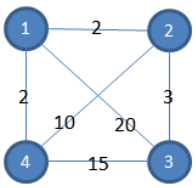
Prob. 1	Prob. 2	Prob. 3	Prob. 4

Problem 1.

This is exactly what the Floyd-Warshall algorithm does. It's a dynamic programming algorithm using three nested loops from 1 to n. Here's how it works.

The idea is to check for each pair of vertices (v_i and v_j) the cost of travelling from one to another directly (without passing through another vertex). If there is no edge linking two vertices we set the cost to infinity. We can store those values in an array such that $path\_cost[i][j]$ equals this cost. To do so we can implement it using two nested loop one iterating i from 1 to n and the second for j from 1 to n.
Now we want to calculate the cost from each pair of vertices passing through v_1. We then update $path\_cost[i][j]$ with the minimum between $path\_cost[i][j]$ and $path\_cost[i][1]+path\_cost[1][j]$. We do that n times for each vertex. So we need to nest the two first loops in a new loop. We have then three nested loops, all from 1 to n (we can either initialize the array $path\_cost$ before in a two nested loops or we can have the first loop starting from 0 to initialize the array). Thus the complexity is $O((n+1)\cdot n\cdot n) \leq O(n^3)$.



	1	2	3	4
1	0	2	20	2
2	2	0	3	10
3	20	3	0	15
4	2	10	15	0

Update using vertex 1

	1	2	3	4
1	0	2	20	2
2	2	0	3	4
3	20	3	0	15
4	2	4	15	0

Update using vertex 2

	1	2	3	4
1	0	2	5	2
2	2	0	3	4
3	5	3	0	7
4	2	4	7	0

Vertex 3 and 4 won't help any further

Figure 1: Execution of Floyd-Warshall

Problem 2.

Problem 3.

Problem 4.

Our recursive *dynamic programming* algorithm:

- Divide the sequence of matrices into two sequences
- Find the cost of the multiplication of each sequence.
- If the cost of the multiplication is in the table, then simply return it. Otherwise, recursively calculate and add final cost into the cost table. Here, base case is the multiplication of two matrices. When base is arrived, we just return the cost: If we have two M_1 is a $k \times l$ matrix and M_2 is a $l \times m$ matrix, then the cost of the multiplication C is klm. For example, $M_1(20 \times 30)$ and $M_2(30 \times 10)$, cost C is $20 \cdot 30 \cdot 10 = 6000$.
- Sum up the costs of the subsequences and record it as a candidate for the optimal cost of the original sequence.
- Repeat above three steps for original sequence where it can be divided into two sequences and take the minimum of all calculated candidate costs and store it into the cost table for the cost of the original sequence.

A sequence is defined by its *beginning* and *ending* indices. Hence, in the cost table, we need $O(n^2)$ space since the number of sequences are defined as in the following:

$$\sum_{i=1}^{n-1} \sum_{j=i}^{n-1} 1 \leq O(n^2)$$

As a result of this number of sequences, we need at least $O(n^2)$ subgrouping computations. However, we also go over the sequence to find the minimum of each possible subgrouping so each entry of the table requires us to scan the sequence which takes $O(n)$ time. That's why defined *top-down dynamic programming* algorithm takes $O(n^3)$ time at overall which is a big improvement over intuitive *brute-force* $O(2^n)$ algorithm.