lst[mid] = 3. How will you know which side to discard? How many sorted parts do you see in the list?

You may define additional functions to solve the problem. You may also use the binary search implemented in class, which can be found in NYU resources. (35 minutes)

**Part A: Find the pivot – *Time Constraint O(log(n))***

Given a rotated sorted list, find the index of the smallest value (aka the pivot point).

Input: nums = [15, 20, 21, 1, 3, 6, 7, 10, 12, 14]

Output: 3

Explanation: At index 3, the minimum value is 1, the pivot point.

def find\_pivot(lst):  
 """  
 : lst type: list[int] #sorted and then shifted

: val type: int  
 : return type: int (index if found), None(if not found)  
"""

**Part B: Find the target value – *Time Constraint O(log(n))***

Given a rotated sorted list and a target value, find the index of the target value.

**Hint:** You may use the find\_pivot() function defined in Part A to get the pivot and process

which side of the list to search for.

Input: nums = [15, 20, 21, 1, 3, 6, 7, 10, 12, 14], target = 21

Output: 2

Explanation: The target value 21 is found in the list at index 2.

def shift\_binary\_search(lst, target):  
"""  
: lst type: list[int] #sorted and then shifted

: target type: int  
: return type: int (index if found), None(if not found)  
"""