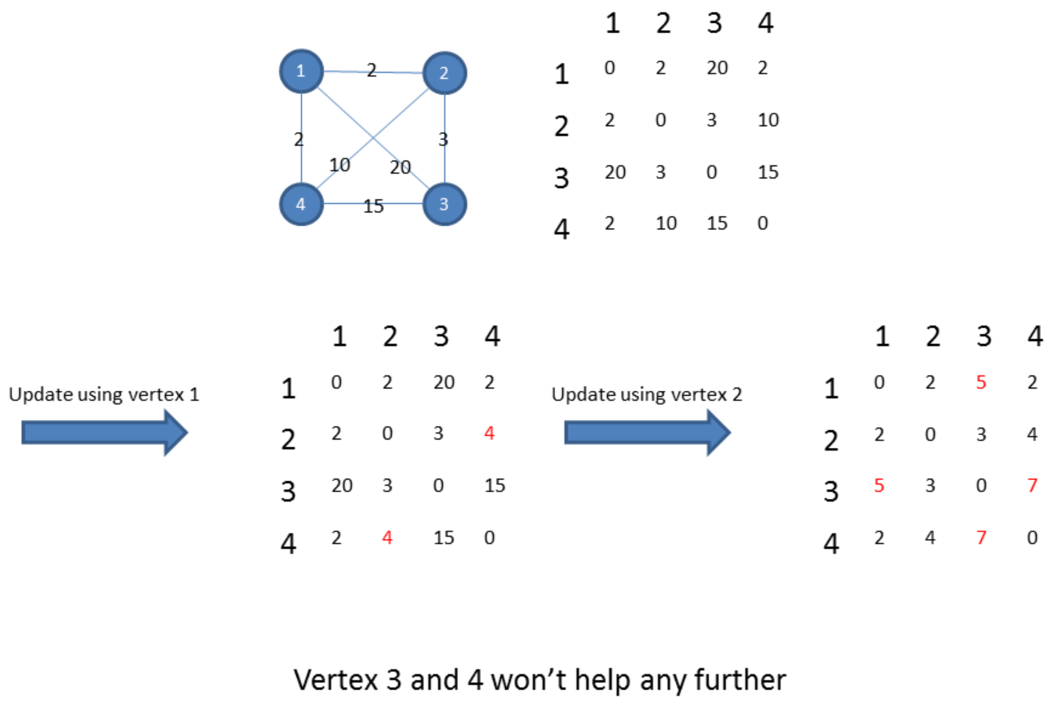
Prob. 1	Prob. 2	Prob. 3	Prob. 4

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 $O((n+1) \cdot n \cdot n) \le O(n^3)$.



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

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

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

Figure 1: Execution of Floyd-Warshall

Problem 2.

Our *dynamic programming* algorithm:

- By using *depth-first* search(choosing an arbitrary node as a root), we calculate three arrays:
  - $d1[i]$: minimum sum achievable by only using node i and its descendants.
  - $p[i]$: predecessor of node i in the path found for $d1[i]$.
  - $d2[i]$: second minimum sum achievable by only using node i and its descendants, a path that is edge-disjoint relative to the path found for $d1[i]$.
- When we calculate these arrays, we can get the overall minimum sum by $\min(d1[k], d2[k], d1[k] + d2[k])$ over all nodes k because path can be in two shapes:
  - Straight path from node k to its descendants or bend over at node k:

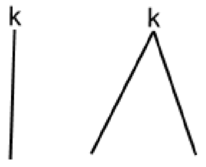


Figure 2: Basic shapes of the minimum path

- Calculation of $d1[i]$:
  - Run a depth-first search from the root node(chosen arbitrarily). In the DFS, keep track of the current node and its predecessor. At each node, assume its children are $n_1, n_2, \ldots n_k$. Then, $d1[i] = \min(\min(d1[n_1] + cost[i, n_1], d1[n_2] + cost[i, n_2], \ldots, d1[n_k] + cost[i, n_k]), \min(cost[i, n_k]))$. Also note that, $d1[i]$ is set to 0 if node i is a leaf. Moreover, we update $p[i]$ with the selected predecessor to retrieve the path later.
- Calculation of $d2[i]$:
  - Run a depth-first search from the root node(chosen arbitrarily which is same node used in $d1[i]$ calculations). Apply the same logic we did for $d1[i]$, except when going from a node i to one of its children k, update $d2[i]$ only if $p1[i] \neq k$. We do this for all children recursively.
  - $d2[i] = \min(d2[i], d1[k] + cost[i, k], cost[i, k])$ where node k is the children of node i.

We call depth-first twice, which is linear, to calculate $d1[i], d2[i]$ and $p[i]$ arrays. We also go over of all vertices again to take the minimum sum by possible $d1[i], d2[i]$ or merge of both. Therefore, overall complexity remains *linear*.

Problem 3.

To find a minimum triangulation of a convex polygon, we have to first make some general statements and observations. First, it has to be noted that for every possible triangulation of a polygon, there is a triangle that if removed from the polygon it splits it into two distinct polygons. Moreover, these two polygons are also validly triangulated (or are triangles). For a minimum triangulation (one where the edges of the triangles are the shortest possible), if we split the polygon into two as described above, the triangulation of each of the two triangles must also be minimum. Also, for a polygon with 4 edges, the optimal triangulation would be the one with the smallest diagonal. We can also notice that in every triangulation, each edge corresponds to exactly one triangle while each inner edge corresponds to exactly two triangles. Thus the cost of the triangulation would be the sum of the perimeter of each triangle decrease by the sum of the edges of the polygon and divided by two. However, since the sum of the edges of the polygon is constant, minimising the sum of the perimeters of each triangle, would correspond to minimising our triangulation.

Thus we can devise a recursive dynamic algorithm that will start splitting the polygon into smaller ones (by selecting a diagonal), trying to find an optimal triangulation of the smaller triangles. In the initial polygon there are $\binom{n}{2}$ chords that we could start from. In order to avoid recalculating costs when same polygons occur, we are going to use memoization and store the minimal triangulation (and its cost) of each smaller polygon the first time we calculate it so that we can have access to it every time it is needed. Thus the algorithm would be the following:

Consider that since we have the list of its vertices, in counterclockwise order along the perimeter, we can consider a sub-polygon that is defined as $Sub(i,j)$ starting at vertex i and ending at vertex j containing all other vertices in counterclockwise order. As such, we initialize matrix SP, which will contain cost of triangulations of each sub-polygon, with zeros. $SP(1,n)$ will contain cost for polygon starting at vertex 1 and ending at vertex n, this is what we are looking for. Fill SP as follows: $SP(i,k) = min_j\{SP(i,j) + SP(j,k) + perimeter(i,j,k)\}$ Where $k > i+1$, $j \in [i,k]$ and perimeter(i,j,k) is the perimeter of the triangle with points i,j and k.

Running-time:

The time to calculate the values of the matrix would be $O(n^2)$ (since there are n^2 elements in the matrix). Each element requires $O(n)$ operations as we need to iterate over j in order to test every possible triangulation and pick the minimum. Hence, the algorithm is $O(n^3)$.

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 sub-sequences 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 $O(n^2)$ sub-sequence computations. However, we also go over the sequence to find the minimum of each possible sub-sequence 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.