Dynamic Programming

Practical Exercises

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

Spring 2024

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*.