- This lab will cover dynamic arrays (list) and run-time analysis of list methods.
- It is assumed that you have reviewed chapter 5 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!
- You do not have to stay for the duration of the lab if you finish early. If you finish early, you may help other students. If you don't finish by the end of the lab, we recommend you complete it on your own time. Ideally, you should not spend more time than suggested for each problem.
- Your TAs are available to answer questions in the lab, during office hours, and on Piazza.

Vitamins (45 minutes)

---

1. For each of the following $f(n)$, write out the summation results, and provide a tight bound $\Theta(f(n))$, using the Θ *notation* (5 minutes).

   Given $log(n)$ numbers, where *n* is a power of *2*:

   $1 + 2 + 4 + 8 + 16 \ldots + n =$ ____2n - 1____ = $\Theta($ ____n____ $)$

   $n + n/2 + n/4 + n/8 \ldots + 1 =$ ____2n - 1____ = $\Theta($ ____n____ $)$

   Provide a tight bound $\Theta(f(n))$, using the Θ *notation*

   $1 + 2 + 3 + 4 + 5 \ldots + \sqrt{n} =$ ____(n + sqrt(n))/2____ = $\Theta($ ____n____ $)$

   $1 + 2 + 4 + 8 + 16 \ldots + \sqrt{n} =$ ____2sqrt(n)) - 1____ = $\Theta($ ____sqrt(n)____ $)$

2. For each of the following code snippets, find $f(n)$for which the algorithm's time complexity is $\Theta(f(n))$in its **worst case** run and explain why. (10 minutes)

   a.
   ```
   def func(lst):
       i = 1
       n = len(lst)
       while (i < n):
           print(lst[i])
           i *= 2
   ```

   log(n) because no matter the n, the loop will run log(n) times

b.
```
def func(lst):
    i = 1
    n = len(lst)
    while (i < n):
        print(lst)
        i *= 2
```

this will run in n log(n) time, because the loop runs log(n) times but each time the print statement takes n to run

c.
```
def func(lst):
    i = 1
    n = len(lst)
    while (i < n):
        print(lst[ : i])
        i *= 2
```

n log(n) because the print statement will run (2n-1) times and the function will run log(n) times

3. Give the **worst case** run-time for each of the following list methods. Write your answer in asymptotic notation 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 **worst-case** run-time when calling the following for lst?

Method	1 Call	n Calls `for i in range(n): ...`
`append()`	theta(n)	1+2+…n/4+n/2+n/2+n+n = theta(n) What will be the total cost if lst = [ ] instead? Will the overall run-time change?
`insert(0, val)`	theta(n)	1+2+1+1+5+1+1+1+9 = theta(n) n+(n+1)+(n+2)+(n+3)+…(n+n) = theta(n^2) What will be the total cost if lst = [ ] instead? Will the overall run-time change?

1+2+…+n/4+n/2+n = theta(n)

   Derive the **amortized cost** of a single call of append.

(1+2+…n/4+n/2+n/2+n+n)/n =(4n-2)/n = 4n-2/n = theta(1)

4. Given the following mystery functions: (5 minutes)
   - **i.** Replace `mystery` with an appropriate name (what does the function do?)
   - **ii.** Determine the function's **worst-case runtime** and **extra space usage** with respect to the input size.

a. counting_array

```
def mystery(n):
    lst = []
    for i in range(n):
        lst.insert(i, i)
```

this function runs theta(n) and only uses n amount of space

b. running_total

```
def mystery(n):
    for i in range(1, n+1):
        total = sum([num for num in range(i)])
        print(total)
```

this function will run in theta(n^2) time because the loop will run n times and the total function amoritized will run n times

1+2+3+n = theta(n)

c. palindrome_checker

```
def mystery(lst):
    lst2 = lst.copy()      n
    lst2.reverse()         n
    if (lst == lst2):      n
        return True
    return False
```

this will run in 3n = theta(n) worst case

Coding

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

Download the **ArrayList.py** file found under /Content/Labs on NYU Brightspace

1. Extend the ArrayList class implemented during lecture with the following methods (note: for each of these methods, simulate the same behaviors as those of the built-in python list):

a. 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 $\Theta(n)$, where n = len(arr).

b. 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) (*think of this as a shallow concatenation of the lists*)

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 n_1 is the size of arr1, and n_2 is the size of arr2, then __add__ should run in $\Theta(n_1 + n_2)$

c. 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 n_1 is the size of arr1, and n_2 is the size of arr2, then __iadd__ should run in $\Theta(n_1 + n_2)$. It's not n_2 because we have to take array resizing into account.

d. 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.

e. 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 $\Theta(k * n)$.

f. Implement the `__rmul__` method to also allow the expression n * arr1. The behavior of n * arr1 should be equivalent to the behavior of arr1 * n. (5 minutes)

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

g. 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]`

h. Implement a remove( ) method that will remove the first instance of val in the ArrayList. **You do not have to account for physically 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]`

i. 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 $\Theta(n)$ **run-time**. **You do not have to account for physically 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 $\Theta(log(n))$ 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)
"""
```

----------**OPTIONAL**----------

3. For this question, you will write a new searching algorithm, *jump search*, that will search for a value in a **sorted list**. With jump search, you will separate your list into n/k groups of k elements each. It then finds the group where the element you are looking for should be in, and makes a linear search in this group only.

For example, let's say k = 4 for the following list of n = 20 elements:

[1, 3, 6, 7 , 10, 12, 15, 20 , 22, 24, 29, 33 , 39, 55, 61, 64 , 99, 101, 134, 150]

Here, we have a list of n/k = 20/4 = 5 groups. If we were to check for val = 15, we would start with the first element (index 0) of the first group and check if we've found our value.

Since, $1 != 15 \ and \ 1 < 15$, we will jump k = 4 elements and move to 10 (at index 4).

[1, 3, 6, 7 , 10, 12, 15, 20 , 22, 24, 29, 33 , 39, 55, 61, 64 , 99, 101, 134, 150]

Since $10 != 15 \ and \ 10 < 15$, we jump another k = 4 steps and move to 22 (at index 8).

[1, 3, 6, 7 , 10, 12, 15, 20 , 22, 24, 29, 33 , 39, 55, 61, 64 , 99, 101, 134, 150]

Now, we have $22 != 15 \ and \ 22 > 15$, so we don't need to jump further. Instead, we will hop back k elements because we know our val has to be somewhere between 10 and 22, (index 4 and index 8). We jump back up to k = 4 elements until we find our val = 15.

[1, 3, 6, 7 , 10, 12, 15, 20 , 22, 24, 29, 33 , 39, 55, 61, 64 , 99, 101, 134, 150]
← ← ←

With the sample list above, we jump from 1 to 10, and 10 to 22. Then we linear search back between 22 and 10 and found 15.

Recap:

1. Divide the list into `n/k` groups of `k` elements.
2. Jump `k` steps each time until either `val == lst[i]` or `val < lst[i]`.
3. if `i + k > len(lst)-1`, jump to index len(lst) - 1 instead.
4. if `val == lst[i]`, return the index i.
5. if `val < lst[i]`, jump back one step each time for a maximum of `k` steps.
6. If `val` is somewhere in those `k` steps, return the index.
7. Otherwise, val not in list so return `None`.

3a. (35 minutes)

Write a function that performs jump search on a sorted list, given an additional parameter, k. This parameter will let the user divide the list into k groups for the search.

Consider some edge cases such as if n isn't divisible by k (there will be a group with fewer than k elements) or if val > all elements in the list? Assume 0 < k < len(lst).

Analyze the worst case run-time of your function in terms of n, the size of the list, and k :

```
def jump_search(lst, val, k):
"""
: lst type: list[int]
: val, k type: int
: return type: int (index if found), None(if not found)
"""
```

3b. (5 minutes)

Let's now optimize our jump search algorithm by defining what our k value should always be. That is, we will no longer have k be passed in as a parameter.

Hint: The jump search algorithm is actually slower than binary search but faster than linear search. You may use functions from the math library for this algorithm.

Analyze the run-time of jump search in terms of n, the length of the list:

```
def jump_search(lst, val):
"""
: lst type: list[int]
: val type: int
: return type: int (index if found), None(if not found)
"""
```

Here is a simple test code to verify that your searching algorithm works.

```
#TEST CODE

lst = [-1111, -818, -646, -50, -25, -3, 0, 1, 2, 11, 33, 45,
46, 51, 58, 72, 74, 75, 99, 110, 120, 121, 345, 400, 500, 999,
1000, 1114, 1134, 4444, 10010, 500000, 999999]

#Testing k

for i in range(1, len(lst)):
    # print("i:",i)
    print("TESTING VALUES IN LIST:", "k = ", i, "\n")

    for val in lst:
        if jump_search_k(lst, val, i) is None:
            print(val, "FAILED - DID NOT FIND")

#just to make sure you’re not stuck in an infinite loop
print("TEST k COMPLETED")



#Testing sqrt

print("\nTESTING VALUES IN LIST: k = sqrt(n) \n")

for val in lst:
    if jump_search(lst, val) is None:
        print(val, "FAILED - DID NOT FIND")


#just to make sure you’re not stuck in an infinite loop
print("TEST sqrt COMPLETED")
```

4. In class, you learned about *binary search*, which has a run-time of $O(log(n))$for searching through a **sorted list**. With binary search, the lower bound begins at index 0 while the upper bound is the last index, len(list) - 1.

You will write a modification of the binary search called *exponential search* (also called doubling or galloping search). With exponential search, we start at index i = 1 (after checking index 0) and we double i until it becomes the upper bound.

Let's try to find `15` in the given sample list by checking index `i = 0` first:

`i = 0, (lst[0] = 1) != (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

From there, you will make a comparison to see if the value is at index, i = 1. If not, you will double i until the index is out of range or until the value at the index is larger than the value you're searching for. **It is important for the two conditions to be in that order!**

`i = 1, (lst[i] = 3) < (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

`i = 2, (lst[i] = 6) < (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

`i = 4, (lst[i] = 10) < (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

`i = 8, (lst[i] = 22) > (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

If the value is less than i or is out of bound, then you know it has to be between `i//2` and `i` (or `i//2` and `len(list)-1`). After confirming your bounds, you perform a binary search (in that range only).

`(lst[4] = 10) < val < (lst[8] = 22):`

[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

With the sample list above, we confirm that val must be between i = 4 and i = 8.

Recap:

1. Start with `i = 0` and check if `val == lst[0]` (if list is not empty).
2. If `val != lst[0]`, start the exponential search (doubling process) at `i = 1`.
3. If `val != lst[1]`, check if `val > lst[i]`. If so, double `i`.
4. Keep doubling while `i*2 < len(lst) and lst[i] < val`.
5. Set left bound to `i//2` and right bound to `i`
6. If `lst[right] < val`, then upper bound `right = len(lst)-1`
7. Perform binary search between the left and right bounds.

What is the run-time for *exponential search* and what advantage might this algorithm have over binary search? (35 minutes)

```
def exponential_search(lst, val):
"""
: lst type: list[int]
: val type: int
: return type: int(if found), None(if not found)

"""

      if len(lst) > 0:           #check if list is not empty
            if lst[0] == val:    #check index 0 first
                  return 0
            else:
                  i = 1 #start at 1 for the exponential search
                  ...
      return None
```

Here is a simple test code to verify that your searching algorithm works.

```
#TEST CODE

lst = [-1111, -818, -646, -50, -25, -3, 0, 1, 2, 11, 33, 45, 46, 51,
58, 72, 74, 75, 99, 110, 120, 121, 345, 400, 500, 999, 1000, 1114,
1134, 10010, 500000, 999999]


#Testing exponential

print("\nTESTING VALUES IN LIST: exponential \n")

for val in lst:
     if exponential_search(lst, val) is None:
          print(val, "FAILED - DID NOT FIND")


#just to make sure you’re not stuck in an infinite loop
print("TEST COMPLETED")
```