- This lab will focus on asymptotic analysis, problem solving, and binary search.
- It is assumed that you have reviewed chapters 1 and 2 of the textbook. You may want to refer to the text and your lecture notes during 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.
- Your TAs are available to answer questions in lab, during office hours, and on Piazza

---

Vitamins (maximum 1 hour)

---

1. Use the **definitions of** O **and** Θ in order to show the following:

a) $n^2 + 5n - 2$ is $O(n^2)$

b) $\frac{n^2+1}{n+1}$ is $O(n)$

c) $\sqrt{5n^2 - 3n + 2}$ is $\Theta(n)$

2. State **True** or **False** and explain why for the following:

d) $8n^2(\sqrt{n})$ is $O(n^3)$

e) $8n^2(\sqrt{n})$ is $\Theta(n^3)$

f) $16\ log(n^2) + 2$ is $O(log(n))$

3. Sort the following 17 functions in an increasing asymptotic order and write `<, <=,` between each two subsequent functions to indicate if the first is asymptotically less than, asymptotically greater than or asymptotically equivalent to the second function respectively.

For example, if you were to sort: $f_1(n) = n$, $f_2(n) = log(n)$, $f_3(n) = 3n$, $f_4(n) = n^2$, your answer could be $log(n) < (n <= 3n) < n^2$

$f_1(n) = n$
$f_2(n) = 500n$
$f_3(n) = \sqrt{n}$
$f_4(n) = log(\sqrt{n})$
$f_5(n) = \sqrt{log(n)}$
$f_6(n) = 1$
$f_7(n) = 3^n$
$f_8(n) = n \cdot log(n)$
$f_9(n) = \frac{n}{log(n)}$
$f_{10}(n) = 700$
$f_{11}(n) = log(n)$
$f_{12}(n) = \sqrt{9n}$
$f_{13}(n) = 2^n$
$f_{14}(n) = n^2$
$f_{15}(n) = n^3$
$f_{16}(n) = \frac{n}{3}$
$f_{17}(n) = \sqrt{n^3}$

Coding

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

1. Given a list of values (`int, float, str, …`), write a function that reverse its order in-place. You are not allowed to create a new list. Your solution must run in $\Theta(n)$, where n is the length of the list.

```
def reverse_list(lst):
    """
    : lst type: list[]
    : return type: None
    """
```

2. Given a string, write a function that returns True, if it is a palindrome, and False, if not. A string is a palindrome if the ordering of the characters from left to right is the same as that read from right to left.

   For example, `"1racercar1"` 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.

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

3. Given a **sorted** list of positive integers with zeros mixed in, write a function to move all zeros to the end of the list while maintaining the order of the non-zero numbers. For example, given the list [0, 1, 0, 3, 13, 0], the function will modify the list to become [1, 3, 13, 0, 0, 0]. Your solution must be in-place and run in $\Theta(n)$, where n is the length of the list.

```
def move_zeroes(nums):
    """
    : nums type: list[int]
    : return type: None
    """
```

4.

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.

   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. 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
      """
```

Optional

In this question we will measure the actual running time of three different algorithms that solve the **Max Contiguous Subsequence Sum** problem. *Max Contiguous Subsequence Sum* is a well-known problem in computer science. See for more info: https://en.wikipedia.org/wiki/Maximum_subarray_problem

In this problem we are supposed to find a contiguous subsequence within a list of numbers which has the largest sum.

For example, for the list [−2, 1, −3, 4, −1, 2, 1, −5, 4], the contiguous subsequence with the largest sum is [4, −1, 2, 1], with sum 6. **You do not have to solve the problem.**

Instead, you will compare the time it takes to solve the Max Contiguous Subsequence Sum problem using the three algorithms provided in the resources folder on NYU Classes. The file, MaxSubsequenceSumAndTimer.py, contains the three algorithms and a `PolyTimer` class. You will run all three algorithms with the following values of n: $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, $2^{10} = 1024$, $2^{11} = 2048$, $2^{12} = 4096$.

What you will need to do:

a) Look at the code for the `PolyTimer` class in the file and then apply it in your own code.

b) Fill a list with n random integers in the range from -1000 to 1000. Recall that there is a random library in Python.

c) Time how long it takes the function `maxSubsequenceSum1` to find the maximum subsequence sum.

To give you an idea, the code *might* look like the following:

```
t = PolyTimer()
nuClicks = 0.0
# your code to fill in the list with n items, etc.
t.reset()
result, start, end = maxSubsequenceSum1(lst)
nuClicks = t.elapsed()
```

d) Time how long it takes the function `maxSubsequenceSum2` to find the maximum subsequence sum, using the same n elements.

e) Time how long it takes the function `maxSubsequenceSum3` to find the maximum subsequence sum, again using the same n elements.

f) Export your results to a CSV file and make three charts in Excel that correspond to each function. To do so…

   i) Create a Scatter Plot for each function using the runtime as the Y axis and the number of elements as the X axis.

   ii) Right-click on the chart that generated and click “Add trendline”

   iii) In the trendline properties, choose “Power”

   iv) Check off “Display equation on chart”

Note that executing your code should not take you more than 20 minutes on a low-spec machine (on an older ThinkPad it took *way* less). Make sure when you print the running times you are printing enough significant digits, preferably to the 6th place after the decimal point.