CHAPTER

VII

Week 4 - Assignment 2C

Warming up exercises

In this week's assignment we will continue to work on **recursive problems**. Use the following warming up exercises to practice. Those exercises were done during the practice session. You can find their solutions in Brightspace.

In this chapter:

Exercise 15.
Given two strings word and sentence, write a recursive function that receives both of them and counts how many times the str in word appears in the str sentence as a standalone word. The word should be counted even if it appears in a different case (ex: lower case vs camel case).

Tip! If you check if `word` matches the beginning of `sentence`, you can check all the possible positions of the sentence by progressively removing the first letter in sentence.

Asking the following questions can help you identify the structure of a **recursive function:**

- What is the base case?
- What argument is passed to the recursive function call?
- How does this argument become closer to the base case?

Exercise 16.
Modify the function above so that sentence is a list of strings. Then, use the head and tail technique to count the words.

Remark. Use of Generative AI:
Remember that using generative AI to solve the exercises is not allowed. If you make use of tools like chatGPT for personal use, please avoid doing that during the workgroups. If you have questions, the TAs are there to you understanding the content.

Recursive Problems 2

Download the template `recursion_problems_2.py` in Brightspace and implement the following recursive functions. See below the description of each function.

- `move_max_rec(array: np.ndarray[int])`:
  Imagine that we want to sort an array of `n` integers in ascending order. One possible way of doing that is through the following algorithm:
  - **Step 1:** loop over all the elements of the array and comparing adjacent elements. Whenever one element is greater than the next adjacent one, you can swap the two elements.
  - **Step 2:** Repeat step one multiple times (note that you can progressively exclude the last element(s) of the array, since they are already in the right place).

  For instance, after Step 1 for the first time, you end up with an array that has its maximum element at the last position. This means that if you repeat this procedure again (with or without the last element), you end up with an array with the last two elements sorted. Repeating that until all the elements are sorted is actually a sorting algorithm called **Bubble sort**!

  Write a recursive function `move_max_rec(array: np.ndarray[int])` that implements the **Step 1** of the algorithm discussed above. Your function should receive an array, and return the array with the max element at the end. Note that some of the other elements might change order as well.

  Example. `move_max_rec(np.array([1, 5, 3, 8, 4, 4]))` should return a modified version of the array with the maximum element at the end:

  ```
  >>> move_max_rec(np.array([1, 5, 3, 8, 4, 4]))
  np.array([1, 3, 5, 4, 4, 8])
  ```

  Tip! You can use `np.concatenate()` to join two arrays back together the head and tail before returning the function.

Bubble sort: you can see a visual demonstration of the bubble sort algorithm using Hungarian folk dance in the video bellow.

- `is_substring(big_string: str, small_string: str)`: a function that receives two strings, *big_string* and *small_string*, and returns `True` if `small_string` is a substring of `big_string` and `False` otherwise.

  Example.

  ```
  >>> is_substring("The treasure is hidden", "sure is")
  True
  >>> is_substring("The treasure is found", "Treasure")
  False
  ```

np.concatenate(): this function can concatenate two arrays. Read the documentation of this function if needed.

Use head-tail recursion!

- `find_combinations(input: list, target: int)`:
  Imagine you have a list of integers `input`. There are multiple ways to add those numbers so that they sum up to a specific `target`.

  For example, if input contains the values: 1, 2, 3, 4 and 5; and the target is 6, you can reach 6 in three different ways:

  $$1 + 2 + 3 = 6$$
  $$1 + 5 = 6$$
  $$2 + 4 = 6$$

  Write a function `find_combinations(input: list, target: int)` that given a list of numbers `input` and a `target`, returns how many different ways are there to reach the target by summing a set of the numbers in the list. Each number should be considered only once at each summation.

  > **Example.**
  > ```
  > >>> find_combinations([3, 2, 4, 1, 5], 6)
  > 3
  > >>> find_combinations([3, 2], 3)
  > 1
  > ```

  > **Tip!** You can use the head-tail technique to separate one element at the time, and either include them or not in the sum.
  > Remember that you can have more than one recursive call in a recursive function.

- *sort_array(arr : np.ndarray[int])*:
  Beyond Bubble Sort, there are multiple ways to sort the elements in an array/list. In this problem, we will use recursion and the head-tail technique to implement another method of sorting.

  Recursive functions have an initial part of the code that is done **before the call of the function**, and operations that are done **after the recursive call**. The operations that are done before the recursive call group together, being executed **from the outermost to the innermost call**. On the other hand, operations done after the recursive call group together, executed **from the innermost to the outermost call**.

  Considering that, we could think of the following algorithm to sort an array:

  - **Step 1:** Using the head-tail technique to break the array into smaller arrays, until we reach an array of two or one element
  - **Step 2:** Sort the array in the base case, which is trivial to sort and return it.
  - **Step 3:** Insert the head of the previous function call in the sorted array in the correct position, keeping the array sorted.

Note that **Step 3** is done after the recursive call, when we already have a sorted array.

Example. The code below adds a number to a sorted array, keeping the array sorted.

```
import numpy as np

sorted_arr = np.asarray([1, 2, 5, 10])
new_value = 8
i=0
while new_value>sorted_arr[i]:
    i+=1
    if i==sorted_arr.shape[0]:
        break
    new_arr = np.concatenate((sorted_arr[0:i],
                              np.asarray([new_value]),
                              sorted_arr[i:]))
print(sorted_arr)
print(new_arr)
```

Using the reasoning and code snippet above, write the recursive function `sort_array(arr : np.ndarray[int])` that returns a sorted array, sorted in ascending order.

Submission instructions

Extra documents:
If you have worked together with another student, create a .txt file called `programming_partner.txt` containing the S-number of both of you.

> **Example.** `programming_partner.txt`
>
> ```
> Recursive Problems 2:
> s012345
> s123456
>
> ```

Both students should submit their work individually on Brightspace.

Assignment 2C is part of Assignment 2, which has its deadline after **this week**. Submit your solutions to ASsignment 2A, 2B and 2C in Brightspace.

General submission:
Create a zip file called `Assignment2` containing the files:

- `mini_proj_3.py`, with the following functions:
  - random_array()
  - element_mult()
  - find_max()
  - transpose()
  - is_magic()
  - is_square()
- `recursive_problems_1.py`, with the following functions:
  - sum_even_rec()
  - array_product_rec()
  - concat_rec()
  - half_christmas_tree_rec()
  - find_max_rec()
  - main()
- `recursive_problems_2.py`, with the following functions:
  - move_max_rec()
  - is_substring()
  - find_combinations()
  - sort_array()
- `programming_partner.txt`

Submit your file to Assignment 2, in Brightspace.