TD : Recursion

Module

★ Exercice 1: Dichotomic search (a proud member of the Divide and Conquer family)

Dichotomic search (the word dichotomy comes from the ancient greek and means « cut in two ») is a search process where the search space is divided by two at each step. A typical example is the guess game where one of the participants mus guess a number randomly chosen between 1 and 100. The most efficient method lies in doing a dichotomic search as illustrated below :

- Question 1 : Is the number greater or equal to 50 ? (100 / 2)
- Yes
- Question 2 : Is the number greater or equal 75 ? ((50 + 100) / 2)
- No
- Question 3 : is the number greater or equal to 63 ? ((50 + 75 + 1) / 2)
- Yes

We reiterate these questions until we reach 65 (given as an example here). On average, dividing the search space by two is the fastest approach to solve such problems.

▷ **Question 1:** Write a recursive function searching the index of a given element in a sorted list. Your search function has to follow the dichotomic search principles.

INPUT :

— A sorted list (`tab`) of n integer elements
— The lower (`index_low`) and higher `index_high`) of the list
— The searched element (`element`)

OUTPUT : the index of the searched element in the list ; -1 if the element is not present. sinon.

No build-in functions from the standard python library are allowed.

▷ **Question 2:** Compute the complexity of this function.

▷ **Question 3:** Why does this function terminate ?

▷ **Question 4:** Build an iterative version of the function.

★ Exercice 2: The (in)-Famous Knapsack Problem

Over the next few hours (including the next lab), You will implement an exploration algorithm in a state-graph. Let's get confident with the problem first.

To **avoid drowning in your code**, one should not (and never) start implementing things without proper (1) understanding and (2) decomposition of the problem and the solution. In our approach, we will solve it in a specific way (that allows also automatic test). We therefore require the students to follow exactly the steps as given in the questions below and **strictly respect the given functions signatures**.

The Knapsack problem is a very classical optimization problem that exists in different flavors. The objective is, given a capacity-limited knapsack, choose as many objects as the knapsack can support and maximize the value. Similar problems appear in cryptography, in applied mathematics, in combinatory optimization, in complexity theory and in may other places.

PROBLEM :

> Given a set of objects having each a value v_i and a weight w_i, select those objects that maximise the value of the set while ensuring that the total weight does not exceed a given threshod N.
> In a first approach of the problem, we set $\forall i, v_i = w_i$. One can imagine for example that all objects are made of the same metal (hopefully gold :-).

▷ **Question 1:** How can we model this problem in Python ? Which data type would you use for this purpose ?

▷ **Question 2:** Compute the number of different knapsack configurations that can be built in an exhaustive search (brute force approach).

If you counted well, you will see that this is actually a lot :-)
A more efficient approach might be to apply a backtracking algorithm (and not explore invalid solutions). As seen in the lecture, this allows to literally cut explorations in the earliest stages. For example, when the bag is already full, it is not necessary to try to add new objects. The remainder of this exercice will focus on building such an algorithm.

▷ **Question 3:** Given the data structure model that we defined in the previous questions, write the following functions `putInBag()`, `removeFromBag()` et `isTaken()` which will simplify the usage of the data structures we chose to model the knapsack and its environment. Which parameters would you define for each of these functions ? What return types do they need ?

▷ **Question 4:** Given the model build with the teacher, write a function named `ks_value` which computes at any given time, the value of the content of a bag. Which parameters are needed for the function ? What return type ?

★ **Exercice 3: Recursive resolution of the knapsack problem**

As the exercise title states (and you will not be surprised), we will address the knapsack problem following a recursive approach. To this end, we will stepwise build an algorithm that will later be implemented into a function.

▷ **Question 1:** Draw the call tree that you function has to go through when given four objects with the following sizes $\{5, 4, 3, 2\}$ and a maximum capacity of 15 for the bag.

▷ **Question 2:** Lets do the same for a capacity of 8.

Based-on the observation of the calls, lets reify our function. It is a recursive function (that what we stated earlier). We therefore need to choose a parameter on which the recursion will operate. To build the signature of your function, you might follow a similar approach than the one taken for the Hanoï towers in the lecture. Lets assume your "core function" (entry to your problem) has the following name ks_search(),

▷ **Question 3:** What will the full signature (function name, list of parameters and return type) of this function be ?

▷ **Question 4:** These parameters may not be sufficient to explore the problem in a recursive way. Therefore, you may need to define a helper function (being it a lambda function or an inner one) with a different signature. This function will be named ks_search_rec. Which parameters does this function need to perform the recursive problem solving ?

The objective of your recursive function is to find the best node in the explored tree. Somehow, this is similar to searching a minimum value in a list (indexed as in an array) : one looks at all elements and for each, if it is better than the best candidate seen so far, it becomes the best candidate, ... and so on. The difference here is that we do not look for simple integers.

▷ **Question 5:** Write a simple ks_dupplicate() function that dupplicates a given knapsack. Which are the parameters ? what is the return type ?

▷ **Question 6:** We now have all ingredients to define the algorithm in a simple and efficient way. Get inspired by the tree exploration rolled out in the examples before to write (in plain text) your recursive algorithm. It should contain : the general case plus the operations for each stage of the recursion.

★ **Exercice 4: Generalized Knapsack Problem**

As stated in the introduction, knapsack problems exist in different flavors : strawberry, chocolate, mint, ... Lets taste (and code) a few of them.

▷ **Question 1:** Generalize you solution to solve knapsack problems where the value of objects is decorrelated from their weight (the $v_i = w_i$ statement does not hold anymore). The challenge is to maximize the value of the bag while keeping the weight under (or equal to) the maximum allow value for the bag.

▷ **Question 2:** Generalize your solution so that it can take multiple values of a same object. Assume that the stock is always bounded and given. Explain how you model this stock and the solution.

★ **Exercice 5: Now lets go to the lab and implement them all !**

▷ **Question 1:** Implement all functions defined during the TD

All files must be present in the same python file named : `knapsack.py`

It is understood that a knapsack is modeled as a sequence of boolean values (`list[bool]`

The following signatures are mandatory :

- `putInBag(bag: list[bool],oindex:int)->list[bool]`
- `removeFromBag(bag: list[bool],oindex:int)->list[bool]`
- `isTaken(bag: list[bool],oindex:int)->bool`
- `ks_value(bag: list[bool],weights:list[int])->int`
- `ks_search(weights:list[int],maxcapacity:int)->list[bool]` (entry fonction to the knapsack search)
- `permutations(numberObjects: int)-> list[list[bool]` (returns all permutations of length number of objects)

— `ks_search_rec(weights:list[int],bag: list[bool], index:int, capacity:int)-> list[bool])` (does the recursive job)

Make sure to have all your tests (everything outside your functions) in the `if __name__ == '__main__':` statement at the end of your code.

▷ **Question 2:** For a given input, enrich your code so that it counts the number of recursive calls.

If you need to add parameters to your function, rename it with the `_count` suffix. For example, `ks_search` could become `ks_search_count` nad change the return types to add the counter in return (return tuples).

▷ **Question 3:** Now test your code on increasingly large instances of the problem. Measure the time it takes (you can use the timeit library as you did it in the previous lab). What can you conclude from your observations ?

▷ **Question 4:** Implement the two generalized versions fo the problem, measure, analyze and get a coffee !