Dynamic Programming

Solutions to the Practical Exercises

DA 2023 Instructors Team

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2023

A - Fundamentals

Exercise 1

a) The base case corresponds to when n = 0 or n = 1, while the inductive step corresponds to when n > 1.

$$n! = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ n * (n-1)! & n > 1 \end{cases}$$

b) See source code. The Dynamic Programming (DP) matrix in this exercise is unidimensional (1D).

c) See source code.

d) Both solutions have a temporal complexity of Θ(n) since all factorials from 1 to n need to be computed in order to find n!.

The recursive solution has a spatial complexity of Θ(n) due to space occupied by the stack in each of the n recursive calls needed to find n!.

On the other hand, the iterative solution has a spatial complexity of Θ(1) since only (k-1)! is needed to find k! (for 0 < k ≤ n).

Exercise 2

a)

$$minCoins(i,k) = \begin{cases} 0 & k = 0 \\ +\infty & k > 0 \wedge i = 0 \\ a & 0 < c_i \leq k \wedge i > 0 \wedge a \leq b \\ b & otherwise \end{cases}$$

where:

- $a = 1 + minCoins(i, k - c_i)$
- $b = minCoins(i-1, k)$

a represents using one (more) coin of value c_i.

b represents not using (any more) coins of value c_i.

$$lastCoin(i,k) = \begin{cases} NULL & k = 0 \\ NULL & k > 0 \wedge i = 0 \\ c_i & 0 < c_i \leq k \wedge i > 0 \wedge a \leq b \\ b' & otherwise \end{cases}$$

where:

- $b' = lastCoin(i-1, k)$

b)Table for minCoins:

i \ k	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	+∞	+∞	+∞	+∞	+∞	+∞	+∞	+∞
1 (1 c)	0	1	2	3	4	5	6	7	8
2 (2 c)	0	1	1	2	2	3	3	4	4
3 (5 c)	0	1	1	2	2	1	2	2	3
4 (10 c)	0	1	1	2	2	1	2	2	3

The values in yellow represent cases where minCoins(i, k) ≤ minCoins(i − 1, k) (for i > 0), i.e. where using a new coin denomination led to an `improved solution.`

Table for lastCoin:

i \ k	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
1 (1 c)	NULL	1	1	1	1	1	1	1	1
2 (2 c)	NULL	1	2	2	2	2	2	2	2
3 (5 c)	NULL	1	2	2	2	5	5	5	5
4 (10 c)	NULL	1	2	2	2	5	5	5	5

Filling the table for lastCoin is much easier using the values highlighted in yellow for minCoin: for the cases where i > 0 and k > 0:

- when the value is yellow, then the cell is equal to c_i;
- otherwise, it is equal to the value in the cell above.

c) See source code. The matrix is filled starting from the first line (i = 0) and moving downwards (increasing i). For a given line (i.e. a given value of i), the matrix is filled from left to right (increasing k, starting at 0).

d) The algorithm uses a DP matrix with dimensions (n + 1) x (T + 1). Thus, the number of cells is $\Theta((n + 1)*(T+1)) = \Theta(n*T + n + T + 1) = \Theta(n*T)$.

The algorithm has two main steps:

- Filling the cells of the DP matrix also takes $\Theta(n*T)$ time since computing the value of each cell takes O(1) time.
- Building the solution (i.e. determining the coins used to obtain T) takes O(T) time in the worst case (if only 1 cent coins are used).

Thus, the temporal complexity is dominated by the first step, which results in the temporal complexity being $\Theta(n*T)$.

At a first glance, the spatial complexity could be defined as $\Theta(n*T)$ due to the DP matrix's dimensions. However, in reality, only the current line of the DP matrix being processed is needed for the computations. Therefore, the algorithm only requires for the memory to reserve space for one line at a time, reducing the spatial complexity to $\Theta(T)$.

Note: A $\Theta(n*T)$ temporal complexity does not mean that the algorithm runs on polynomial time. To be more accurate, one can express the temporal complexity with respect to the number of bits used to represent T (which intuitively corresponds to T's size), called TBits. This make it more evident that the temporal complexity is actually exponential: $\Theta(n*2^{TBits})$. In fact, it runs in pseudo-polynomial time as it is polynomial to the input's magnitude, but exponential to the input' size.

Exercise 3

a) See source code. The main idea of the algorithm is to use a 3D DP matrix where the dimensions represent:
- the coin denominations available (i), similarly to exercise 2.
- the maximum amount of stock that is usable by coin i (s).
- the change to be paid (k), similarly to exercise 2.

b) The algorithm uses a DP matrix with dimensions (n + 1) x (S + 1) x (T + 1). Thus, the number of cells is $\Theta((n + 1)*(S + 1)*(T+1)) = \Theta(n*S*T)$.

Using the same reasoning as in exercise 2, the temporal complexity is $\Theta(n*S*T)$ as each cell takes O(1) to be filled.

The spatial complexity is $\Theta(n*S*T)$ since values from multiple lines are needed to fill a cell, which prevents a clear spacing-saving trick from being used.

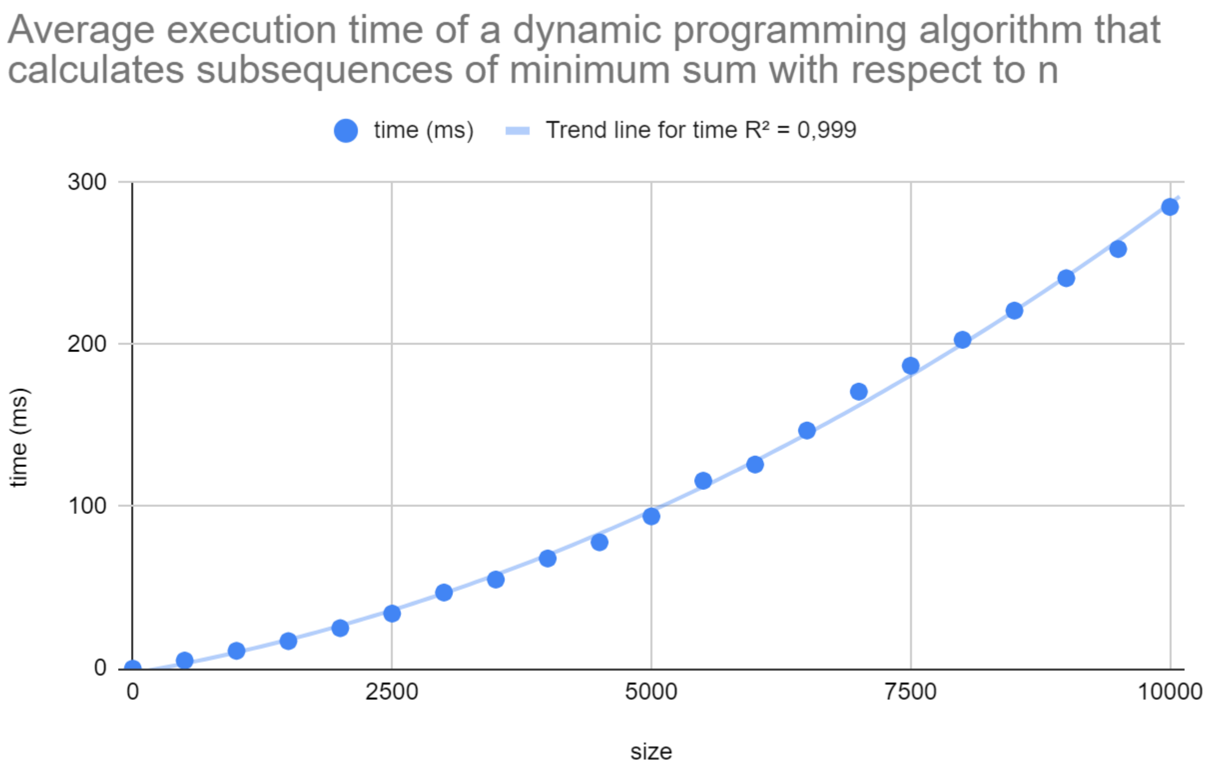
Exercise 4

a) See source code.

b) For all (N + 1) possible lengths i between 0 and N, it computes the sum of the subsequence of length i with the smallest sum. Computing each sum takes O(n) time, at most, each index of the sequence is considered as the starting index.

Therefore, the temporal complexity is $O(n*n) = O(n^2)$.

c) The results below were obtained using an Intel Core i7-8750H processor (2.20 GHz) and a 32 GB DDR4 RAM (2667 MHz).



Exercise 5

a) See source code.
b) See source code.
c) Temporal complexities:

Technique	Computation of S(n,k)	Computation of B(n)
Recursion	$O(n*k)$	$O(n^2)$
Iterative DP	$O(n*k)$	$O(n^2)$

Both programming techniques produce the same temporal complexity.

d) Spatial complexities:

Technique	Computation of S(n,k)	Computation of B(n)
Recursion	$O(n*k)$	$O(n^2)$

Iterative DP	O(n)	O(n)

Similarly, to exercise 1, the iterative algorithm uses less memory by using the observation that only one line of the DP matrix needs to be loaded at any given time, for this specific problem and DP matrix formulation.

Exercise 6

a) See source code. The implementation uses Kadane's algorithm, which runs in O(n) time.

b) The temporal complexities indicate that the DP approach is the one that scales the best in terms of execution time when n increases, while the brute-force approach is the one that scales the worst. This is corroborated by the experimental results below.

n	DP time (s)	BF time (s)	DC time (s)
0	6,40E-08	6,00E-08	2,80E-08
500	4,21E-06	8,18E-02	1,54E-04
1000	6,38E-06	6,50E-01	1,95E-04
1500	9,30E-06	2,17E+00	3,19E-04
2000	1,59E-05	5,21E+00	3,92E-04
2500	1,96E-05	1,03E+01	4,86E-04
3000	1,87E-05	1,77E+01	5,98E-04
3500	2,12E-05	(too long to compute)	7,66E-04
4000	3,51E-05		7,95E-04
4500	4,31E-05		8,72E-04
5000	4,26E-05		9,01E-04
5500	3,67E-05		9,86E-04
6000	2,96E-05		1,04E-03
6500	2,34E-05		1,13E-03
7000	2,50E-05		1,23E-03
7500	4,61E-05		1,32E-03
8000	2,90E-05		1,40E-03
8500	3,20E-05		1,49E-03
9000	3,30E-05		1,62E-03
9500	3,47E-05		1,69E-03
10000	3,59E-05		1,74E-03

Exercise 7

a) As seen in the divide-and-conquer solution, the optimal solution (i.e. with the minimum possible amount of moves) of the problem with n disks can be expressed using the optimal solution of problems with n-1, thus this problem has the Optimal Substructure property, one of the properties needed for dynamic programming to be applicable.

This problem has Overlapping Sub-problems since the solution of some sub-problems is needed multiple times to compute the solution of the largest problem. For instance, using the nomenclature of the Hanoi towers divide-and-conquer pseudo-code algorithm from the TP4 sheet, the solution of hanoi(1,A,B,C) is required twice to solve hanoi(3,A,B,C). Therefore, since the problems are not interdependent, it makes more sense to use dynamic programming, where the solution of previously-solved sub-problems is stored for future reference, which saves execution time.

b) See source code.

Exercise 8

a) The reasoning for this exercise has some similarities to the unlimited coin-change problem.

$$maxValue(i,k) = \begin{cases} 0 & k = 0 \\ 0 & k > 0 \wedge i = 0 \\ a & 0 < weights_i \leq k \wedge i > 0 \wedge a \geq b \\ b & otherwise \end{cases}$$

where:

- $a = values_i + maxValue(i-1, k - weights_i)$
- $b = maxValue(i-1, k)$

a represents using the i-th item (indices starting at 1). Notice that a is not maxValue(i, k - weights$_i$) to prevent the same item from being used twice. The symbol b represents not using the i-th item.

$$lastItem(i,k) = \begin{cases} NULL & k = 0 \\ NULL & k > 0 \wedge i = 0 \\ i & 0 < weights_i \leq k \wedge i > 0 \wedge a \geq b \\ b' & otherwise \end{cases}$$

where:

- $b' = lastItem(i-1, k)$

b)Table for maxValue:

i \ k	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1(v:10,w:1)	0	10	10	10	10	10
2(v:7,w:2)	0	10	10	17	17	17
3(v:11,w:1)	0	11	21	21	28	28
4(v:15,w:3)	0	11	21	21	28	36

The values in yellow represent cases where maxValue(i, k) ≥ maxValue(i - 1, k) (for i > 0), i.e. where using the i-th item led to an improved solution.

Table for lastItem:

i \ k	0	1	2	3	4	5
0	NULL	NULL	NULL	NULL	NULL	NULL
1(v:10,w:1)	NULL	1	1	1	1	1
2(v:7,w:2)	NULL	1	1	2	2	2
3(v:11,w:1)	NULL	3	3	3	3	3
4(v:15,w:3)	NULL	3	3	3	3	4

Filling the table for lastItem is much easier using the values highlighted in yellow for maxValue: for the cases where i > 0 and k > 0:

- when the value is yellow, then the cell is equal to i;
- otherwise, it is equal to the value in the cell above.

c) See source code. Unlike the recurrence formula presented in this exercise, the item indices start at index 0 to save some execution time and show a slightly more varied solution to the analogue one for the unlimited coin change problem. Thus, the algorithm uses a DP matrix with dimensions n x (maxWeight + 1). The matrix is filled starting from the first line (i = 0) and moving downwards (increasing i). For a given line (i.e., a given value of i), the matrix is filled from left to right (increasing k, starting at 0).

d) The algorithm uses a DP matrix's number of cells is Θ(n*(maxWeight+1)) = Θ(n*maxWeight + n) = Θ(n*maxWeight).

The algorithm has two main steps:

- Filling the cells of the DP matrix also takes Θ(n*T) time since computing the value of each cell takes O(1) time.
- Building the solution (i.e. determining the items used in a knapsack with capacity maxWeight) takes O(maxWeight) time in the worst case (if all items were used).

Thus, the temporal complexity is dominated by the first step, which results in the temporal complexity being Θ(n*maxWeight).

The spatial complexity is Θ(n*maxWeight) due to the DP matrix's dimensions. Unlike the analogue solution to the unlimited coin change problem, no space-saving trick was used to ensure it is possible to find all the items used for the optimal solution.

Note: Similarly, to exercise 2, the temporal complexity is pseudo-polynomial and can be expressed using the number of bits used to represent maxWeight's size, maxWeightBits, which leads to an exponential complexity: $\Theta(n*2^{\text{maxWeightBits}})$.

Exercise 9

a) The recurrence formula is:

$$D(i,j) = \begin{cases} i & j = 0 \\ j & i = 0 \\ min(1 + D(i-1,j), 1 + D(i,j-1), D(i-1,j-1)) & i \geq 1, j \geq 1,\ A(i) = B(j) \\ min(1 + D(i-1,j), 1 + D(i,j-1), 1 + D(i-1,j-1)) & i \geq 1, j \geq 1,\ A(i) \neq B(j) \end{cases}$$

In the two recursive expressions (for when i and j are both greater than 0), each term of min corresponds to an operation:

- the 1st term corresponds to addition;
- the 2nd term corresponds to deletion;
- the last term corresponds to substitution (which has a cost of 0 if the last character of both string prefixes is equal).

Let |A| and |B| be the respective lengths of strings A and B. The edit distance between A and B is captured in the table entry located at D(|A|-1,|B|-1).

b) The computation of the edit distance can be detailed by filling in the dynamic programming table, as presented below.

	ε	m	o	n	e	y
ε	0	1	2	3	4	5
n	1	1	2	2	3	4
o	2	2	1	2	3	4
t	3	3	2	2	3	4
e	4	4	3	3	2	3

Thus, only 3 operations are needed to convert "money" to "note": swap the "m" with an "n", swap the "n" with a "t" and remove the "y".

c) See source code.

d) According to the recursive formula, all the cells of the dynamic programming table are needed to compute the edit distance (i.e. to compute the bottom right cell). Each cell takes O(1) time to be filled assuming that characters in strings A and B can be accessed and compared in O(1) time, which is possible if the strings are implemented as an array/vector, with direct access to any element by index. Therefore, given the dynamic programming table has (|A| + 1) by (|B|

+ 1) dimensions, the temporal complexity of the edit distance computation is O((|A| + 1)*(|B| + 1)) = O(|A|*|B|).

Initially, one could consider the spatial complexity to be O(|A|*|B|) due to the dynamic programming table's dimensions. However, to compute each cell, three cells are needed at most: the one to its left, the one above it and the one adjacent to the top-left corner. Thus, during the computations, only two lines of the table are needed to be stored in memory at any given time, thus reducing the spatial complexity to O(2*|A|) = O(|A|).

Note: since the edit distance of A and B is the symmetric (i.e. converting A to B requires the same number of steps as converting B to A), the transpose of the dynamic table can be computed if |B| < |A|, producing a spatial complexity of O(|B|).
e) See source code.

B - Applications to the shortest-path problem

Exercise 10
See source code.

Exercise 11
See source code.

Exercise 12
a) See source code.

b) See source code.

Exercise 13
a) For both vehicles, finding the optimal path corresponds to the **single-source shortest path problem**, where the distance corresponds to the amount of energy consumed. For vehicle A, all the graph's edge weights are positive, so Dijkstra's algorithm can be used.

The table below details the computations for the Dijkstra algorithm. In each iteration, the table details: the current vertex being processed, the updates to vertices' shortest distance and previous node in the shortest distance, and the current state of the heap/priority queue.

Iteration	Current vertex (dist, path)	Vertices (dist, path)	Heap [top .. bottom]
1	a	b(2, a) c(3, a)	[b, c]
2	b(2, a)	d(7, b) e(5, b)	[c, e, d]
3	c(3, a)	e(4, c)	[e, d]
4	e(5, b)	f(15, e)	[d, f]
5	d(7, b)	f(14, d)	[f]
6	f(14, d)	_	[]

The algorithm ends once the heap is empty.

By reading the table, for vehicle A, the most energy efficient path from node a to f consumes 14 MJ and traverses the following nodes in order: a -> b -> d -> f

b) For vehicle B, the edges in green have negative weights, so the Bellman-Ford algorithm can be applied.

The table below details the computations for the Bellman-Ford algorithm. The value of each cell is a pair (shortest distance, previous node in the shortest distance). Iteration 0 represents the initial values for each node. An equal sign (=) is used when the value for a node is the same as in the previous iteration.

Note: the graph's edges are processed in alphabetical order of the source node, and, in case of a tie, in alphabetical order of the destination node.

Node \ Iteration	0	1	2	3	4	5	6
a	0, NULL	=	=	=	=	=	=
b	+∞, NULL	2, a	=	1, e	=	=	=
c	+∞, NULL	3, a	=	=	=	=	=
d	+∞, NULL	+∞, NULL	7, b	=	6, b	=	=
e	+∞, NULL	+∞, NULL	4, c	=	=	=	=
f	+∞, NULL	+∞, NULL	+∞, NULL	14, d	=	13, d	=

The algorithm ends after 6 iterations since none of the distances is updated.

By reading the table, for vehicle B, the most energy efficient path from node a to f consumes 13 MJ and traverses the following nodes in order: a -> c -> e -> b -> d -> f

Therefore, vehicle B can travel from a to f while consuming 1 MJ less.

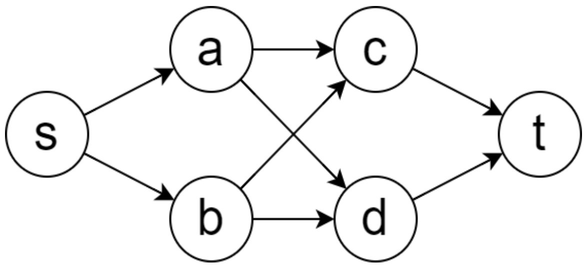
c) A cycle composed of only green edges would correspond to a path where energy was produced. Therefore, by having vehicle B continuously traverse this path, an infinite amount of energy could be created, which is not physically possible. Therefore, negative weight cycles do not make sense in this problem.

Exercise 14

Firstly, Dijkstra's algorithm is greedy as it selects the best local option according to a certain cost function at every choice point. In this algorithm, each choice point corresponds to an iteration of the main loop, where the next node from the queue to be explored is selected. Dijkstra's algorithm chooses to explore the node that minimizes the distance to the origin node.

Secondly, Dijkstra uses dynamic programming. It defines the shortest distance from the source node to each other node using a recurrence relation. In fact, the array with the shortest distance found so far from the source node to each other node can be considered as a dynamic programming matrix, where each element is updated iteratively based on the values of the other elements (i.e. the shortest distances are interdependent). In addition, the single-source shortest path problem has the two fundamental properties for dynamic programming to be applicable:

- Optimal Substructure: the optimal solution of the shortest path from the source node s to a given node n can be computed using the optimal solutions from s to each node p that has an edge connecting them to n. Thus, the solution of the problem can be computed using the solution of several sub-problems.
- Overlapping Sub-problems: finding the shortest path from s to n may involve using the shortest path from s to some other node p several times during the computation. Thus, the solutions of the sub-problems may need to be consulted several times. For instance, consider the graph below. The shortest path from s to t can be determined by looking at the shortest path from s to c and from s to d, which, in turn, both require knowledge of the shortest path from s to a and from s to b. The shortest paths from s to a and from s to b need to be consulted more than once!



Exercise 15

a) The airplane routes can be modeled using a directed graph, where each node is a capital city and two nodes are connected if there is a direct flight between them. The weight of the edges corresponds to the travel time.

The problem can then be solved by applying the Floyd-Warshall algorithm to the graph and finding the node that minimizes the distance to its farthest city. For this algorithm, the graph is more conveniently represented using a matrix of adjacencies. Thus, the 2D matrix provided in the input can be used directly.

Pseudo-code:
Input and auxiliary functions:
- adj: 2D adjacency matrix of a graph.
- floydWarshall(adj): runs the Floyd-Warshall algorithm using a graph's adjacency matrix. Returns a 2D matrix with the shortest path distances between all pairs of nodes.

```
find_headquarters(adj)
    dists ← floydWarshall(adj)
    minFarthestDist ← +∞
    bestCity ← NULL
    for (i = 0; i < dists.size(); i++)
        maxDist ← -∞
        for (j = 0; j < dists[i].size(); j++)
            if (dists[i][j] > maxDist)
                maxDist ← dists[i][j]
        if (maxDist < minFarthestDist)
            maxDist ← minFarthestDist
            bestCity ← i
    return bestCity
```

b) The temporal complexity of the Floyd-Warshall algorithm is $\Theta(n^3)$, where n is the number of vertices in the graph, given that all the cells of the n x n dynamic programming matrix are updated over n iterations (i.e. triple nested for cycle over n), and these updates are assumed to take $\Theta(1)$ time. Its spatial complexity is $\Theta(n^2)$ due to the dimensions of the dynamic programming matrix.

After running the Floyd-Warshall algorithm, a double-nested for cycle is used to find the best city, which takes $\Theta(n^2)$ time, given that all cells of the matrix are visited exactly once and processed in $\Theta(1)$ time.

Therefore:

- the temporal complexity of the whole algorithm is $\Theta(n^3)$, as it is dominated by the portion that runs the Floyd-Warshall algorithm.
- the spatial complexity is $\Theta(n^2)$ given that the other variables used in the algorithm are atomic types (int's and double's), so they are dominated by the 2D dynamic programming matrix.

c) Since there are 4 nodes, the outer loop of the Floyd-Warshall algorithm needs to be executed 4 times, thus resulting in the four successive dynamic programming matrices that follow (highlighted in yellow are the values that were updated between iterations).

1st iteration (which considers non-direct flights that pass through the first vertex, i.e. through Bluekistan):

0	2	2.5	6
2	0	1	8
2.5	1	0	5
6	8	5	0

After computing this first table, a route was discovered from Redkistan to Blackistan (and for the reverse route), by making a detour to Bluekistan (which takes 6 hours) and them flying to Blackistan (which takes 2 hours) which results in 6 + 2 = 8 hours.

After the 2nd iteration (which considers non-direct flights that pass through Blackistan), no values of the table are updated, which means that no shorter routes were found.

3rd iteration (which considers non-direct flights that pass through Pinkistan):

0	2	2.5	6
2	0	1	6
2.5	1	0	5
6	6	5	0

A shorter route was discovered between Redkistan and Blackistan.

After the 4th iteration (which considers non-direct flights that pass through Redkistan), no values of the table are updated.

By analyzing the final table (same as the one after the 3rd iteration), the distance from farthest city from each capital can now be computed:

- Farthest city from Bluekistan: Redkistan - 6 hours = min(2, 2.5 ,6)
- Farthest city from Blackistan: Redkistan - 6 hours = min(2, 1, 6)
- Farthest city from Pinkistan: Redkistan - 5 hours = min(2.5, 1, 5)
- Farthest city from Redkistan: Bluekistan and Redkistan - 6 hours = min(6, 6, 5)

Therefore, the city with the minimum distance from all the other cities is Pinkistan, which is where the headquarters should be built.

Exercise 16

To prove this property, induction shall be performed over the natural number k, which denotes the number of iterations currently performed on the outer loop of the Floyd-Warshall algorithm.

Base case: for k = 0 (before any iteration), the dynamic programming matrix is equal to the graph's representation as an adjacency matrix. Since the graph is undirected, by definition, the matrix is symmetric.

Inductive step: let k > 0. Let us assume that, after the first (k-1) iterations, the dynamic programming matrix is symmetric. In other words, the induction hypothesis states that:

$$\forall i, j \in [1, N], d_{ij}^{(k-1)} = d_{ji}^{(k-1)}$$

In the k-th iteration, which considers detours through the k-th node, each cell of the dynamic programming table is updated according to equation 1.

$$d_{ij}^{(k)} = min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \qquad \text{(equation 1)}$$

According to the induction hypothesis, the following equalities hold:

$$d_{ij}^{(k-1)} = d_{ji}^{(k-1)}$$

$$d_{ik}^{(k-1)} = d_{ki}^{(k-1)}$$

$$d_{kj}^{(k-1)} = d_{jk}^{(k-1)}$$

Plugging these new values in equation 1 results in:

$$\forall i, j \in [1, N], d_{ij}^{(k)} = min(d_{ji}^{(k-1)}, d_{ki}^{(k-1)} + d_{jk}^{(k-1)}) = min(d_{ji}^{(k-1)}, d_{jk}^{(k-1)} + d_{ki}^{(k-1)}) = d_{ji}^{(k)}$$

Therefore, proving that the property holds for k.

Conclusion: since the property is true for k = 0 and, for any k > 0, if the property holds for k-1, then it holds for k, then the property is true for any k. Therefore, after applying the Floyd-Warshall algorithm, the resulting dynamic programming table is symmetric.