

3rd Lab class – Part A – Dynamic programming (Programação Dinâmica)

Instructions

- Download the zipped file **DA_TP03a_unsolved.zip** from the course's Moodle area and unzip it. It contains the folder **TP3a**, with the *.h* and the *.cpp* files needed for this lab class. The *exercises.h* file specifies the prototypes of the functions to be implemented. Implement your solutions in the respective *.cpp* file of each exercise.
- Copy the whole folder to the root of the CLion project provided in the first lab class.
- Edit the *CMakeLists.txt* in the root of the project, so as to set up an executable run/debug configuration for **TP3a**, accordingly:
 - Add the source files for the **TP3a** class:

```
file(GLOB TP3a_FILES CONFIGURE_DEPENDS "TP3a/*.cpp")
```
 - Add executable target with source files listed in *TP2a_FILES* variable:

```
add_executable(TP3a main.cpp ${TP3a_FILES})
```
 - Link the Google Test library to the *TP2a* target:

```
target_link_libraries(TP3a gtest_main gmock_main)
```
- Do “Load CMake Project” over the file *CMakeLists.txt*. The **TP3a** configuration should now appear and be available from the “Select Run/Debug Configuration” drop-down list, on the upper-right corner of the CLion IDE.
- Compile and run the project after selecting the **TP3a** configuration.

Exercises

1. Factorial

Consider that you want to calculate the factorial of a given integer n (≥ 0).

- Implement and test a function that calculates the result in a recursive method (**factorialRecurs** function).
- Implement and test a function that calculates the result in a iterative method with dynamic programming (**factorialDP** function).
- Compare the two previous approaches (line a and b) in terms of their time and space complexity.

2. Coin change with unlimited stock

Consider you want to produce m cents of change with the least amount of coins. The value of the coins available to you (e.g. 1, 2 and 5 cents) are passed as a function parameter. Consider an unlimited supply

of coins of each value for the change calculation. For example, to produce 9 cents of change, the expected result is two coins of 2 cents and one coin of 5 cents.

- Formulate this exercise as a linear programming problem. (Suggestion: see the bag problem in the slides from the theory classes)
- 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.
- Calculate the table of values for $minCoins(i, k)$ and $lastCoin(i, k)$ for the previously mentioned “9 cent change with 1, 2 and 5 cents coins” example.
- Implement an algorithm using dynamic programming that determines the optimal solution for a given m , returning the coins used to achieve this (**changeMakingUnlimitedDP** function). 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).

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

For example: $C = [1, 2, 5, 10]$, $n=4$, $T = 8$

Result: [1, 1, 1, 0]

For example: $C = [1, 2, 5, 10]$, $n=4$, $T = 38$

Result: [1, 1, 1, 3]

3. Coin change with limited stock

Consider the same description for the change-making problem of exercise 3 of the first practical class, where, unlike the previous problem, there is a limited amount of coins for each stock. Implement the function *changeMakingDP* below using a dynamic programming strategy instead. You can adapt your solution for the previous problem by modifying the auxiliary data structures.

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

For example: $C = [1, 2, 5, 10]$, **Stock** = [3, 5, 2, 1], $n=4$, $T = 8$

Result: [1, 1, 1, 0]

For example: $C = [1, 2, 5, 10]$, **Stock** = [1, 2, 4, 2], $n=4$, $T = 38$

Result: [1, 1, 3, 2]

4. Smallest sum contiguous subarrays

Given a sequence of n integers ($n > 0$), for each value of m from 1 to n , determine i that represents the index in the sequence of the first value of the subsequence of m contiguous elements whose sum (s) is the smallest possible ($0 \leq i \leq n-m$).

For example, in the sequence (4,7,2,8,1), where $n = 5$, we have the following smallest sum contiguous subarrays:

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

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

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

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

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

- Produce an algorithm using dynamic programming, that determines the optimal solution for each m , returning i and s for each case (**calcSum** function). Test using the example above.
- 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 $10 \times n$ (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 (μs).

```
#include <chrono>
...
auto start = std::chrono::high_resolution_clock::now();
...
auto finish = std::chrono::high_resolution_clock::now();
auto mili = chrono::duration_cast<chrono::milliseconds>(finish - start).count();
```

5. Partitioning a set

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, $B(15) = 1382958545$.

- Implement the $S(n,k)$ and $B(n)$ functions using a recursive approach, considering their definitions (**s_recursive** and **b_recursive** functions).
- Implement the $S(n,k)$ and $B(n)$ functions using dynamic programming, based on the nC_k calculation method presented in the theory classes (**s_dynamic** and **b_dynamic** functions). What is the temporal and spatial efficiency of the resulting solution?
- Compare the performance of the recursive and dynamic programming versions.

6. The maximum subarray problem

Recall the first TP class when we solved the maximum subarray problem using brute-force... We managed to get a $O(n^3)$ time complexity. Then, on the third TP class, we solved the problem with a divide-and-conquer solution, achieving a $O(n \log(n))$ complexity.

Given any one-dimensional array $A[1..n]$ of integers, the **maximum sum subarray problem** tries to find a contiguous subarray of A , starting with element i and ending with element j , with the largest sum: $\max \sum_{x=i}^j A[x]$, with $1 \leq i \leq j \leq n$. Implement the function *maxSubsequence* below.

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

The function returns the sum of the maximum subarray, for which i and j are the indices of the first and last elements of this subsequence (respectively), starting at 0.

For example: $A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$

Solution: $[0, 0, 0, 1, 1, 1, 1, 0, 0]$, as subsequence $[4, -1, 2, 1]$ ($i = 3, j = 6$) produces the largest sum, 6.

- Implement the function using dynamic programming in order to solve this problem in linear time, $O(n)$.
- Produce a graph with the average execution times of the dynamic programming, brute-force (TP1) and divide-and-conquer algorithms (TP3) for the increasing values of n , in a similar fashion to exercise 4b. Compare the results. Are they consistent with the respective temporal complexities of each solution?