# Advanced software engineering reflective essay

# Group 3

## Introduction

This document will be a critical reflection on hour our group performed in the set tasks for advanced software engineering. Where we exceled, where we had room to improve and what we would do differently if we were to complete these tasks again.

## Tools used

During these projects the group made use of variety of tools and techniques. From the start of the first project, we made good use of AGILE, setting up the tasks that needed to be completed on a Kanban board hosted on our GitHub repository. Joe acted as our scrum manager and organised who would be assigned to which tasks based on who had the most relevant experience. We checked in regularly with each other on a Discord server. In which we held virtual stand-up meetings to make sure everyone was on task and if they were stuck in any way, getting the support that they need. We used the weekly advanced software engineering lab sessions for our scrum meetings which we found very helpful for solving problems collaboratively.

We made good use of paired programming in this project. Members of our group worked together to come up with solutions and speed up the process of writing code. We also used GitHub co-pilot to help write some more basic code snippets and figure out how to write some functions that we were unsure how to realise.

Our codebase in the end mainly consisted of Typescript and JavaScript. We developed our first submission in JavaScript as it was a language, we were familiar with and could develop solutions in quickly. It was also well suited to creating web applications as it could be easily hooked into html. We changed to writing our programs in typescript when we started using Angular to develop the front end of the application as it is a typescript-based framework.

The front end of our code that turned our solutions into a fully-fledged web application was created using the Angular framework although our first submission (N Queens) was displayed just using HTML and CSS.

Git hub was used for many different functions in this project. It was used for version control, hosting the sprint boards and for developing and hosting the web applications in GitHub code space. And running automatic tests.

# N queens

The first task was the to solve the N-queens completion problem. A solution to the n Queens puzzle necessitates that no two chess queens occupy the same row, column, or diagonal in order to ensure that no two queens pose a threat to one another.

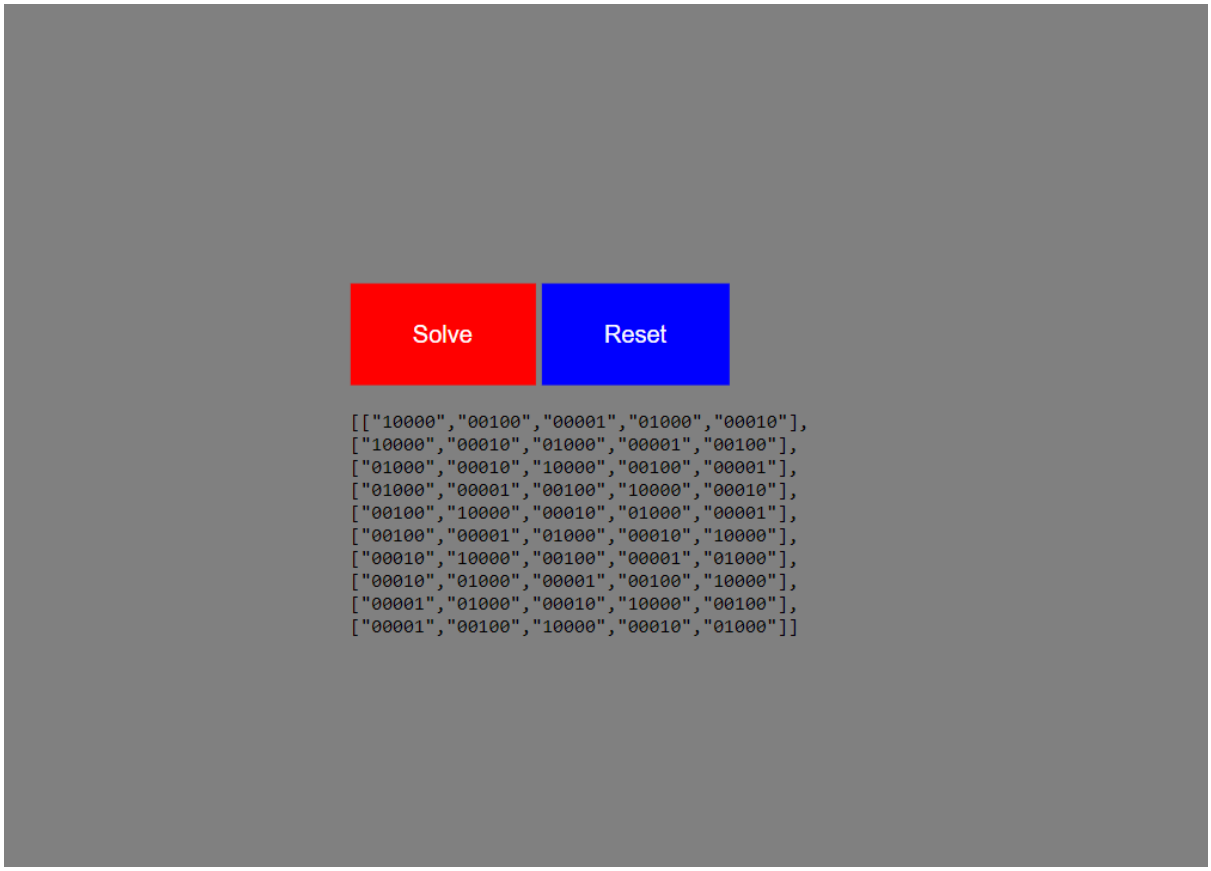
Our n queens solution consisted of JavaScript that was displayed on a html page and styled with CSS. The front end was not as professional as our later submissions. This is largely due to none of our then active group members having experience working on the front end of web applications.

The first way that we tried to solve the n Queens problem was by brute force. We wrote code that defined the rules on where a queen could be placed and then had it randomly place queens in legal spaces. If it can place n Queens, then we know that a solution has been found and if it's different to any existing solutions we have found then we can add it to our existing solutions.

We quickly realised that this was not a good enough solution. As it would frequently fail while building a solution and could not recurse one step back it would have to start a solve all the way from the beginning if it placed one piece in a bad position that stopped a solution. It also wasted time accidently redoing existing solutions.

We improved the code by adding recursion to the solution to allow it to step back one move at a time instead of starting over. We also had it work systematically instead of randomly placing a queen in a legal spot. This improved the speed of our code by a significant margin, allowing us to find all solutions in just a few seconds. The problem with this code was its ability to scale. While our solution worked great when n was a relatively low number it would soon start to take a long time to find all the solutions when n increases above around 20.

After learning more about exact cover problems while completing the other tasks we now see that applying algorithm x with dancing links to n queens would have led to much better performance and allowed us to scale the application to accept very high values of n. If we were to do this project again then we would design the back end to use dancing links and upgrade the front end using a proper web app framework like Angular.




Above our webapp that solved n Queens with an output where n is 5 and no queens were preplaced. While it outputs the solution correctly it doesn't look very good aesthetically and it can be quite hard to read the solutions as they are output in a line rather than as individual boards.

## Pentomino puzzle

The second task was to solve the Pentomino puzzle. The puzzle is an 5x11 rectangular board where 12 individual pentominoes may fit in. The application when run presents all of the different possible placements of these pentominoes on the board.

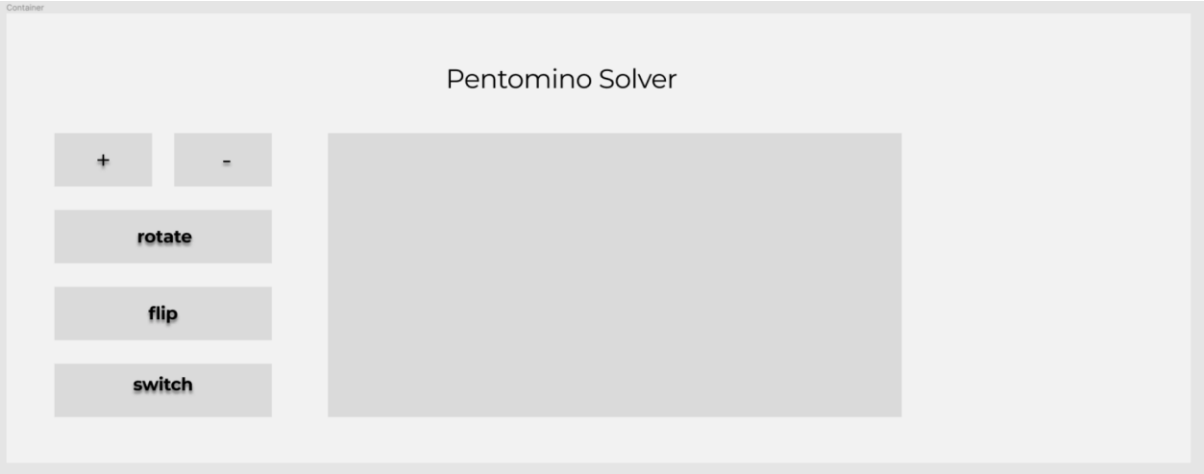
Before starting this project, we spend a few hours researching how best to solve this problem. This is when we learned that this could be represented as an exact cover problem and could be solved using Knuth's algorithm x. we also learned that the best implementation of this algorithm was dancing links. We decided that we should try and create the best possible solution to this project and tried to implement dancing links.

Unfortunately, at this stage, while we had a solid grasp of algorithm x, we did not understand how to fully implement algorithm x. we didn't understand how to generate a circular linked list on the scale we needed to solve algorithm x. We spent a lot of time trying to figure it out but in just over a week before the deadline finally realising that we wouldn't be able to solve it in that time.

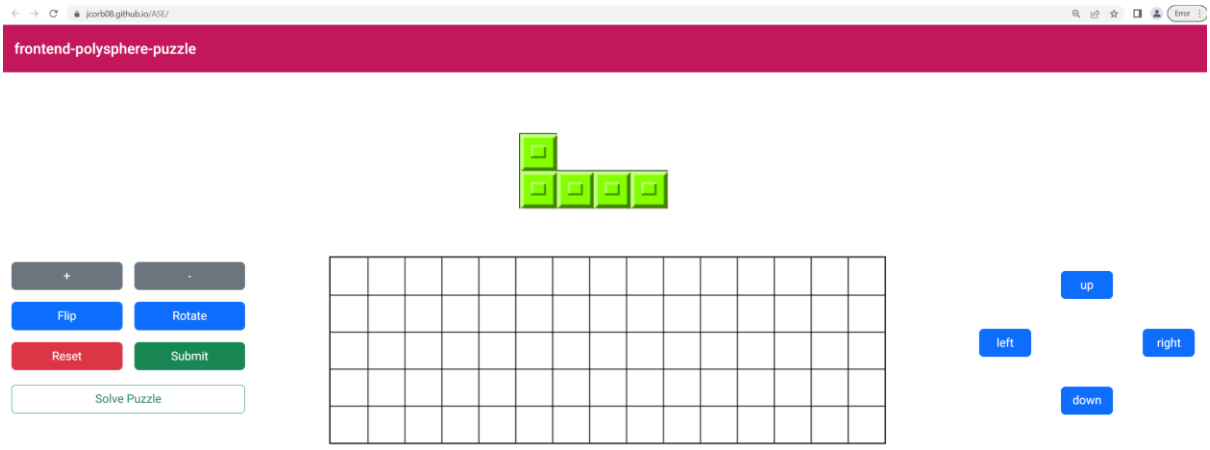
We then started to implement algorithm x without dancing links which was much more straightforward, and we were able to get a working version of this in time despite the tight timeframe. We created the matrix that algorithm x would run on by passing the shapes into a

function that would represent them on the board where they fit and create another row with them moved along one position until they won't fit on the board anymore.

Another part of the task involved being able to accept inputs from the user to create partial solutions and find all the other solutions from there. We achieved this by creating a front-end interface to accept shapes being preplaced into the solution and then writing a back-end function that triggers before the main solver function. This new function would commit the preplaced shapes to all our temporary solutions and remove the relevant conflicting rows from the matrix we would solve algorithm x on so that the solution will not try and place them again.




Above is a wireframe we created to design what our input and output will look like. And bellow is what our input and output page looked like after we had finished.




As we had successfully implemented algorithm x, the program was much faster it would have been if we had just taken a brute force approach. Unfortunately, at this point we did not know how to implement dancing links. If we had been able to implement this, we would have been able to find all the solutions much faster. If we were to do this part of the project again then we would make sure to implement dancing links properly.

Another issue we faced that would have made this portion of the project easier is that our solution to n queens was just brute force. If we had implemented dancing links or even just algorithm x without n queens, then we would have had a good portion of our code base that we could have reused. As we had not done this, we had to write the whole codebase for this section from scratch.

# Pentomino pyramid

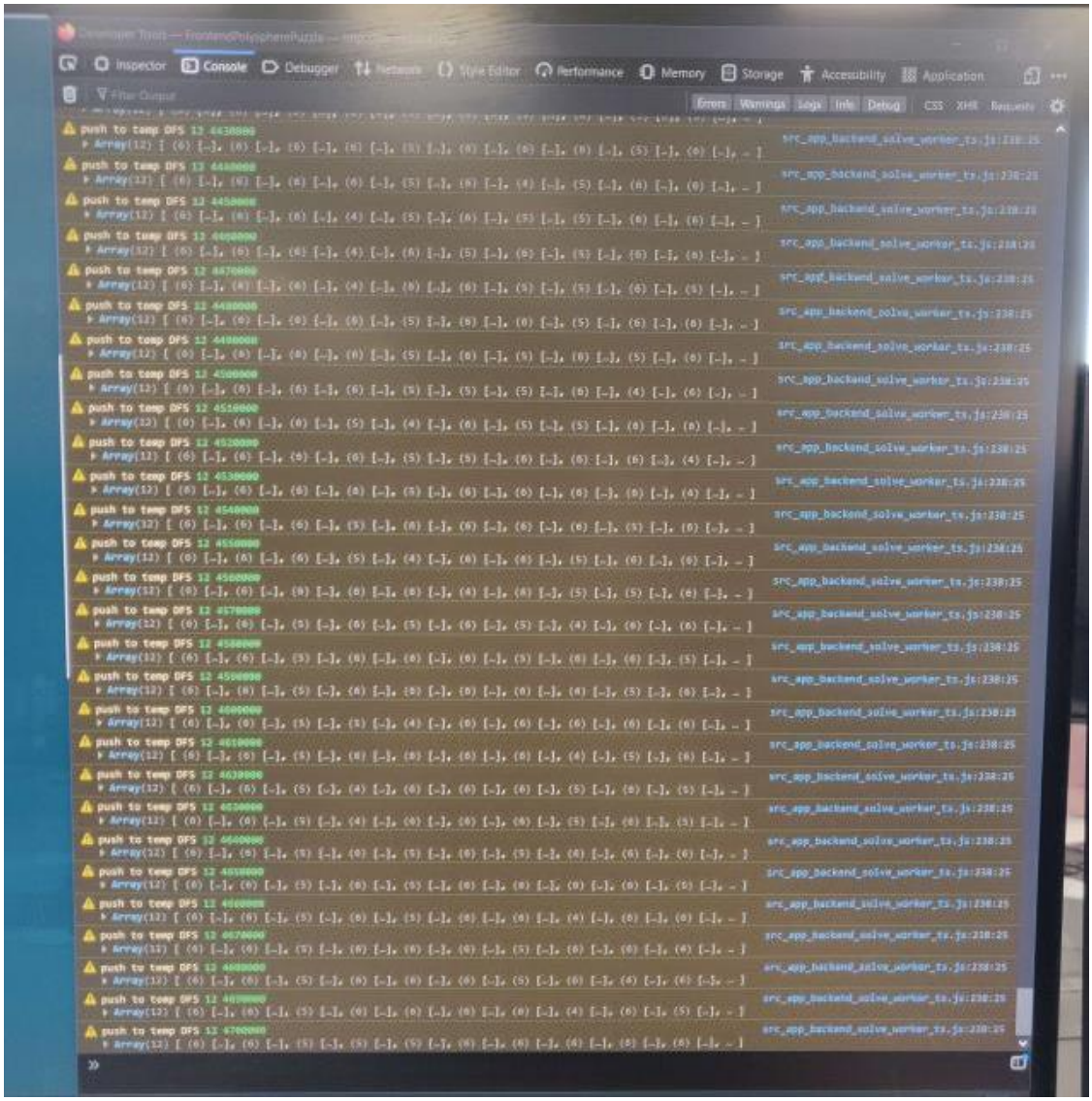
This was very similar to the previous project but instead of placing the shapes on a two-dimensional board, we would solve the problem on a 3-dimensional pyramid. we realised that this could be represented as four two dimensional boards which would allow us to solve the problem is a way quite similar to the previous project.

We knew that as this version of the exact cover allowed the shapes to be placed vertically and have sections of them on multiple of the virtual boards at once, the complexity would be much higher. This would mean there are far more possible solutions to the problem and a straightforward implementation of algorithm x without dancing links, like we used on the last section, would not be good enough to find all solutions to this problem. The first thing we did on this project, after creating our sprint board and assigning tasks to the group, was to figure out how to implement dancing links correctly.

We were able to implement dancing links correctly in a short space of time. This was due to what we learned of algorithm x from our previous project and to the research we did on dancing links at the start of this one. The more time-consuming part of the backend of the project turned out to be figuring out how to programmatically build the circular doubly linked list required for dancing links to be applied to our problem.

The method we used for creating the matrix to run algorithm x on in the last project was not suitable for dancing links and the way we stored the shapes would also have to be changed. We created a way to programmatically input all the shapes that were only lying flat in two dimensions quite quickly. Writing a function that would add them to the list, shuffle them across the board they were on one space and if the position was legal place them again, if moving them took them out of bounds we moved them down a row or to the next board if possible.

It was much more difficult to add the shapes that are present of multiple layers at once to the list. At first, we tried to hard code all of these positions into the list but quickly found that this would take too long and leave too much room for mistakes that would be hard to debug. We then came up with a method for doing this programmatically that involved defining all the possible orientations of each shape and then creating a list entry for each possible position by moving them across one position. this was more complicated than the two-dimensional shapes as we had to track which layer each part of the shape is on in order to properly move it. Once we had this working the code ran flawlessly in a backend worker outputting all the correct solutions to the console at a rate of about 1000 every half second.




Above is the console view of our code outputting 4700000 unique solutions. It kept running after this image was captured.

The front end of this project was more complex than it was in the last project. We needed to figure out how to represent the shapes in a three-dimensional space in Angular and we needed to figure out how to accept pre placements in the new board sizes. We were able to reuse a good deal of the front-end code from the last project, including all the shape representations, a sub-section of the board and the various controls that ran the program. This was helpful but there was still a lot that needed changing, including add new features to allow for a 4-dimensional pyramid, and how to preplace on a 3d space, and moving between the levels of this pyramid.

## Conclusion

In Conclusion, through this project our submissions increased in quality. Each one found solutions much faster each time as we improved our algorithm, and the front end became more professional each project. At first, we struggled as we tried to design solutions to the problems from scratch, but we soon learned how to find the generally accepted best solution to these kinds of problems and implement them. Our workflow through these projects also improved, as we made better use of git, agile and other organisational tools.