- This lab will cover Asymptotic Analysis.
- 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 lab as you solve the problems.
- When approaching the problems, think before you code. It is good practice and generally helpful to lay out possible solutions for yourself.
- You should write test code to try out your solutions.
- You must stay for the duration of the lab. If you finish early, you can help other students to complete the lab. If you don't finish by the end of the lab, it is recommended that you complete the lab on your own time.
- Your TAs are available to answer your questions in lab, during office hours, and on Piazza.

---

**Vitamins (maximum 45 minutes)**

---

1. Use the **definitions of** $O$ **and** $\Theta$ in order to show the following:

a) $\frac{n^2+1}{n+1}$ is $O(n)$
b) $2n^2 - 4n + 7$ is $\Theta(n^2)$

2. Sort the following 17 functions in an increasing asymptotic order and write '<', '>' or '=' 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 $log(n) < 3n = n < n^2$

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

$f_{13}(n) = 2^n$
$f_{14}(n) = n^2$
$f_{15}(n) = n^3$
$f_{16}(n) = \frac{n}{3}$
$f_{17}(n) = \sqrt{n^3}$

3. Given the following functions, what is the output of the code:

a.

```
def factorial(num):
        value = 1
        for i in range(1, num + 1):
            value *= i
            yield value

for val in factorial(5):
    print(val)
```

b.

```
def letters(word):
   for i in range(len(word)):
       yield word[i]

for l in letters("computer"):
    print(l)
```

## Coding

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

1. The function `roll_the_dice_str(n)` is given a positive integer *n*, and returns a string containing *n* dice rolls results, separated by a space (each roll is 1-6). For example, the call `roll_the_dice_str(10)` could return `'4 4 5 3 3 5 6 5 1 3'` .

    a. Consider the following implementation of `roll_the_dice_str(n)`:

```
import random
def roll_the_dice_str(n):
      s = ""
      for i in range(1, n+1):
            curr_val = randint(1, 6)
            s = s + str(curr_val) + " "
      return s[:-1]
```

    Analyze the worst-case running time of the call `roll_the_dice_str(n)`.

    Note: In your analysis assume:
    1) The `+` operator for strings creates a **new** string instance containing the concatenated value, and **runs in a linear time**, proportional to the length of the new string that was created.
    2) The run time of `randint()` is $\Theta(1)$

    b. The `join(iterable)` method is a string method that returns a string which is the concatenation of the strings in the iterable. A `TypeError` is raised if there are any non-string values in the iterable. The separator between elements is the string providing (calling) this method.

    For example, the call `';'.join(['abc', 'def', '123'])` would return `'abc;def;123'`

    We don't know how the join method is implemented, however it is guaranteed that its running time is linear. That is, proportional to the length of the string it returns.

    Give a **linear time** implementation for `roll_the_dice_str(n)`.
    Analyze the running time of your implementation.

2. 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 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
      # other code to fill in the list with n items, etc.
      t.reset()
      result, start, end = maxSubsequenceSum1(someList)
      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.

3. Given a sorted list of positive integers with zeros placed in between some of them, 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], the function will modify the list to become [1, 3, 13, 0, 0]. Note that this must be done in-place and you are only allowed $\Theta(1)$ additional space. Your solution must run in $\Theta(n)$.

```
def move_zeroes(nums):
"""
: nums type: list[int]
: return type: void
"""
```