Dynamic Programming

Practical Exercises

*Gonçalo Leão**

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

Spring 2023

**Various of these problems were originally developed by the team of instructor of the 2021 CAL (Concepção de Algoritmos") course of the L.EIC at the Faculty of Engineering of the University of Porto (FEUP), University of Porto and adapted and extended here by the author.*

A - Fundamentals

Exercise 1

The factorial of an integer *n* (*n!,* with $n \geq 0$) is the product of all the integers between 1 and *n*.

a) Write the recurrence formula for n!.
b) Implement and test a function that calculates the result in a recursive method (*factorialRecurs* function).
c) Implement and test a function that calculates the result in an iterative method with dynamic programming (*factorialDP* function).
d) Compare the two previous algorithms in terms of their time and space complexity, with respect to n.

Exercise 2

Consider the **variant of the change-making problem with unlimited stock**, where one wants to produce *m* cents of change with the least amount of coins. In this variant, there is an unlimited supply of coins of each value for the change calculation.

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

Input example: **C** = [1, 2, 5, 10], **n** = 4, **T** = 8
Expected result: [1, 1, 1, 0]

Input example: **C** = [1, 2, 5, 10], **n** = 4, **T** = 38
Expected result: [1, 1, 1, 3]

a) Write in mathematical notation the recursive functions *minCoins(i, k)* and *lastCoin(i,k)* that return the minimum amount of coins and the value of the last coin used to produce k value of change ($0 \leq k \leq m$) using only the first i coins ($0 \leq i \leq n$, where n is the number of the different coins available). Use a symbol or special value if a function is not defined.
b) Calculate the table of values for *minCoins(i, k)* and *lastCoin(i,k)* for the input example of the 8-cent change.
c) Implement *changeMakingUnlimitedDP*, which uses a dynamic programming algorithm to solve the problem.

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

Suggestion: The values of *minCoins* and *lastCoin* should be calculated for the increasing values of *i* and *k* (as *arrays*), memoizing only values for the last *i* value (one single dimension array per function).

d) Indicate and justify the algorithm's temporal and spatial complexity, with respect to the number of coin denominations, n, and the desired change amount, T.

Exercise 3

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*.

Note: this exercise is more challenging than exercise 2.

Exercise 4

Consider the *calcSum* function below, which given a sequence of *n* integers (*n* > 0), returns a string with m parts. The m-th part (0 ≤ *m* < *n*) has the format *s,i;* where *i* represents the index in the sequence of the first value of the subsequence of *m + 1* contiguous elements whose sum (*s*) is the smallest possible (0 ≤ *i* < *n-m*).

```
std::string calcSum(int sequence[], unsigned long n)
```

Input example: **sequence** = [4, 7, 2, 8, 1], **n**=4
Expected result: "1,4;9,1;11,2;18,1;22,0;"

In this example, we have the following smallest sum contiguous subarrays:

Subarray of size 1 (*m* = 0): [1], where *s* = 1, *i* = 4

Subarray of size 2 (*m* = 1): [7, 2], where *s* = 9, *i* = 1 *(if there is more than one solution, the first is returned)*

Subarray of size 3 (*m* = 2): [2, 8, 1], where *s* = 11, *i* = 2

Subarray of size 4 (*m* = 3): [7, 2, 8, 1], where *s* = 18, *i* = 1

Subarray of size 5 (*m* = 4): [4, 7, 2, 8, 1], where *s* = 22, *i* = 0

a) Implement *calcSum* using dynamic programming.
b) Indicate and justify the temporal complexity of the algorithm, in the worst case, with respect to n.
c) Using the provided *testPerformanceCalcSum* function, produce a graph with the average execution times of the algorithm for the increasing values of *n* 500, 1000, …, 10000 (*testPerformanceCalcSum* function). For each value of n, generate 1000 random sequences of integers between 1 and 10 x *n* (repetitions allowed) and measure the average elapsed time. Suggestion: generate a CSV format file and generate a graph using Excel or similar tool; consider the code methods below to measure the elapsed time in milliseconds (ms).

Exercise 5

The number of ways of dividing a set of n elements into k non-empty disjoint subsets $(1 \leq k \leq n)$ is given by the Stirling number of the Second Kind, $S(n,k)$, which can be calculated with the recurrence relation formula:

$$S(n,k) = S(n-1,k-1) + k\,S(n-1,k), \text{ se } 1 < k < n$$
$$S(n,k) = 1, \text{ se } k=1 \text{ ou } k=n$$

On the other hand, the total number of ways of dividing a set of n elements $(n \geq 0)$ in non-empty disjoint subsets is given by the n^{th} Bell number, denoted $B(n)$, which can be calculated with the formula:

$$B(n) = \sum_{k=1}^{n} S(n,k)$$

For example, the set {a, b, c} may be divided 5 different ways:

{a, b, c}
{a, b}, {c}
{a, c}, {b}
{b, c}, {a}
{a}, {b}, {c}

In this case we have $B(3) = S(3,1) + S(3,2) + S(3,3) = 1 + (S(2,1)+2S(2,2)) + 1 = 1 + 3 + 1 = 5$.

B(n) grows quickly. For example, B(15) =1382958545.

a) Implement the $S(n,k)$ and $B(n)$ functions using a top-down recursive approach, considering their definitions (*s_recursive* and *b_recursive* functions).
b) Implement the $S(n,k)$ and $B(n)$ functions using an iterative method with dynamic programming, based on the ${}^{n}C_{k}$ calculation method presented in the theoretic classes (*s_dynamic* and *b_dynamic* functions).
c) What are the temporal complexities of the four algorithms?
d) What are the spatial complexities of the four algorithms?

Exercise 6

Recall the TP2 sheet, when we solved the **maximum subarray problem** using brute-force and only managed to get a $O(n^3)$ time complexity. Then, in the TP4 sheet, we solved the problem with a divide-and-conquer solution, achieving a O(n*log(n)) complexity. Consider the function *maxSubsequence* below.

```
int maxSubsequenceDP(int A[], unsigned int n , int &i, int &j)
```

a) Implement *maxSubsequenceDP* using dynamic programming in order to solve this problem in linear time, O(n).
b) Using the provided *testPerformanceMaxSubsequence* function, compare the average execution times of the dynamic programming, brute-force (TP2) and divide-and-conquer algorithms (TP4) for the increasing values of *n*, in a similar fashion to exercise 5. Compare the results. Are they consistent with the respective temporal complexities of each solution?

Exercise 7

Consider the same description for the **Hanoi towers problem** as in the TP4 sheet. Consider the function *hanoiDP* below.

```
std::string hanoiDP(unsigned int n, char src, char dest)
```

a) Explain why it makes more sense to solve the Hanoi towers problem with dynamic programming than with a divide-and-conquer approach.
b) Implement *changeMakingDP* using a strategy based on dynamic programming.

Exercise 8

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 ($0 \leq k \leq maxWeight$) 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 9

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*.

B - Applications to the shortest-path problem

Exercise 10

To solve this and the following exercises, use the **DPGraph** class provided in the *exercises.h* file. **DPGraph** derives from the **Graph** base class (declared in *data_structures/Graph.h*), which implements a graph's basic functionalities and includes all the necessary attributes for the **Vertex**, **Edge** and **Graph** classes to solve these exercises.
All exercises can be solved without modifying the **Graph**, **Vertex** and **Edge** classes and by only adding auxiliary methods to the **DPGraph** class. No class attribute needs to be created on any class.

a) Using the **DPGraph** class provided in the *exercisses.h* file (which derives from the **Graph** base class), implement *dijkstra*, which uses the Dijkstra algorithm to find the shortest path from a node to all other nodes in a directed graph.

```
std::vector<Vertex *> dijkstra(const int &origin)
```

Suggestions:

- use the provided **MutablePriorityQueue** class.
- use the *dist* and *path* attributes (and associated getters and setters) from the **Vertex** class.
- implement the private auxiliary method *relax* in the **DPGraph** class. This method performs relaxation along an edge with weight equal to *weight* between vertices *v* and *w* and returns a boolean indicating if the relaxation actually occurred.

```
bool relax(Vertex *v, Vertex *w, double weight)
```

b) Implement the *getPath* public method in the **Graph** class.

```
vector<int> getPath(const int &origin, const int &dest)
```

Considering that the *path* property of the graph's vertices has been updated by invoking a shortest path algorithm from one vertex *origin* to all others, this function returns a vector with the sequence of the vertices of the path, from the *origin* to *dest*, inclusively (*dest* is the identifier of the destination vertex of the path). It is assumed that a path calculation function, such as *dijkstra*, was previously called with the *origin* argument, which is the origin vertex.

Exercise 11

Using the **DPGraph** class provided in the *exercises.h* file, implement *bellmanFord*, which uses the Bellman-Ford algorithm to find the shortest path from a node to all other nodes in a directed graph that can have edges of negative weight.

```
void bellmanFord(const int &origin)
```

Suggestions:

- use the *relax* auxiliary method from exercise 10.
- use the *dist* and *path* attributes (and associated getters and setters) from the **Vertex** class.

Exercise 12

a) Using the **DPGraph** class provided in the *exercises.h* file, implement *floydWarshall*, which uses the Floyd-Warshall algorithm to find find the shortest path between all pairs of nodes in a directed graph.

```
void floydWarshall()
```

Suggestion: Use the *distMatrix* and *pathMatrix* attributes from the **Graph** class.

b) Implement the *getFloydWarshallPath* public method of the **DPGraph** class, which returns a vector with the sequence of elements in the graph in the path from *origin* to *dest* (where *origin* and *dest* are the values of the identifiers of the origin and destination vertices, respectively).

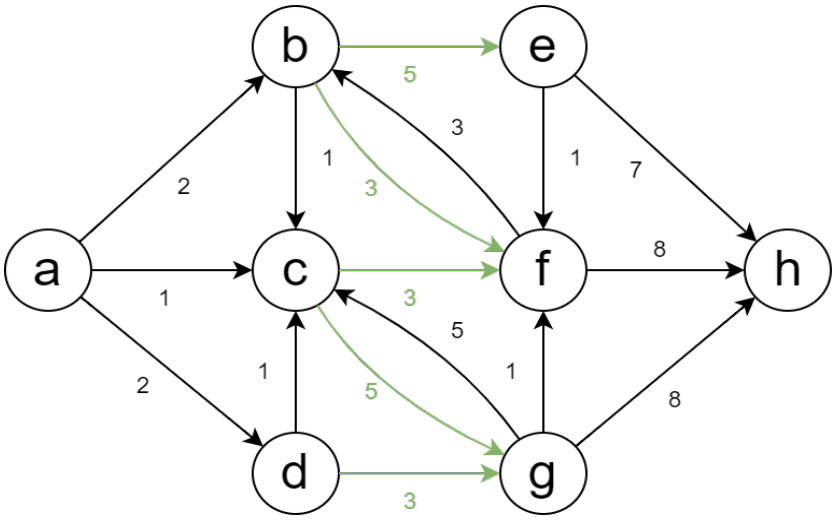
```
vector<int> getFloydWarshallPath(const int &origin, const int &dest)
```

Exercise 13

A small town is modeled as a set of intersections and roads in a small town. Two vehicles are parked in node a want to travel to node h while minimizing energy consumption. One of the vehicles (vehicle A) runs on fuel while the other one (vehicle B) is hybrid. Vehicle B is able to produce energy (rather than consume) when traveling downwards.

Consider the graph below where the nodes represent the town's intersections and the edges, the town's roads. The edges' weight reflects the energy needed to traverse the road (in MJ) according to the following rules:

- for vehicle A: the energy consumption in each edge corresponds to the weights shown in the graph.
- for vehicle B: the energy consumption in the black edges corresponds to the edges' weights, but for the green edges the weights indicate the energy that was produced.



a) Determine the shortest path from node a to h for vehicle A. Detail all the steps of the computation.
b) Determine the shortest path from node a to h for vehicle B. Detail all the steps of the computation.
c) Would it make sense to have a cycle in the graph that includes the green edges? Explain.

Exercise 14

Which algorithm design techniques (brute-force, greedy, divide-and-conquer, dynamic programming, backtracking) does the Dijkstra algorithm employ? Justify your answer.

Exercise 15

A group of countries decided to form an allegiance. The headquarters of the annual meeting of the allegiance must be located in the capital city of one of the forming countries. The leaders agreed to to build the headquarters in the city that minimized the travel time by airplane to the farthest city away.

You have access to the travel time by airplane between all pairs of capital cities. These times are represented as a 2D matrix and are measured in hours. Some cities may not be accessible directly by airplane, requiring other capital cities to be visited first. Example:

from/to	Bluekistan	Blackistan	Pinkistan	Redkistan
Bluekistan	0	2	2.5	6
Blackistan	2	0	1	x
Pinkistan	2.5	1	0	5
Redkistan	6	x	5	0

a) Write in pseudo-code (or in C++) an efficient algorithm to solve this problem.
b) Indicate the temporal and spatial complexity of the algorithm. Justify your answers.
c) Apply your algorithm to the specific case presented above while detailing all the algorithm's steps.

Exercise 16

Prove by induction that, in an undirected weight graph, after applying the Floyd-Warshall algorithm, the resulting 2D matrix of shortest path distances between nodes is symmetric (i.e. that all the shortest path distance between any pair of nodes (i, j) is the same as the distance for the pair (j, i)).