- -This lab will cover more asymptotic analysis, problem solving, and searching.
- It is assumed that you have reviewed chapter 3 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 must stay for the duration of the lab. 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 (35 minutes)

For **big-O proof**, show that there exists constants c, and n_0 such that $f(n) \leq c*g(n)$ for every $n \geq n_0$, then $f(n) = O(g(n))$.

For **big-Θproof**, show that there exists constants c1, c2, and n_0 such that $c1*g(n) \leq f(n) \leq c2*g(n)$ for every $n \geq n_0$, then $f(n) = \Theta(g(n))$.

For **big-Ω proof**, show that there exists constants c, and n_0 such that $f(n) \geq c*g(n)$ for every $n \geq n_0$, then $f(n) = O(g(n))$.

1. State **True** or **False** for the following (5 minutes):

   a) $n^3 = O(nlog(n^n))$

   b) $\sqrt{n} = O(log(n))$

   c) $\sqrt{n} = O(\frac{n}{log(n)})$

   d) $n! = O(100^n)$

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

   Given n numbers:

   $1 + 1 + 1 + 1 + 1 \ldots + 1$= _____ = Θ(_____)

   $n + n + n + n + n \ldots + n =$ _____ = Θ(_____)

   $1 + 2 + 3 + 4 + 5 \ldots + n =$ _____ = Θ(_____)

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

   $1 + 2 + 3 + 4 \ldots + log(n) =$ _____ = Θ(_____)

3. 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. (15 minutes)

a)
```
def func(lst):
    for i in range(len(lst)):
        for j in range(i):
            print(lst[j], end = " ")
```

b)
```
def func(lst):
    for i in range(0, len(lst), 2):
        for j in range(i):
            print(lst[j], end = " ")
```

c)
```
def func(n):
    for i in range(n):
        j = 1
        while j <= 80:
            print("i = ",i,", j =", j)
            j *= 2
```

d)
```
def func(n):
    for i in range(n):
        j = 1
        while j <= n:
            print("i = ",i,", j =", j)
            j *= 2
```

Coding

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

1. Given a string, write a function that returns True, if it is a palindrome, and False, if not. A string is a palindrome if its characters are read the same backwards and forwards.

   For example, `"1racecar1"` is a palindrome but `"1racecar2"` is not. You are not allowed to create a new list or use any str methods. Your solution must run in $\Theta(n)$, where n is the length of the input string. (5 minutes)

   ```
   PATTERN: TWO POINTERS MOVING INWARD -> <-

   def is_palindrome(s):
       """
       : s type: str
       : return type: bool
       """
   ```

2. Write a function that takes in a string as input and returns a new string with its vowels reversed. For example, an input of “tandon” would return “tondan”. Your function must run in $\Theta(n)$ and you may assume all strings will only contain lowercase characters. Two lines of code have already been given to you.
   (10 minutes)

   ```
   PATTERN: TWO POINTERS MOVING INWARD -> <-
   ```

   You should use:

   1. The **list constructor** converts the string into a list in linear time.
   2. The **.join() string method**, which is guaranteed to run in linear time.
      The join() method is a string method that can take in a list of string values and returns a string concatenation of the list elements joined by a str separator.
      For example: `",".join(["a", "b", "c"])` will return `"a,b,c"`.
   3. low and high variables as pointers to traverse through the list.

   ```
   def reverse_vowels(input_str):
   """
   : input_str type: string
   : return type: string
   """
   list_str = list(input_str) #list constructor guarantees
   Theta(n)
   # Your code implementation goes here
   return "".join(list_str)
   ```

3. Complete the following (35 minutes):

   a. The function below takes in a **sorted** list with n numbers, all taken from the range 0 to n, with one of the numbers removed. Also, none of the numbers in the list is repeated. The function searches through the list and returns the missing number. This list contains no duplicate integers.

      For instance, lst = [0, 1, 2, 3, 4, 5, 6, 8] is a list of 8 numbers, from the range 0 to 8, with the number 7 missing. Therefore, the function below will return 7.

      Analyze the worst case run-time of the function:

```
def find_missing(lst):
    for num in range(len(lst) + 1):
        if num not in lst:
            return num
```

   b. Rewrite the function so that it finds the missing number with a better run-time: **Hint:** the list is sorted – this algorithm should be in log(n). Also, make sure to consider the edge cases.

```
def find_missing(lst):
    """
    : nums type: list[int] (sorted)
    : return type: int
    """
```

   c. Suppose the given list is **not sorted** but still contains all the numbers from 0 to n with one missing.

      For instance, lst = [8, 6, 0, 4, 3, 5, 1, 2] is a list of numbers from the range 0 to 8, with the number 7 missing. Therefore, the function below will return 7.

      How would you solve this problem? Do not use the idea in step a, or sort the list and reuse your solution in step b.

```
def find_missing(lst):
    """
    : nums type: list[int] (unsorted)
    : return type: int
    """
```

4. 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)
"""
```

5. **OPTIONAL**

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`.

5a. (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)
"""
```

5b. (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")
```

6.

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")
```