INFO0054 - Functional Programming

Project: Nonograms

Introduction

A nonogram is a popular Japanese puzzle game that combines elements of logic and pixel art. The goal of a nonogram is to reveal a hidden picture by filling in a grid of squares based on numerical hints provided for each row and column. The numbers indicate how many consecutive squares should be filled in, separated by at least one empty square.

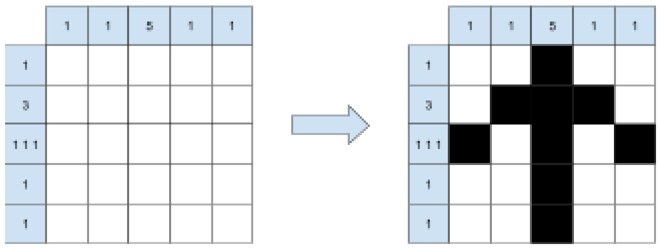
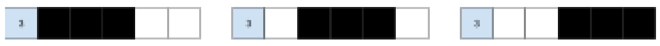


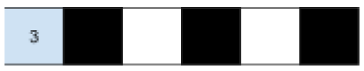
Figure 1: Example of a nonogram revealing an arrow.

A (classical) nonogram starts with a rectangular grid. Along the top and left sides of the grid, there are collections of numbers that provide **hints**. These numbers indicate how many consecutive squares in that row or column should be filled in with the picture, and they are separated by at least one empty square. For example, if a row has the clue "3 2," then there are three consecutive filled squares followed by at least one empty square and then two more consecutive filled squares in that row.

For example, there are three possibilities to fill a row of length 5 with the hint “3”:



And there is only one possibility to fill a row of length 5 with the hint “1 1 1”:



You use the hints to logically deduce which squares should be filled in and left empty. By analyzing the clues and cross-referencing them with the adjacent rows and columns, you can gradually reveal the hidden picture. Computer scientists are interested in nonograms as conceiving a nonogram solver poses an interesting challenge. Brute-force solutions are inefficient, as the problem space can grow exponentially large!

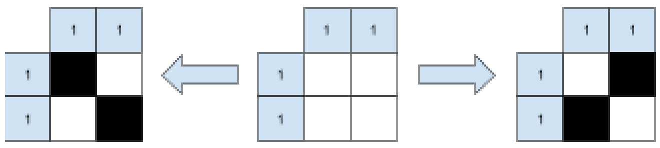
The goal of this project is to create a nonogram solver. You do not have to be (overtly) concerned with the efficiency of your algorithm; a brute-force approach starting from the rows suffices to obtain a good grade for the purpose of this course. Students who develop a more efficient nonogram solver will be recognized for their efforts, provided they respect the course’s requirements (e.g., developing pure functions). Students will notice that many of the efficient implementations rely on side effects.

Requirements

Requirement	Weight
Create ADTs (Algebraic Data Types). You will have ADTs for squares (blank, filled in, etc.), grids, and solutions. Document and motivate your choices.	10%
Write a function **hints** that takes as input a solution (i.e., a filled-in grid) and returns two lists of hints, one for the rows and one for the columns. Ensure that you have implemented the function as efficiently as possible. Some auxiliary functions developed for hints may be useful for the nonogram solver.	20%
Write a function **solve** that takes as input two lists of hints, the first representing the hints for each row and the second representing the hints for each column. The function solve returns a list of solutions *(or an ADT representing whether solve has found solutions)*. You may assume that the input is well-formed: all numbers are strictly positive. As stated before, you are not required to develop a solver that is efficient; a brute-force approach suffices. That said, generating all possible solutions prior to testing the hints will not scale at all, so you must start from the hints in some way, shape, or form. Your implementation will depend on auxiliary functions that break down the problem. Ensure that each of these functions is defined as efficiently as possible. When relying on Scala’s standard library, ensure you know its efficiency (e.g., foldLeft vs. foldRight) and discuss this in the report. You will likely need to define functions to generate a list of rows from hints, combine rows, compute hints, etc. Any code you have found or reused must be adequately attributed to the code and the report.	30%

Efficiency of your solver. Have you (managed) to develop a method that is (slightly) more efficient than a brute-force approach? Explain how and why your approach is more efficient. You can also illustrate and demonstrate this (e.g., via benchmarking).	20%
Implementation and documentation. - You will develop a module for the project and use that module in a Scala script or Scala program, demonstrating how your module will be used. In other words, there will be at least two files. You will also provide a README detailing how to compile, run, etc., your code, and end examples. - Your code has been clearly documented, and you have provided a specification for each function you have defined. A specification describes the expected behavior without specifying how it is implemented. - Provide a method **draw** that "pretty prints" solutions to the console. That function is not pure as it outputs something to the console, but it must rely on something functional.	10%
Report. You will submit, together with your code, a **technical report**. Your report may be written in French. In this report, you will briefly describe and motivate your ADTs and the choices for implementing your functions. You are also asked to reflect critically on your work.	10%
Bonus. For "bonus" points, you can create a parser for nonograms. The parser should take as input a string that represents both hints. E.g., `"[[[1],[3],[1,1,1],[1],[1]], [[1],[1],[5],[1],[1]]]"` This part of the project will help you exercise exception handling in Scala if you have not used it in the project.	10%

If you stick to a brute-force approach, you will notice that your implementation will be slow, and you will soon run out of memory. So, limit your examples to grids that are 5 by 5. Remember that a nonogram may yield more than one solution. A typical example of such a nonogram is



Nonograms are well-studied in computer science, and you will find many resources in literature and on the Web. You must cite any resources you used to realize this project in your report. A mere list of references does not suffice; references must be used in the text.

Deducing Partial Solutions

One way of optimizing the algorithm is by looking for partial solutions (e.g., by deducing the values of a square or by pruning partial solutions that do not satisfy requirements). You can thus significantly reduce the problem space (and thus the number of objects in memory) by such deductions. You are encouraged to tackle this problem. A good approach is to first develop a naïve implementation in one module and then develop an optimized version in another.

The two modules can be used for some benchmarking (in time and space). If you wish to compare the behavior of the two modules (or only one module), then you are allowed to provide the code for the benchmark in a separate file.

Modalities

The project will be carried out in groups of three students and must be submitted by December 1, 2023, at 11:59 p.m. at the latest. Your code must be submitted as a ZIP file and your report as a PDF. Ensure that you submit the two separately. All submissions will be made through eCampus.

To anticipate any problems related to the formation of groups, you must constitute these on the submission platform by October 19, 2023. After this date, it will no longer be possible to form a group and, therefore, to submit the project.

All students must fill in a peer assessment form after the deadline. Filling in the form will be a prerequisite to obtaining a grade. Individual grades will be adjusted using these peer assessments, which can deviate at most 2 points from the group's grade (unless a student clearly underperforms or outperforms).

Participation in the project is mandatory. Students who do not submit anything for the project, which includes a blank report and/or barren source code, will receive an absence grade (A) for the course.

As a reminder, plagiarism is severely punished. I also remind students to consult the *"Charte d'utilisasion des outils d'intelligence artificielle par l'étudiant"* (french) or *"Charter for the use of artificial intelligence tools by students"* (English) You are encouraged to discuss ideas and approaches with your peers, but you are not allowed to share your code.