Prof. Dr. Goran Glavaš,
M.Sc. Fabian David Schmidt
M.Sc. Benedikt Ebing
Lecture Chair XII for Natural Language Processing, Universität Würzburg

1. Exercise for "Algorithmen, KI & Data Science 1"

15.12.2023

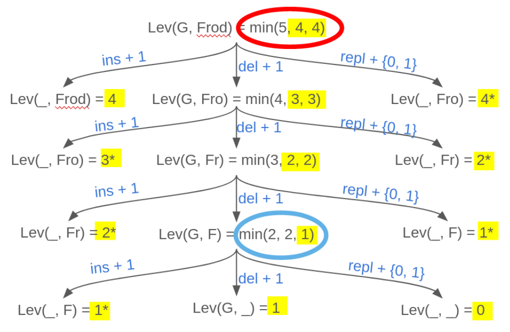
1 Dynamic Programming

1. Recall Merge-sort from lecture 5. Would it be beneficial if we use top-down dynamic programming with memoization in merge-sort? Explain briefly why or why not.

   No, as there are no overlapping subproblems. Therefore, it does not make sense to memoize anything.

2. We want to compute the Levenshtein edit distance between *Frodo* and *Gondor*. Consider the sub-problem of computing the distance between *G* and *Frod*. What are the costs for insertion, deletion and replacement at this stage, respectively.

   The below chart and table highlight the costs associated with insertion, deletion, and replacement, respectively.



* redundant computations

Dynamic programming approach:

	_		F		r		o		d	
–		0	1	1	2	2	3	3	4	4
G		1	1	2	2	3	3	4	4	5
		1	2	1	2	2	3	3	4	4

replacement	insertion
Deletion	minimum

3. Write down the full 6×5 array of distances between all prefixes as shown in the lecture 13. What is the minimum edit distance between Frodo and Gondor?

The below table is not as such required and mostly serves to demonstrate the full solution exploring all viable alternatives at every step.

	_		F		r		o		d		o	
–		0	1	1	2	2	3	3	4	4	5	5
G		1	1	2	2	3	3	4	4	5	5	6
		1	2	1	2	2	3	3	4	4	5	5
o		2	2	2	2	3	2	4	4	5	4	6
		2	3	2	3	2	3	2	3	3	4	4
n		3	3	3	3	3	3	3	3	4	4	5
		3	4	3	4	3	4	3	4	3	4	4
d		4	4	4	4	4	4	4	3	4	4	5
		4	5	4	5	4	5	4	5	3	4	4
o		5	5	5	5	5	4	5	5	4	3	5
		5	6	5	6	5	6	4	5	4	5	3
r		6	6	6	5	6	6	5	5	5	5	4
		6	7	6	7	5	6	5	6	5	6	4

replacement	insertion
Deletion	minimum

Minimum edit-distance(Frodo, Gondor) = Lev(Frodo,Gondor) = 4

4. Implement `levenshtein_distance` in the accompanied Jupyter Notebook using memoization.

5. Consider the following two-player game: There is a row of n objects (assuming n is even) with values $v_1, v_2, \cdots, v_n$. In the game, the two players make moves alternatively. In each odd move, the first player either selects the first or the last object of the row and removes it permanently from the row. In even turns, the opponent plays the game in the same way. Each time the player picks some object,

it receives its value.

Devise a dynamic programming algorithm akin to the four steps as per the lecture to determine the maximum value that the first player (i.e., the player starting the game) can receive by the end of the game.

Let us define $P(i, j)$ to be the maximum value that a player (any player!) can get by playing on successive objects starting from object i and ending with object j, inclusively, by making the first move. Suppose the first player chooses object i with value v_i, and the remaining objects in the row are objects $i+1$ to j. The opponent either chooses object $i+1$ or object j. Note that the opponent is smart enough and always chooses the object which yields the minimum value for the first player. If the opponent takes object $i+1$, the remaining objects are objects $i+2$ to j, on which the first player's maximum value is denoted by $P(i+2, j)$. On the other hand, if the opponent takes object j, the maximum is $P(i+1, j-1)$. Therefore, the maximum amount first player can get when he chooses object i is

$$v_i + \min\{P(i+2, j), P(i+1, j-1)\}$$

Similarly, the maximum amount first player can get if chooses object j is

$$v_j + \min\{P(i+1, j-1), P(i, j-2)\}.$$

Concluding from the above statements, the recurrence relation for an optimal solution is as follows.

$$P(i,j) = \max\{v_i{+}\min\{P(i{+}2,j), P(i{+}1,j{-}1)\}, v_j{+}\min\{P(i{+}1,j{-}1), P(i,j{-}2)\}\}$$

And the base cases are

$$P(i,i) = v_i \text{ and } P(i,i+1) = \max\{v_i, v_i + 1\}$$

Now, let us use this formula for dynamic programming approach, since it can help us avoid computing the same optimal solutions for the subproblems multiple times. To solve the problem, we need to fill a 2D table of size $n \times n$, where entry $[i, j]$ in the table will contain the optimal solution $OPT(i, j)$.

First, we initialize our table with zeros and fill it afterwords using the following strategy. On each iteration k of filling the table, we compute $P(i, j)$ using the aforementioned recurrence relation for all i and j, such that $i - j = k$ and $i < j$. Our result in table is in place $[1, n]$.