1st Lab class – Dynamic programming

Instructions

- Download the zipped file **cal_fp01_CLion.zip** from the course's Moodle área and unzip it (it contains the folder **lib,** the folder **Tests** with files **Factorial.h**, **Change.h**, **Sum.h**, **Partitioning.h**, and the files **CMakeLists** and **main.cpp**)
- In the CLion IDE, open a project, selecting the folder that contains the files mentioned in the previous bullet point.
- Do "*Load CMake Project*" over the file *CMakeLists.txt*
- Run the project (**Run**)
- Please note that the unity tests of this project may be commented. If this is the case, uncomment the tests as you make progress in the implementation of the respective exercises.
- You should implement the exercises following the order suggested.
- Implement your solutions in the respective .cpp file of each .h .

Exercises

1. Factorial *(Factorial.h)*

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

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

2. Change *(Change.h)*

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.

a. Formulate this exercise as a linear programming problem. (Suggestion: see the bag problem in the slides from the theory classes)
b. 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.

c. 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.
d. Implement an algorithm using dynamic programming that determines the optimal solution for a given *m*, returning the coins used to achieve this (**`calcChange`** 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).

3. Smallest sum contiguous subarrays *(Sum.h)*

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 ($0 \leq i \leq n\text{-}m$).

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 $s = 22$, $i = 0$

a. 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.
b. Produce a graph with the average execution times of the algorithm for the increasing values of *n* 10, 20, …, 500 (**`testPerformanceCalcSum`** function). For each value of n, generate 1000 random sequences of integers between 1 and 10 x *n* (repetitions allowed) and measure the 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();
```

4. Partitioning a set *(Partitioning.h)*

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:

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

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 nth 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)*+2*S(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 recursive approach, considering their definitions (**`s_recursive`** and **`b_recursive`** functions).
b. Implement the *S(n,k)* and *B(n)* functions using dynamic programming, based on the ${}^{n}C_{k}$ calculation method presented in the theory classes (**`s_dynamic`** and **`b_dynamic`** functions). What is the temporal and spatial efficiency of the resulting solution?
c. Compare the performance of the recursive and dynamic programming versions using *gnu profiler* (see instructions in the slides of the theory classes).