CSE 431/531: Algorithm Analysis and Design — Fall 2022

Homework 4

Instructor: Shi Li — **Deadline: 11/13/2022**

Github link to access solutions and run them: https://github.com/AbhilashBharadwaj

Problem 1 Solve the matrix-chain-multiplication instance with the following sizes.

matrix	A_1	A_2	A_3	A_4	A_5
size	3×4	4×10	10×6	6×8	8×7

You need to fill the following two tables for the *opt* and π values, give the minimum cost of the instance (i.e., the number of multiplications), and describe the best way to multiply the matrices (using either a tree, or a formula with parenthesis).

$opt[i,j]$ (i \ j)	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

$\pi[i,j]$ (i \ j)	1	2	3	4	5
1					
2					
3					
4					

Table 1: *opt* and π values for the matrix chain multiplication instance.

The minimum cost for the instance is ____.
Describe the best way to multiple the matrices:

Solution

The minimum cost for the instance is 612. Best way to multiply is (((A1.A2)A3)(A4.A5))

opt[i,j]

i → / j ↓	1	2	3	4	5
1	0	120	300	444	612
2		0	240	432	656
3			0	480	756
4				0	336
5					0

π[i,j]

i → / j ↓	1	2	3	4	5
1		1	2	3	4
2			2	3	4
3				3	3
4					4

Problem 2 An independent set of a graph $G = (V, E)$ is a set $U \subseteq V$ of vertices such that there are no edges between vertices in U. Given a graph with node weights, the maximum-weight independent set problem asks for the independent set of a given graph with the maximum total weight. In general, this problem is very hard. Here we want to solve the problem on trees: given a tree with node weights, find the independent set of the tree with the maximum total weight. For example, the maximum-weight independent set of the tree in Figure 1 has weight 47.

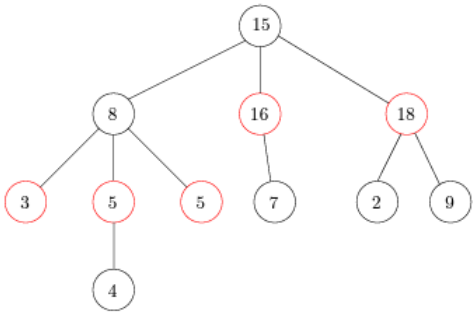

Figure 1: The maximum-weight indpendent set of the tree has weight 47. The red vertices give the independent set.

Design an $O(n)$-time algorithm for the problem, where n is the number of vertices in the tree. We assume that the nodes of the tree are $\{1, 2, 3, \cdots, n\}$. The tree is rooted at vertex 1, and for each vertex $i \in \{2, 3, \cdots, n\}$, the parent of i is a vertex $j < i$. In the input, we specify the weight w_i for each vertex $i \in \{1, 2, 3, \cdots, n\}$ and the parent of i for each $i \in \{2, 3, \cdots, n\}$.

Solution

Pick the current node, add its weight and find the max weighted cell of its children and grandchildren

Exclude the current node's weight from being picked to the set, explore the node's children, and find their max-weighted cell

Time complexity:

The algorithm runs in O(n) because every node is calculated only once.

```python
class MaximumIndependentSet:
    def get_max_independent_set(parent_positions, node_weights):
        length = len(parent_positions)
        current_node = length - 1

        #  length x 2 array
        table = [[0 for _ in range(2)] for _ in range(length)]
        # For each node, we have two options: include it or not include it
        # table[i][0] indicates if i should be picked up for the maximum independent set
        # table[i][1] indicates if i should not be picked up for the maximum independent set

        while (current_node > 1):
            child_node = current_node
            table[parent_positions[child_node]
                  ][0] = node_weights[parent_positions[child_node]]
            while (child_node > 0 and parent_positions[current_node] == parent_positions[child_node]):
                table[child_node][0] = max(
                    table[child_node][0], node_weights[child_node])
                table[parent_positions[current_node]
                      ][0] += table[child_node][1]

                table[parent_positions[current_node]
                      ][1] += max(table[child_node][0], table[child_node][1])
                child_node -= 1
            current_node = child_node

            max_value = table[current_node][0] if table[current_node][0] > table[current_node][1] else table[current_node][1]
        return max_value
```

✓ 0.1s

```python
# Example from the question

# Parent node postion of each node.
# The root node has a parent position of 0
parent_positions = [0, 0, 1, 1, 1, 2, 2, 2, 3, 4, 4, 6]
node_weights = [0, 15, 8, 16, 18, 3, 5, 5, 7, 2, 9, 4]
                # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 => node index

print(MaximumIndependentSet.get_max_independent_set(
    parent_positions, node_weights))
```

✓ 0.1s Python

```
47
```

```python
# Example from class slides
parent_positions = [0, 0, 1, 1, 2, 2, 2, 3, 3, 5, 5, 8]
node_weights = [0, 10, 5, 8, 4, 4, 9, 3, 11, 2, 7, 8]
                # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 => node index
print(MaximumIndependentSet.get_max_independent_set(
    parent_positions, node_weights))
```

Python

```
46
```

Problem 3 Given a sequence $A = (a_1, a_2, \cdots, a_n)$ of n numbers, we need to find the longest increasing sub-sequence of A. That is, we want to find a maximum-length sequence $(i_1, i_2, \cdots, i_t)$ of integers such that $1 \leq i_1 < i_2 < i_3 < \cdots < i_t \leq n$ and $a_{i_1} < a_{i_2} < a_{i_3} < \cdots < a_{i_t}$.

For example, if the input $n = 11$, $A = (30, 60, 20, 25, 75, 40, 10, 50, 90, 70, 80)$, then the longest increasing sub-sequence of A is $(20, 25, 40, 50, 70, 80)$, which has length 6. The correspondent sequence of indices is $(3, 4, 6, 8, 10, 11)$.

Again, you only need to output the length of the longest increasing sub-sequence. Design an $O(n^2)$-time dynamic programming algorithm to solve the problem.

Solution

```python
input_arr = [30, 60, 20, 25, 75, 40, 10, 50, 90, 70, 80]
n = len(input_arr)

memo = [1 for _ in range(n)]  # 1 as 1 is the smallest sub_seq

'''
The simplistic method would compute lists of sub-sequences for each element in the list,
necessitating a continuous recalculation of the subsequence length.


Instead, by storing the LIS up to index i and updating the memoization list to include any smaller elements that come before index i
we can reduce this by extending the longest sequence to date.

'
res = 0
for i in range(n):
    j = 0
    while j < i:
        if input_arr[j] < input_arr[i]:
            memo[i] = max(memo[i], memo[j]+1)
            res = memo[i]
        j += 1
    # find max of the memoized sequence lengths
    res = max(res, memo[i])
print("Length of longest increasing sequence is ", res, "\n for input", input_arr)
```

✓ 0.1s Python

```
Length of longest increasing sequence is  6
 for input [30, 60, 20, 25, 75, 40, 10, 50, 90, 70, 80]
```

Time complexity:

The algorithm runs in O(n^2) because the loop runs n times and rechecks the maximum for each iteration.