# Homework #2

## Due by Tuesday 7/21, 11:55 pm

Submission instructions:

1. For this assignment, you should turn in 5 files:
    - 5 '.py' files, one for each question 1-5. Name your files: YourNetID_hw1_q2.py' and 'YourNetID_hw1_q3.py', etc.
      Note: your netID follows an abc123 pattern, not N12345678.

2. You should submit your homework via Gradescope.
    - Name all classes, functions, and methods exactly as they are in the assignment specifications.
    - Make sure there are no print statements in your code. If you have a tester code, please put it in a "main" function and do not call it.

**Question 1**:
You are given 2 implementations for a recursive algorithm that calculates the sum of all the elements in a list (of integers):

```
def sum_lst1(lst):
    if (len(lst) == 1):
        return lst[0]
    else:
        rest = sum_lst1(lst[1:])
        sum = lst[0] + rest
        return sum

def sum_lst2(lst, low, high):
    if (low == high):
        return lst[low]
    else:
        rest = sum_lst2(lst, low + 1, high)
        sum = lst[low] + rest
        return sum
```

Note: The implementations differ in the parameters we pass to these functions:

- In the first version we pass only the list (all the elements in the list have to be taken into account for the result).
- In the second version, in addition to the list, we pass two indices: be considered.
  The initial values (for the first call) passed to low and high would represent the range of the entire list.

1. Make sure you understand the recursive idea of each implementation.
2. Analyze the running time of the implementations above. For each version:
   a. Draw the recursion tree that represents the execution process of the function, and the local-cost of each call.
   b. Conclude the total (asymptotic) running time of the function.
3. Which version is asymptotically faster?

**Question 2**:

Give a **recursive** implement to the following function:

```
def list_min(lst, low, high)
```

The function is given `lst`, a list of integers, and two indices: `low` and `high` (`low ≤ high`, which indicates the range of indices that need to be considered.
The function should find and return the minimum value out of all the elements at the position *low, low+1, ..., high* in `lst`.

**Question 3:**

Give a **recursive** implement to the following functions:

a. `def count_lowercase(s, low, high):`

   The function is given a string `s`, and two indices: `low` and `high` (`low ≤ high`), which indicate the range of indices that need to be considered.
   The function should return the number of lowercase letters at the positions *low, low+1, ..., high* in `s`.

b. `def is_number_of_lowercase_even(s, low, high):`

   The function is given a string `s`, and two indices: `low` and `high` (`low ≤ high`), which indicates the range of indices that need to be considered.
   The function should return `True` if there are even number of lowercase letters at the positions *low, low+1, ..., high* in `s`, or `False` otherwise.

**Question 4:**

Give a **recursive** implement to the following function:

`def split_by_sign(lst, low, high)`

The function is given a list `lst` of non-zero integers, and two indices: `low` and `high` (`low ≤ high`), which indicate the range of indices that need to be considered.
The function should reorder the elements in `lst`, so that all the negative numbers would come before all the positive numbers.
Note: The order in which the negative elements are at the end, and the order in which the positive are at the end, doesn't matter, as long as all the negative are before all the positive.

**Question 5:**

A ***nested list of integers*** is a list that stores integers in some hierarchy. That is, its elements are integers and/or other nested lists of integers.

For example `nested_lst=[[1, 2], 3, [4, [5, 6, [7], 8]]]` is a nested list of integers.

Give a **recursive** implement to the following function:

```
def flat_list(nested_lst, low, high)
```

The function is given a nested list of integers `nested_list`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should flatten the sub-list at the positions *low, low+1, ..., high* of `nested_list`, and return this flattened list. That is, the function should create a new 1-level (non hierarchical) list that contains all the integers from the *low...high* range in the input list.

For example, when calling `flat_list` to flatten the list `nested_lst` demonstrated above (the initial call passes `low=0` and `high=2`), it should create and return `[1, 2, 3, 4, 5, 6, 7, 8]`.