- This lab will cover <u>dynamic arrays (list) and run-time analysis of list methods.</u>
- 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, <u>think before you code</u>. Doing so is good practice and can help you lay out possible solutions.
- <u>Think of any possible test cases</u> 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. <u>Ideally, you should not spend more time than suggested for each problem.</u>
- Your TAs are available to answer questions in the lab, during office hours, and on Edstem.

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 = $ _____ $= \Theta($ _____ $)$

   $n + n/2 + n/4 + n/8 \ldots + 1 = $ _____ $= \Theta($ _____ $)$

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

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

   $1 + 2 + 4 + 8 + 16 \ldots + \sqrt{n} = $ _____ $= \Theta($ _____ $)$

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
   ```

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


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

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()		What will be the total cost if lst = [ ] instead? Will the overall run-time change?
insert(0, val)		What will be the total cost if lst = [ ] instead? Will the overall run-time change?

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

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

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

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

,

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**.

   <u>Your implementation should return *self*, which is the object being mutated.</u> (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]`.

   <u>Note</u>: 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

   <u>Note</u>: 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]`.

    <u>Note</u>: 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 removeOdds( ) method that will remove all odd values 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, 5]`

   →   `arr.removeOdds()`
   →   `print(arr2)` outputs `[2, 2, 4, 2]`

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, $1 \mathrel{!=} 15 \; and \; 1 \; < \; 15$, 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 $10 \mathrel{!=} 15 \; and \; 10 \; < \; 15$, 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 $22 \mathrel{!=} 15 \; and \; 22 \; > \; 15$, 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")
```