# Report for Programming Problem 1

**Team**:
Student ID: 2019216753  Name: João Carlos Borges Silva
Student ID: 2019216826  Name: Pedro Afonso Ferreira Lopes Martins

## 1. Algorithm description

### 1.1 Pre-processing:

**Corner technique:** Firstly, as we are cycling through all the pieces, we count the occurrences of each found color. We then access how many of these colors have an odd count, if the given value surpasses 4 then the puzzle is automatically impossible to solve. This works because it's a given that colors with an odd count must have a piece in a corner of the puzzle since an even number would mean that it can match properly between the pieces within the board. If in the layout of the puzzle, which is a rectangle, there are only four corners we can automatically exclude a case that exceeds this number, rendering some rejection cases almost instantaneous.

**Sides storage:** As we are reading each piece, we store in a dictionary each distinct combination of color found (side of the piece) and we add as a value the piece and rotation in question. The result of this is a complete structure where we have stored as a key all the different sides found and linked to each one of them all the pieces that match with the required rotation. This will in turn allow for a much quicker method of searching the combinations possible within the algorithm.

**Intersection storage:** At the same time as we do the **Sides storage,** we also have a separate data structure in which we perform a similar method, but in the keys, we store the combination of two consecutive sides of the piece. Again, this will optimize the search of pieces to match within the non-initial borders of the puzzle, further increasing the speed of the algorithm.

### 1.2 Base case:

Since it's given that of number of pieces will always match the size of the board, when all the pieces are placed, we know that we have reached a solution, returning **True**.

### 1.3 Rejection case:

When we do a depth search and find all possibilities exhausted there can't be a solution for the given input of pieces, returning **False**.

## 1.4 Recursive step:

This step consists in fitting a piece within the vector of pieces in the given position of the board returning **True** if the next recursive steps are successful and **False** if this doesn't occur, which leads to a rejection case.

## 2. Data structures:

**Board of the puzzle (board):** Vector with R vectors of C arrays in which each array corresponds to a piece (R – rows, C – columns).

**Used pieces (used):** A one dimensional vector of integers in which a 1 represents that the piece situated in that index is already in place. On the other hand, a 0 means that the piece is still available.

**Pieces (pieces):** A vector of arrays in which we store all the pieces given. Each array contains 4 integers.

We also have 3 unordered maps named **color_count, sides** and **intersetions** which are used for the pre-processing techniques describes in **1.1.**

## 3. Correctness:

The implementation made uses **recursion** to solve the given problem. As a simple approach our algorithm would exceed the time limit for several large problems. To increase its speed, several shortcuts and speedup tricks were added such as:

- Corner technique: which was explained in 1.1
- Side and Intersection storage: combination storage in which we quickly find matches for the first row or column by using the side storage and the rest of the board, using the intersection structure (as explained in 1.1)

The combination of these techniques allowed us to reduce the number of pieces that we cycle through without making any comparation and dramatically increase the speed of the algorithm and solve impossible puzzles almost instantaneously. We also passed certain variables by reference instead of value.
With these implementations our solution was just shy of the maximum score, which led us to take a look at our data structures. We then proceeded to change some of our vectors to arrays (only in those with a pre-established size) which in turn slightly decreased the running time of our solution, which was enough for our algorithm to reach 200 points.

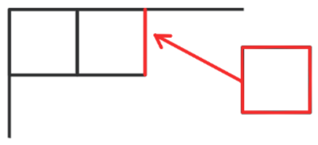

*Fig. 1 – First row insertion*

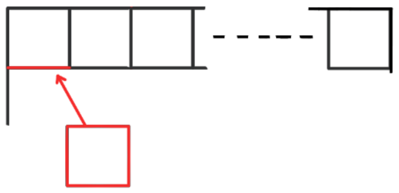

*Fig. 2 – First Column insertion*

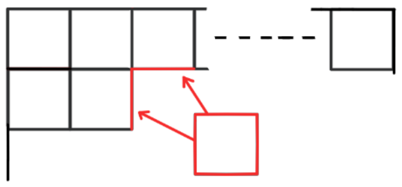

*Fig. 3 – Non-Initial Borders insertion*

## 4. Algorithm Analysis:

### Time Complexity:

In order to calculate the time complexity of an algorithm we must first take a look at the worst case, which correlates to the scenario where all the pieces to insert are at the end of the pieces vector and all pieces have n-1 possibilities for subsequent insertion. To calculate it we need the number of recursive calls of this case.
The calculus of the time complexity for the algorithm is as follows,

$$
\begin{aligned}
T(n) &= (n-1)T(n-1) + c \\
&= (n-1)[(n-2)T(n-2) + c] + c \\
&= (n-1)(n-2)T(n-2) + (n-1)c + c \\
&= (n-1)(n-2)[(n-3)T(n-3) + c] + (n-1)c + c \\
&= (n-1)(n-2)(n-3)T(n-3) + (n-2)(n-1)c + (n-1)c + c \\
&= (\dots) \\
&= \overbrace{(n-1)(n-2)\dots(n-n)}^{0}\underbrace{T(n-n)}_{T(0)} + \left[1 + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{(n-1)!}\right]c(n-1)! \\
&= \left(\sum_{k=1}^{n-1} \frac{(n-1)!}{(n-k)!}\right)c
\end{aligned}
$$

**Time complexity:** $O((N-1)!)$

The first line of the calculus demonstrates n-1 recursive calls which correlate to the worst case previously described, T(n-1) will then expand the recursion to n-2 recursion calls and so forward. T(n) represents the total temporal cost of all the recursive calls and c being the constant time reserved to each call.
Best case will be $O(\mathrm{N})$ which represents the scenario where all pieces are already sorted for insertion. The base case will have complexity $O(1)$, which means c.

### Spatial Complexity:

In order to calculate the spatial complexity of our algorithm we will need to compute the space necessary to store the required information. This will give us $4R * C$, which is the same as $4N$ required space for the board of the puzzle. S(n) represents the total spatial cost and c being the constant. All the remaining data structures will not surpass the given limit which will in turn give us a spatial complexity of,

$$S(n) = 4n + n + 1000 + 4n + \dots + c$$

**Spatial complexity:** $O(N)$

## 5. References:

https://www.cplusplus.com/reference/
https://stackoverflow.com/questions/42701688/using-an-unordered-map-with-arrays-as-keys