CS-119(h) Final: Solutions for the Programming Questions

Question 8

```
def addme(x, y):
    global cnt
    cnt += 1

    res.add(x + y)

    if x > 0 and y > 0:
        addme(x - 2, y)
        addme(x, y - 1)

cnt = 0
res = set()
addme(4, 2)
print(cnt, len(res))
```

☐ `7 3` ☐ `11 11` ☐ `5 3` ■ `11 6` ☐ `3 3`

Solution for Question 8

The code defines a recursive function, `addme`, and calls this function with `(4, 2)` as input. The answer to this question is the final value of the variable `cnt` and the length of the set `res`. Note that:

- `cnt` is a global counter that increases by one every time the function `addme` is called. It is initialized to 0.
- `res` is a global set that stores the sum of `x` and `y` for each function call. It is initialized to an empty set.
- The recursion stops when either `x` or `y` is less than or equal to 0.
- If both `x` and `y` are greater than 0, the function makes two recursive calls, namely `addme(x - 2, y)` and `addme(x, y - 1)`.

Step-by-Step Execution: The following trace shows the order of execution, including when each call starts, finishes, and returns to the previous call. Each step is labeled with the current values of `x`, `y`, `cnt`, and `res`.

1. **Call 1:** `addme(4, 2)`
   - `cnt` increments to 1.
   - `res` inserts `4 + 2 = 6` to the set and becomes `res = `{`6`}.
   - Because `x = 4 > 0` and `y = 2 > 0`, two recursive calls are made:
     - `addme(4 - 2, 2) = addme(2, 2)` (Call 2)
     - `addme(4, 2 - 1) = addme(4, 1)` (Call 3)
2. **Call 2:** `addme(2, 2)`
   - `cnt` increments to 2.
   - `res` inserts `2 + 2 = 4` to the set and becomes `res = `{`4, 6`}.
   - Because `x = 2 > 0` and `y = 2 > 0`, two recursive calls are made:

  - `addme(2 - 2, 2) = addme(0, 2)` (Call 4)
  - `addme(2, 2 - 1) = addme(2, 1)` (Call 5)

3. **Call 4:** `addme(0, 2)`
   - `cnt` increments to 3.
   - `res` inserts `0 + 2 = 2` to the set and becomes `res = {2, 4, 6}`.
   - Because `x = 0` (not greater than 0), no further recursive calls are made.
   - **Call 4 finishes and returns to Call 2.**
4. **Call 5:** `addme(2, 1)`
   - `cnt` increments to 4.
   - `res` inserts `2 + 1 = 3` to the set and becomes `res = {2, 3, 4, 6}`.
   - Because `x = 2 > 0` and `y = 1 > 0`, two recursive calls are made:
     - `addme(2 - 2, 1) = addme(0, 1)` (Call 6)
     - `addme(2, 1 - 1) = addme(2, 0)` (Call 7)
5. **Call 6:** `addme(0, 1)`
   - `cnt` increments to 5.
   - `res` inserts `0 + 1 = 1` to the set and becomes `res = {1, 2, 3, 4, 6}`.
   - Because `x = 0` (not greater than 0), no further recursive calls are made.
   - **Call 6 finishes and returns to Call 5.**
6. **Call 7:** `addme(2, 0)`
   - `cnt` increments to 6.
   - `res` inserts `2 + 0 = 2` to the set, but 2 already exists in the set, `res = {1, 2, 3, 4, 6}`.
   - Because `y = 0` (not greater than 0), no further recursive calls are made.
   - **Call 7 finishes and returns to Call 5.**
7. **Call 5 finishes and returns to Call 2.**
8. **Call 2 finishes and returns to Call 1.**
9. **Call 3:** `addme(4, 1)`
   - `cnt` increments to 7.
   - `res` inserts `4 + 1 = 5` to the set and becomes `res = {1, 2, 3, 4, 5, 6}`.
   - Because `x = 4 > 0` and `y = 1 > 0`, two recursive calls are made:
     - `addme(4 - 2, 1) = addme(2, 1)` (Call 8)
     - `addme(4, 1 - 1) = addme(4, 0)` (Call 9)
10. **Call 8:** `addme(2, 1)`
    - `cnt` increments to 8.
    - `res` inserts `2 + 1 = 3` to the set, but 3 already exists in the set, `res = {1, 2, 3, 4, 5, 6}`.
    - Because `x = 2 > 0` and `y = 1 > 0`, two recursive calls are made:
      - `addme(2 - 2, 1) = addme(0, 1)` (Call 10)
      - `addme(2, 1 - 1) = addme(2, 0)` (Call 11)
11. **Call 10:** `addme(0, 1)`
    - `cnt` increments to 9.
    - `res` inserts `0 + 1 = 1` to the set, but 1 already exists, `res = {1, 2, 3, 4, 5, 6}`.
    - Because `x = 0` (not greater than 0), no further recursive calls are made.
    - **Call 10 finishes and returns to Call 8.**
12. **Call 11:** `addme(2, 0)`

- `cnt` increments to 10.
- `res` inserts `2 + 0 = 2` to the set, but 2 already exists, `res =` {`1, 2, 3, 4, 5, 6`}.
- Because `y = 0` (not greater than 0), no further recursive calls are made.
- **Call 11 finishes and returns to Call 8.**

13. **Call 8 finishes and returns to Call 3.**
14. **Call 9:** `addme(4, 0)`
    - `cnt` increments to 11.
    - `res` inserts `4 + 0 = 4` to the set, but 4 already exists, `res =` {`1, 2, 3, 4, 5, 6`}.
    - Because `y = 0` (not greater than 0), no further recursive calls are made.
    - **Call 9 finishes and returns to Call 3.**
15. **Call 3 finishes and returns to Call 1.**
16. **Call 1 finishes.**
17. The program outputs the value of the `cnt`, `11`, and the length of the set `res`, `6`.

Hence, the final answer is `11 6`.

Question 9

```
def foo(a):
    b = 100
    c = a + b or a - b
    a, b, c = b, c, a
    return a, b, c

a = 10
b = a + 10
c = b + 10

x, y, z = foo(b)
print(y + z, a + b)
```

- ☐ `120 30`
- ☐ `220 110`
- ☐ `140 220`
- ☐ `40 30`
- ■ `140 30`
- ☐ `21 30`

Solution for Question 9

The code defines a function, `foo(a)`, and performs the following steps:

- Initialization of the following variables: `a = 10`, `b = a + 10 = 20`, and `c = b + 10 = 30`.
- The function `foo` is called with `20` as input. Inside the function:
  - `b = 100`. Here, the variable shadowing is used; any modification on this `b` variable does not affect the global variable `b`.
  - `c = a + b or a - b = 120`. Note that because `a + b = 120` is `True`, the rest of the expression is not evaluated.
  - The values are swapped: `a, b, c = 100, 120, 20`.
  - The function returns `(100, 120, 20)`.
- Unpacking the return values: `x, y, z = 100, 120, 20`
- Printing the final result: `y + z = 120 + 20 = 140` and `a + b = 10 + 20 = 30`.

Hence, the final answer is `140 30`.

Question 10

```
i = 1
new_dict = dict.fromkeys(range(6), i)
elements = list(new_dict.keys())

for elem in elements:
    i += 1
    for e in elements[::i]:
        new_dict[elem] -= e

new_dict.popitem()
new_dict[new_dict.pop(2)] = i

print(len(new_dict), set(new_dict.values()))
```

- [ ] 6 {1}
- [x] 5 {1, 7, -5, -4, -2}
- [ ] 6 {1, 7, -5, -4, -2}
- [ ] 5 {1, -5, -4, -3, -2}
- [ ] 6 {1, 7, -5, -4, -3, -2}
- [ ] 6 {1, -5, -4, -3, -2}

Solution for Question 10

The code does the following operations:

- `i = 1`: Initializes the variable `i`.
- `new_dict = dict.fromkeys(range(6), i)`: Creates a dictionary with keys from 0 to 5 (inclusive), all initialized to `i = 1`. Therefore:
  `new_dict = {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1}`
- `elements = list(new_dict.keys())`: Stores the list of keys in `elements`:

  `elements = [0, 1, 2, 3, 4, 5]`

- `i += 1`: Increments `i` by 1 in each iteration.
  - `for elem in elements`: Iterates over each item in `elements`.
  - `for e in elements[::i]`: Loops over a slice of `elements` with step `i`.
- `new_dict[elem] -= e`: Decrements the value of `new_dict[elem]` by the current value of `e`.
- There are updates on `new_dict` during each iteration of `elem`, as follows:
  - **Iteration 1:** `elem = 0, i = 2`
    Slice: `elements[::2]` = `[0, 2, 4]`
    Updates:

    `new_dict[0] = 1 - 0 - 2 - 4 = -5`

    Resulting dictionary:

    `new_dict = {0: -5, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1}`

  - **Iteration 2:** `elem = 1, i = 3`
    Slice: `elements[::3]` = `[0, 3]`
    Updates:

    `new_dict[1] = 1 - 0 - 3 = -2`

    Resulting dictionary:

    `new_dict = {0: -5, 1: -2, 2: 1, 3: 1, 4: 1, 5: 1}`

- **Iteration 3:** `elem = 2, i = 4`
  Slice: `elements[::4]` = `[0, 4]`
  Updates:

  `new_dict[2] = 1 - 0 - 4 = -3`

  Resulting dictionary:

  `new_dict = {0: -5, 1: -2, 2: -3, 3: 1, 4: 1, 5: 1}`

- **Iteration 4:** `elem = 3, i = 5`
  Slice: `elements[::5]` = `[0, 5]`
  Updates:

  `new_dict[3] = 1 - 0 - 5 = -4`

  Resulting dictionary:

  `new_dict = {0: -5, 1: -2, 2: -3, 3: -4, 4: 1, 5: 1}`

- **Iteration 5:** `elem = 4, i = 6`
  Slice: `elements[::6]` = `[0]`
  Updates:

  `new_dict[4] = 1 - 0 = 1`

  Resulting dictionary:

  `new_dict = {0: -5, 1: -2, 2: -3, 3: -4, 4: 1, 5: 1}`

- **Iteration 6:** `elem = 5, i = 7`
  Slice: `elements[::7]` = `[0]`
  Updates:

  `new_dict[5] = 1 - 0 = 1`

  Resulting dictionary:

  `new_dict = {0: -5, 1: -2, 2: -3, 3: -4, 4: 1, 5: 1}`

- `new_dict.popitem()`: Removes the item that was last inserted into the dictionary.

  `new_dict = {0: -5, 1: -2, 2: -3, 3: -4, 4: 1}`

- `new_dict[new_dict.pop(2)] = i`: Removes the key `2`, whose value is `-3`, and assigns `i = 7` to a new key `-3`:

  `new_dict = {0: -5, 1: -2, 3: -4, 4: 1, -3: 7}`

- `print(len(new_dict), set(new_dict.values()))`: Outputs the length of `new_dict` and the unique set of its values: `(5, {-5, -4, -2, 1, 7})` which is equivalent with `(5, {1, 7, -5, -4, -2})`

Hence, the final answer is `(5, {1, 7, -5, -4, -2})`.

Question 11

```
lst = []
for i in range(6):
    for j in range(i):
            lst.append(i)
s = set(lst)

s = (s & {4, 5, 6, 7}) - {6, 7}
s = (s | {8}) ^ {4, 9}

print(s)
```

☐ {1, 2, 3, 5, 8, 9} ■ {8, 5, 9} ☐ {5, 6, 7, 8, 9}

☐ {9, 4} ☐ {5, 8, 6, 7}

Solution for Question 11

The code constructs the set `s` from the list `lst`. Initially, this list is empty, but the nested loops append the value of `i` a total of `i` times for each value of `i` from 0 to 5 (inclusive). Hence, the list contains only the numbers from 0 to 5, and therefore `s = {0, 1, 2, 3, 4, 5}`. Afterward, the code executes the following operations:

- Intersection with {4, 5, 6, 7} and removal of {6, 7}:

$$\texttt{s} = (\texttt{s} \cap \{\texttt{4, 5, 6, 7}\}) - \{\texttt{6, 7}\} = \{\texttt{4, 5}\}$$

- Union with {8} and XOR (symmetric difference) with {4, 9}:

$$\texttt{s} = (\{\texttt{4, 5}\} \cup \{8\}) \oplus \{\texttt{4, 9}\} = \{\texttt{5, 8, 9}\}$$

Hence, the final output is {5, 8, 9}, or {8, 5, 9} as given in the question.

Question 12

```
def bar(n, r=1):
    if n == 0:
        return r
    elif n % 2:
        return bar(n - 1, r + n)
    else:
        return bar(n - 1, r + 2 * n)


print(bar(10))
```

☐ 56 ☐ 81 ☐ 79 ☐ 85 ■ 86

Solution for Question 12

The function is called with `10` as input. Let's evaluate the function's execution step by step:

1. `bar(n = 10, r = 1)`: Because `n` is even, call `bar(9, r + 2 * 10)` = `bar(9, 21)` is made.
2. `bar(n = 9, r = 21)`: Because `n` is odd, call `bar(8, r + 9)` = `bar(8, 30)` is made.
3. `bar(n = 8, r = 30)`: Because `n` is even, call `bar(7, r + 2 * 8)` = `bar(7, 46)` is made.
4. `bar(n = 7, r = 46)`: Because `n` is odd, call `bar(6, r + 7)` = `bar(6, 53)` is made.
5. `bar(n = 6, r = 53)`: Because `n` is even, call `bar(5, r + 2 * 6)` = `bar(5, 65)` is made.
6. `bar(n = 5, r = 65)`: Because `n` is odd, call `bar(4, r + 5)` = `bar(4, 70)` is made.
7. `bar(n = 4, r = 70)`: Because `n` is even, call `bar(3, r + 2 * 4)` = `bar(3, 78)` is made.
8. `bar(n = 3, r = 78)`: Because `n` is odd, call `bar(2, r + 3)` = `bar(2, 81)` is made.
9. `bar(n = 2, r = 81)`: Because `n` is even, call `bar(1, r + 2 * 2)` = `bar(1, 85)` is made.
10. `bar(n = 1, r = 85)`: Because `n` is odd, call `bar(0, r + 1)` = `bar(0, 86)` is made.
11. `bar(n = 0, r = 86)`: Because `n` is 0, returns 86.

Hence, the final result of the recursive function is `86`.

Question 13

```
def matrix_manipulation(v_in, matrix_in, matrix_out):
    global n
    for i in range(n):
        for j in range(n):
            if i == 2:
                continue
            if j == 1:
                break
            matrix_out[j][0] += matrix_in[j][i] * v_in[i]

n = 3
vector = [1, 2, 3]
matrix_a = [[1, 2, 3], [1, 3, 2], [2, 1, 3]]
matrix_b = [[0] for i in range(3)]

matrix_manipulation(vector, matrix_a, matrix_b)
print(matrix_b)
```

- [ ] `[[3], [0], [0]]`
- [ ] `[[14], [0], [0]]`
- [ ] `[[1], [4], [0]]`
- [x] `[[5], [0], [0]]`
- [ ] `[[5], [7], [4]]`

Solution for Question 13

The function `matrix_manipulation` performs operations on the input vector and matrices. The global variable `n = 3`, and the inputs are initialized as:

$$\texttt{vector} = [1, 2, 3], \quad \texttt{matrix\_a} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \end{bmatrix}, \quad \texttt{matrix\_b} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The function iterates over `i` and `j` from 0 to 2. Inside the loops:

1. If `i = 2`, the `continue` statement skips the rest of the inner loop.
2. If `j = 1`, the `break` statement exits the inner loop.

The key operation is:

$$\texttt{matrix\_out}[j][0] \mathrel{+}= \texttt{matrix\_in}[j][i] * \texttt{v\_in}[i]$$

1. **First Iteration (`i = 0`):**
   - `j = 0`: Update `matrix_out`[0][0]:

$$\texttt{matrix\_out}[0][0] \mathrel{+}= \texttt{matrix\_in}[0][0] * \texttt{v\_in}[0] = 0 + 1 * 1 = 1$$

   - `j = 1`: The `break` statement exits the inner loop.
2. **Second Iteration (`i = 1`):**
   - `j = 0`: Update `matrix_out`[0][0]:

$$\texttt{matrix\_out}[0][0] \mathrel{+}= \texttt{matrix\_in}[0][1] * \texttt{v\_in}[1] = 1 + 2 * 2 = 5$$

   - `j = 1`: The `break` statement exits the inner loop.
3. **Third Iteration (`i = 2`):**
   - The `continue` statement skips the inner loop.

After all iterations, the final state of `matrix_out` is:

$$\texttt{matrix\_out} = \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}$$

Hence, the program outputs the `matrix_b = [[5], [0], [0]]`.

Question 17

Problem Statement

Write a Python function `aggregate_score()` that takes as input a list of strings. Each string in the list contains a student's name followed by a space and a score representing the points obtained. Your function should aggregate the scores corresponding to the same student and return a dictionary containing the aggregate score for each student.

Input

A list of strings, where each string is formatted as "name score", where `name` is the student's name and `score` is a non-negative integer. You can assume that names are single words without spaces and that each name starts with a capital letter. For example, given the list `["Tom 5", "Lea 10", "Tom 7", "Lea 15"]`, the output should be `{"Tom": 12, "Lea": 25}`.

Output

A dictionary where each key is a unique name from the input list, and the corresponding value is the sum of all scores associated with that name.

Solution 1

This solution iterates through the list of scores, splits each entry into the student's name and score, and updates the dictionary accordingly.

```
def aggregate_score(scores):
    score_dict = {}
    for entry in scores:
        # Split each entry into student and score
        words = entry.split()
        student = words[0]
        # Convert the score to an integer
        score = int(words[1])
        # If the name is already in the dictionary, add the score; otherwise, initialize it
        if student in score_dict:
            score_dict[student] += score
        else:
            score_dict[student] = score
    return score_dict

scores = ["Tom 5", "Lea 10", "Tom 7", "Lea 15"]
score_dict = aggregate_score(scores)
print(score_dict)
# Output: {'Tom': 12, 'Lea': 25}
```

- The function `aggregate_score` initializes an empty dictionary `score_dict`.
- For each entry in the input list, the entry is split into the student's name and score.
- The score is converted to an integer.
- If the student's name is already in the dictionary, the score is added to the existing value. Otherwise, a new entry is created with the student's name as the key and the score as the value.
- The function returns the dictionary containing the aggregated scores.

Solution 2

This solution uses the `dict.get()` method to simplify the process of updating the dictionary.

```
def aggregate_score(scores):
    score_dict = {}
    for entry in scores:
        words = entry.strip().split()
        score_dict[words[0]] = score_dict.get(words[0], 0) + int(words[1])
    return score_dict

print(aggregate_score(["Tom 5", "Lea 10", "Tom 7", "Lea 15"]))
# Output: {"Tom": 12, "Lea": 25}
```

- The function `aggregate_score` initializes an empty dictionary `score_dict`.
- For each entry in the input list, the entry is stripped of leading/trailing whitespace and split into the student's name and score.
- The `dict.get()` method is used to retrieve the current score for the student, defaulting to 0 if the student is not already in the dictionary. The new score is then added to this value.
- The function returns the dictionary containing the aggregated scores.

Question 18

Problem Statement

You are given two lists, `list1` and `list2`, both of length n, and a window of size w. Your task is to write a **recursive** Python function `window_average_recursive()` to compute a new list where each element represents the average of the values from the two input lists within a sliding window of size w.

Steps for the Solution

1. For each position i of the sliding window ($0 \leq i \leq n - w$):
   - Consider the sub-sections:
   $$\text{window1} = \text{list1}[i], \text{list1}[i+1], \ldots, \text{list1}[i+w-1]$$
   and
   $$\text{window2} = \text{list2}[i], \text{list2}[i+1], \ldots, \text{list2}[i+w-1].$$
   - Compute the element-wise averages:
   $$\text{element\_average}[j] = \frac{\text{window1}[j] + \text{window2}[j]}{2}, \quad j = 0, 1, \ldots, w-1.$$
   - Compute the global average of these values within the window:
   $$\text{window\_average} = \frac{1}{w} \sum_{j=0}^{w-1} \text{element\_average}[j].$$
   - Add `window_average` to the result list.
2. Repeat this process for all possible positions of the sliding window until $i = n - w$.

Output

The resulting list will have a length of $n - w + 1$. The program should return a list containing all the window averages.

Example

```
list1 = [1, 2, 3, 4, 5]
list2 = [0, 2, 1, 3, 8]
w = 3
result = window_average_recursive(list1, list2, w)
print(result)
# Output: [1.5, 2.5, 4.0]
```

Note

If your implementation is entirely correct but not in the form of a recursive function, the maximum number of points you can obtain for this question is 5.

Recursive Solution 1

This solution uses a helper function with additional parameters to track the current position of the sliding window and accumulate the results.

```
def window_average_recursive(list1, list2, w, start=0, result=None):
    """
    Computes the window averages of two lists recursively.

    Parameters:
    - list1: First list of numbers (length n)
    - list2: Second list of numbers (length n)
    - w: Window size
    - start: Current start index of the sliding window (default is 0)
    - result: Accumulator list for the result (default is None)

    Returns:
    - A list of length n-w+1 containing the window averages.
    """
    # Initialize the result list on the first call
    if result is None:
        result = []

    # Base case: Stop recursion when the window exceeds the list bounds
    if start > len(list1) - w:
        return result

    # Compute the window averages for the current window
    window_sum = 0
    for i in range(w):
        window_sum += (list1[start + i] + list2[start + i]) / 2
    window_avg = window_sum / w
    result.append(window_avg)

    # Recursive case: Move the window by incrementing start
    return window_average_recursive(list1, list2, w, start + 1, result)
```

- The function `window_average_recursive` initializes the result list if it is not provided.
- The base case stops the recursion when the window exceeds the bounds of the lists.
- For each window, the function calculates the sum of the element-wise averages and then computes the global average for the window.
- The result is appended to the accumulator list, and the function calls itself with the window shifted by one position.

Recursive Solution 2

This solution uses list slicing to move the window and builds the result list by prepending the current window average to the result of the recursive call.

```
def window_average_recursive(list1, list2, w):
    if len(list1) < w:
        return list()
    window_list = []
    for i in range(w):
        window_list.append((list1[i] + list2[i]) / 2)
    window_avg = sum(window_list) / w
    result = window_average_recursive(list1[1:], list2[1:], w)
    result = [window_avg] + result
    return result
```

- The function `window_average_recursive` checks if the remaining list is shorter than the window size and returns an empty list if true.
- For each window, the function calculates the element-wise averages and computes the global average for the window.
- The result is built by prepending the current window average to the result of the recursive call on the rest of the lists.

Iterative Solution

This solution uses a loop to iterate through each position of the sliding window and computes the window averages without recursion.

```
def window_average_non_recursive(list1, list2, w):
    n = len(list1)
    result = []

    # Loop through each position of the sliding window (i from 0 to n-w)
    for i in range(n - w + 1):
        # Initialize sum for each window
        sum_averages = 0

        # Calculate the sum of averages for each element in the window
        for j in range(w):
            sum_averages += (list1[i + j] + list2[i + j]) / 2

        # Add the global average of the window to the result list
        result.append(sum_averages / w)

    return result
```

- The function `window_average_non_recursive` initializes an empty result list.
- It loops through each position of the sliding window and calculates the sum of the element-wise averages for each window.
- The global average for each window is computed and appended to the result list.