Dynamic Programming

Practical Exercises

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

Spring 2025

Exercise 1
Consider the same description for the **change-making problem** as in the TP2 sheet. Unlike in exercise 2 of this sheet, there is a limited amount of coins for each stock. Consider the function *changeMakingDP* below.

```
bool changeMakingDP(unsigned int C[], unsigned int Stock[],
  unsigned int n, unsigned int T, unsigned int usedCoins[])
```

Note: You can assume that the coin denominations are ordered by increasing value in *C*.

a) Implement *changeMakingDP* using a strategy based on dynamic programming.
b) Indicate and justify the algorithm's temporal and spatial complexity, with respect to the number of coin denominations, n, the maximum stock of any of the coins, S, and the desired change amount, T. You can assume that the coin denominations are ordered by increasing value in *Stock*.

Exercise 2
Consider the same description for the **0-1 knapsack problem** as in the TP2 sheet. Implement *knapsackDP* using a strategy based on dynamic programming.

a) Write in mathematical notation the recursive functions *maxValue(i, k)* and *lastItem(i,k)* that return the maximum total value and the index of the last item used in a knapsack with maximum capacity k ($0 \leq \leq k \leq$ *maxWeight*) using only the first i items ($0 \leq i \leq n$, where n is the number of the different items available). Use a symbol or special value if a function is not defined.
b) Calculate the table of values for *maxValue(i, k)* and *lastItem(i,k)* for the input example below:

Input: `values` = [10, 7, 11, 15], `weights` = [1, 2, 1, 3], `n` = 4, `maxWeight` = 5
Expected result: [1, 0, 1, 1] (the total value is 10 + 11 + 15 = 36)

c) Implement *knapsackDP*, which uses a dynamic programming algorithm to solve the problem.

```
unsigned int knapsackDP(unsigned int values[], unsigned int weights[],
    unsigned int n, unsigned int maxWeight, bool usedItems[])
```

d) Indicate and justify the algorithm's temporal and spatial complexity, with respect to the number of items, n, and the knapsack's maximum capacity, T.

Exercise 3

The **edit/Levenshtein distance** between two strings is defined as the minimum number of operations to convert a string A to a string B. Three operations are possible to perform this conversion:

- Insertion: Adding a character in any position of the string.
- Substitution: Swapping a character in any position of the string with any other character.
- Deletion: Removing a character in any position of the string.

Let D(i,j) be the edit distance between the first (i + 1) characters of a string A, A[0:i], and the first (j + 1) characters of a string B, B[0:j] (to convert A[0:i] to B[0:j]).

a) Indicate the recursive formula for the edit/Levenshtein distance, D(i,j).
b) Compute the edit distance between "money" and "note" (i.e. to convert "money" to "note").

After searching for documents which include words similar to a given one, there is the need to sort them by relevance (edit distance). Consider the function *numApproximateStringMatching*, which returns the average edit distance of words in the file (named *filename*) to the searched for expression (*toSearch*). The average distance is computed by the formula (sum of distances / number of words).

```
float numApproximateStringMatching(std::string filename,
    std::string toSearch)
```

c) Implement the auxiliary function *editDistance*, which computes the edit distance between two words, using dynamic programming.

```
int editDistance(std::string pattern, std::string text)
```

d) Indicate the temporal and spatial complexities of *editDistance* with respect to the lengths of strings A and B (|A| and |B|, respectively). Justify your answers.
e) Using the *editDistance* function, implement *numApproximateStringMatching*.

Exercise 4

Given a **context-free grammar (CFG)**, parsing consists of determining if a given sequence of terminal symbols can be derived from the start symbol of the grammar through a series of derivations, i.e., the application of productions of the grammar that replace one non-terminal symbol with zero or more sequence of terminal and non-terminal symbols.

In this exercise you are to explore the use of a dynamic programming algorithm, due to Cocke, Young and Kasami, namely the **CYK algorithm**, that determines if a given input string (or sequence of tokens) belongs to the language generated by a CFG. The CYK algorithm, however, requires that the CFG be in the so-called **Chomsky Normal Form (CNF)** where a production of the grammar can have no more than two non-terminal symbols and a single terminal symbol. Note that translation from a generic CFG to CNF can be automated, so the CNF grammar restriction of the CYK algorithm does not imply a loss of its parsing capability.

For example consider the CFG below with starting symbol S, non-terminals {S, A, B, C}, terminals {a,b} and the various productions below:

```
S → AB | BC
A → BA | a
B → CC | b
C → AB | a
```

To show that the input string "baaba" belongs to L(G), the algorithm needs to show that it is possible to derive this string starting from S.

The CYK algorithm rests on the observation that for a string of length n to be generated by the CNF grammar, one explores all the possibilities to generate strings of length n, i.e., that the first symbol is at index 1 and has length n. This suggests a recurrence, as for generating such strings, one must find a production that can split this string in two strings, one starting at index 1 with length 1 (thus ending at index 2) combined with a string that starts at index 2 and with length n-1. Alternatively, one can also split a string that starts at index 1 of length n, as two strings, one starting at 1 with length 2 with a second string starting at index 3 with length n-2. In general, the recurrence for this CYK algorithm can be described as shown below where each symbol $S_{i,k}$ corresponds to sets of productions p of the grammar that can derive the input string that starts at index i with length k (thus ending at index i+k).

$$S_{i,j} = \{A \rightarrow BC \in P \mid B \in S_{i,k} \text{ and } C \in S_{i+k,j-k} \text{ for k in } [1,j]\}$$

with

$$S_{i,1} = \{A \rightarrow a \in P\} \text{ if string at index i has character a.}$$

In terms of its implementation, the algorithm works "backwards" from smaller strings to increasingly large strings. The implementation is based on a table, where the algorithm starts by computing a first row that corresponds to all the ways to generate single terminal symbols that match the input string and thus can be viewed as string of length 1 starting at each index. Then, a second row is derived by combining two substrings of the previous row and thus starting at a given index and having lengths

of size 2. The process is repeated working towards a single entry at the top of this (triangular) table, which will correspond to the generation of the string starting at index 1 with length n. The table is L-shaped as the lower section of a squared matrix.

Given this description, derive the parsing table for the above grammar and the given input string, indicating if this grammar is ambiguous. Recall that a grammar is ambiguous, if it can generate one sentence with more than one different derivation. In terms of the algorithm, this means that there is one or more cells in the table with more than one grammar production.