CSCI-SHU 210 Data Structures — 100 Points

HOMEWORK ASSIGNMENT 2 - ANALYSIS OF ALGORITHMS

PROBLEM 1 – DEEPEST ELEMENT IN NESTED LIST - 25 POINTS

Write a recursive Python function that takes a list of lists l and returns the deepest nested element.

Requirements

- You have to use recursion. Non-recursive solutions are awarded 0 points.
- You can only create a constant amount of variables of constant space complexity O(1)
- You can only use primitive variables (no stack, no queue, no string, etc.)
- You cannot cast l to string or any representation other than a list.
- You are not allowed to use any third-party or built-in libraries.

Important

- There is no solution in which two or more lists have equally deep elements as their deepest element. This means that you don't have to consider cases such as: [[[1]] , [[2]]]

Example 1

- Input: $s = [[[7, 1]]]$
- Calculation: $res = deepest(s)$
- Result: $[7, 1]$

Example 2

- Input: $s = [[[7, [8, [9]]]]]$
- Calculation: $res = deepest(s)$
- Result: $[9]$

Problem 2 – knapsack - 25 Points

The knapsack is an optimization challenge where given a set of items with associated weights and values, the goal is to select a subset of items to maximize the total value, without exceeding a predefined weight limit (capacity of the knapsack). Write a recursive Python function to solve the knapsack problem given a list of items with their weights and values, and a maximum weight capacity for the knapsack.

Requirements

- You have to use recursion. Non-recursive solutions are awarded 0 points.
- You can only create a constant amount of variables of constant space complexity O(1).
- You can only use primitive variables (no stack, no queue, no string, etc.).
- You are not allowed to use any third-party or built-in libraries.

Example 1

- weights = [1, 3, 4, 5]
- values = [1, 4, 5, 7]
- capacity = 7
- n = len(weights)
- result = knapsack_recursive(weights, values, capacity, n)
- print(result) # Output: 9 because weights 3 and 4 have been selected with values 4 and 5.

Problem 3 – Element Uniqueness Problem - 25 Points

The *element uniqueness problem* is to dertermine whether there are duplicates in a given S of n elements. Implement an efficient recursive function *unique(S)* for solving the element uniqueness problem, which runs in time that is at most *O(n^2)* in the worst case.

Important

- You are not allowed to sort the list. Elements may not be compatible for cxomaprison.
- You are not allowed to lose any form of loops, including for and while loops.
- You are not allowed to use the __contains__ function or use the x in S syntax.
- You are not allowed to use any third-party or built-in libraries.
- You are not allowed to use slicing.

Example 1

- Input: unique([1, 54, 3, 25, 39, 25, 2])
- Returns: False

Example 2

- Input: unique([9, 'a', [], [[35, 2], ['NYU']], (100,)])
- Returns: True

Problem 4 – Stone Allocation - 25 Points

Suppose you have N ≥ 1 stones, and M ≥ 1 days to throw at a target. Each day, you can throw 1 or 2 or 3 stones. In addition, on an even number of days, you can only throw 1 or 3 stone(s) as a constraint. Implement a recursive function throw_stones(N, M) to return all possible valid sequences to throw N stones in M days under such constraint.

Requirements

- You are not allowed to lose any form of loops, including for and while loops.
- You are not allowed to use PYthon dictionary to store temporary results.
- Your result could be returned in a list of lists, or a list of tuples. The order does not matter.
- Your solution has to be recursive.

Example 1

- When N = 5, M = 3,
- valid sequences:

Day 1	Day 2	Day 3
1	1	3
2	1	2
3	1	1
1	3	1

- invalid sequences:

Day 1	Day 2	Day 3		
3	2	1		
3	1	2		
1	1	1	1	1

- Input: throw_stones(5, 3)
- Returns: [(1, 1, 3), (2, 1, 2), (3, 1, 1), (1, 3, 1)]