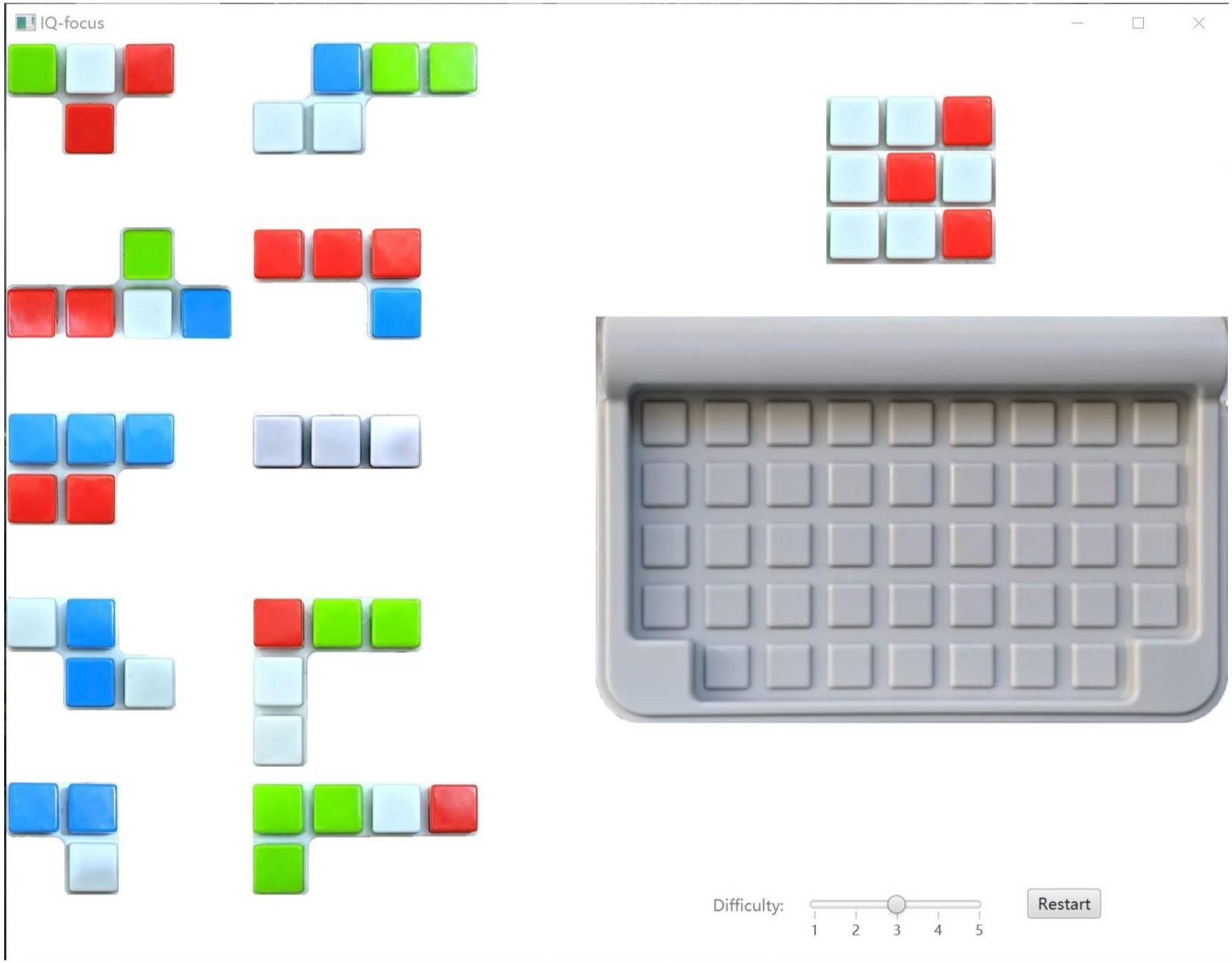
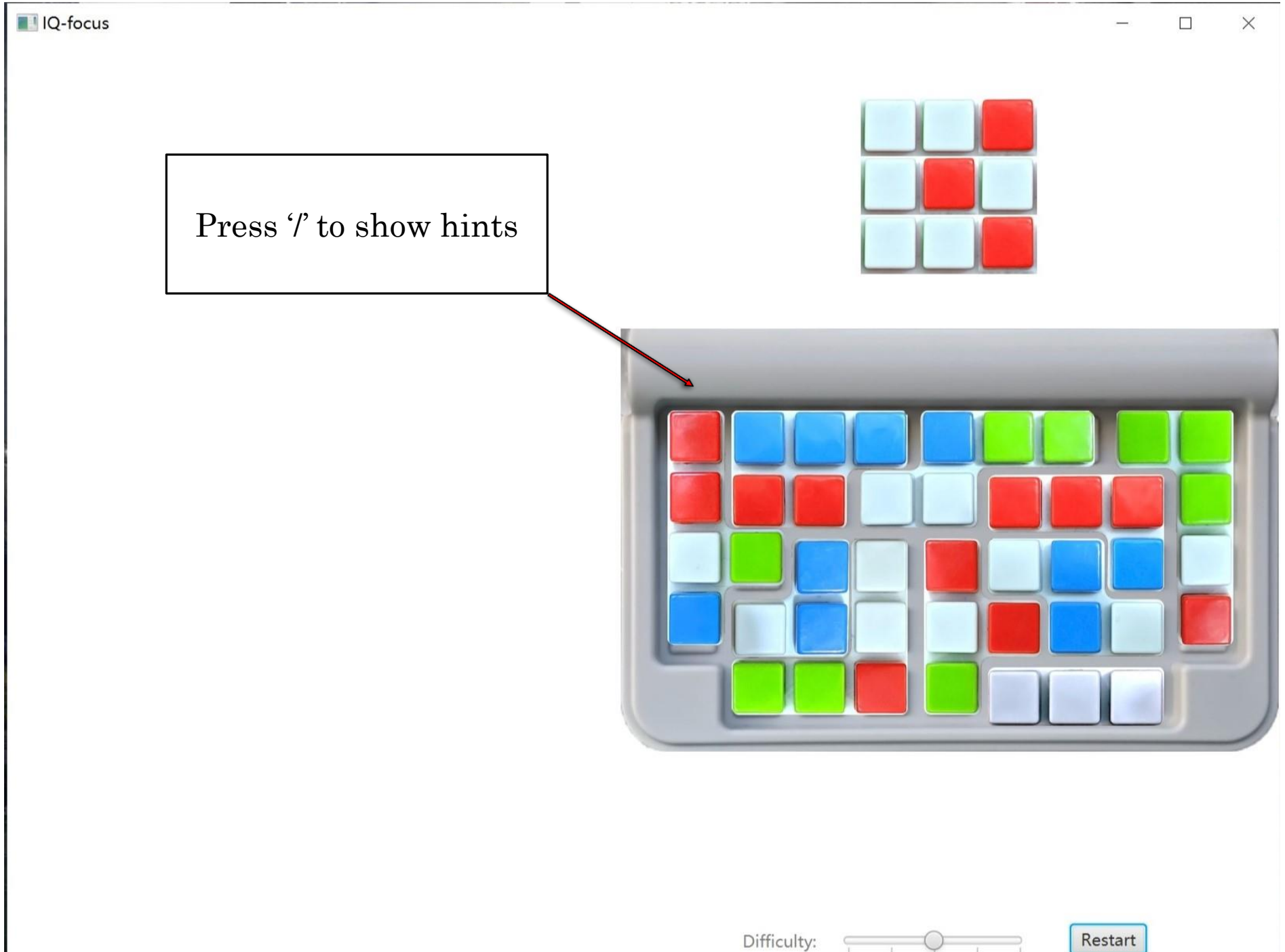
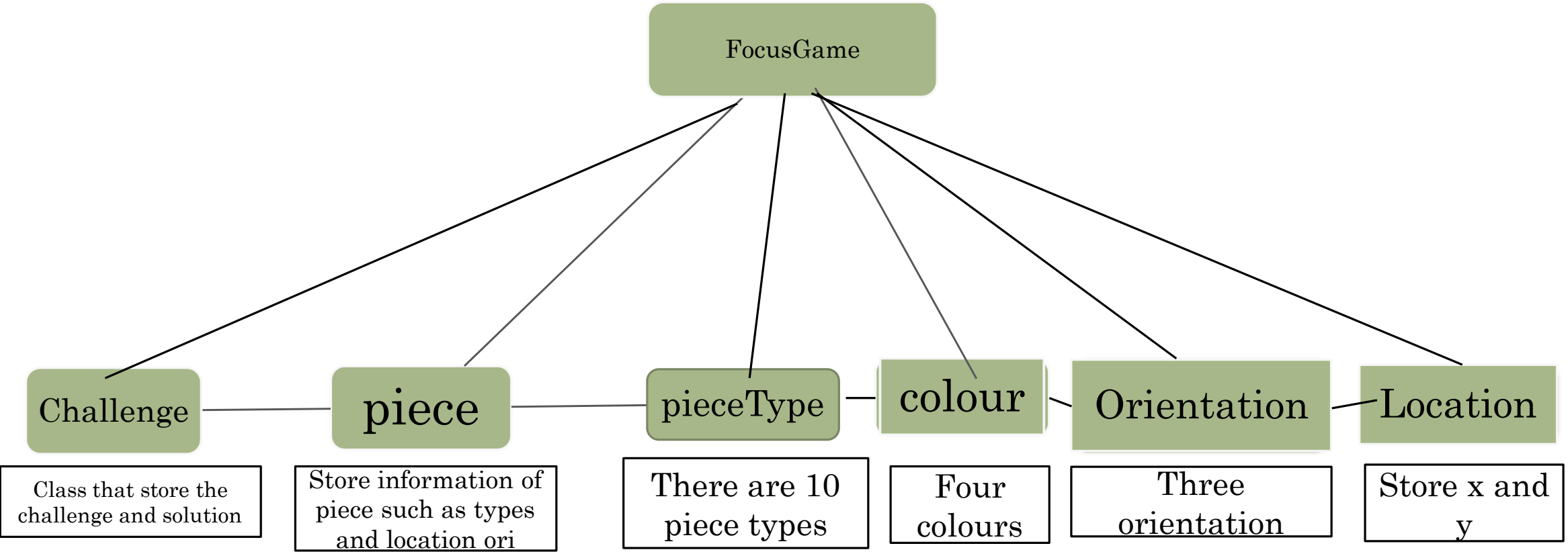
IQ – focus Game

Developed by Yixi Rao, Wenxuan Li and Fang Peng


IQ-focus
Difficulty:
1
2
3
4
5
Restart


IQ-focus
Press '/' to show hints
Difficulty:
Restart

Class structure

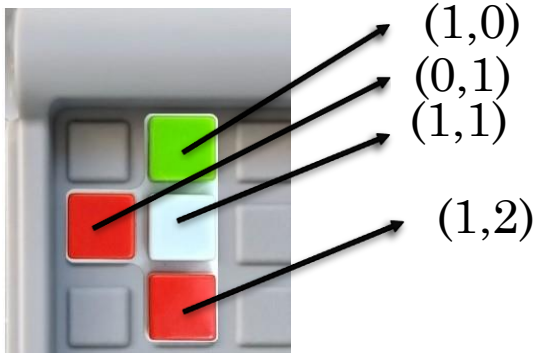


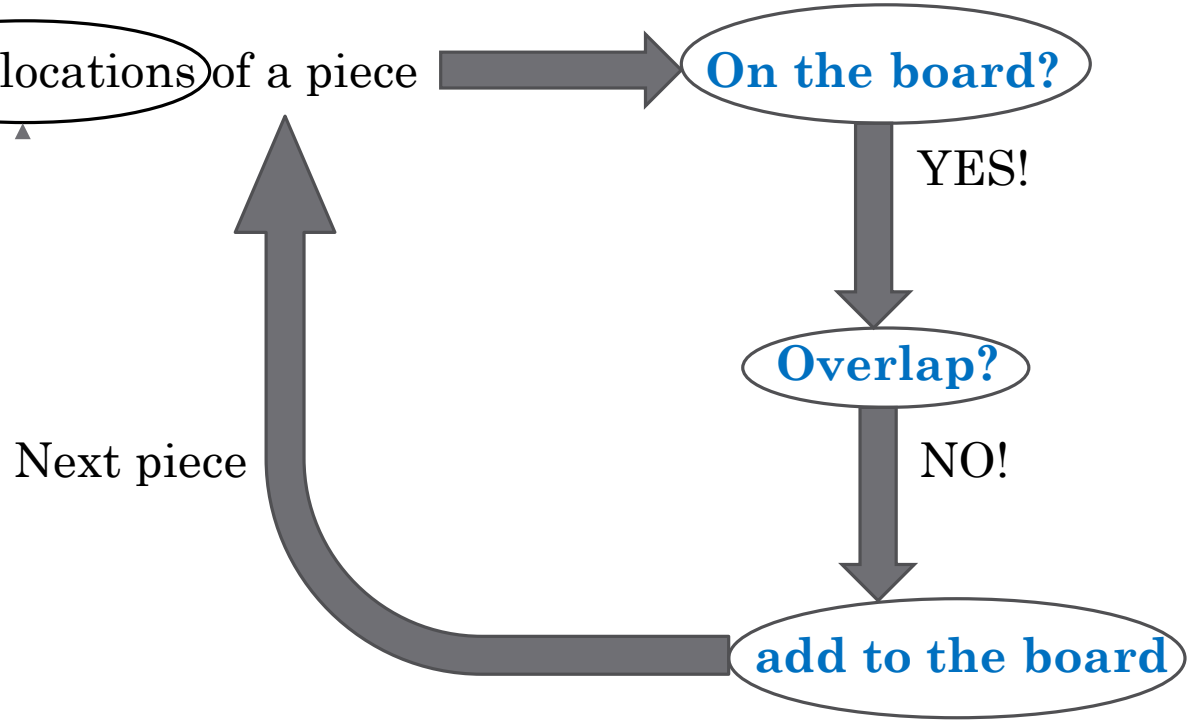
What we did

- Implement a playable board game which is IQ-Focus game and player can choose a different level of challenges and the solution corresponding to it is also given, which can be seen after the player press the ‘/’

Design approach

- **Task 5 algorithm** (we first design the algorithm then try to code it)
- Find all the locations of a piece





-

Task 6 optimisation and approach

- **Basis idea** : for every point on the board and every type and every orientation to determine whether it is viable pieces.

For all Y(from 0 to 5)

For all X(from 0 to 8)

For all types(from a to i

For all orientation

```
for (int y:Yrange){                    //for all the x on the board try to find a piece
    for (int x:Xrange){                //for all the y on the board try to find a piece
        for (String type: types){      //for all the types on the board try to find a piece
            for (String orientation:orientations){
                if (isPlacementStringValid(placement + type + "" + x + "" + y + "" + orientation) && isOccupyGrid
                    result.add(type + "" + x + "" + y + "" + orientation);
```

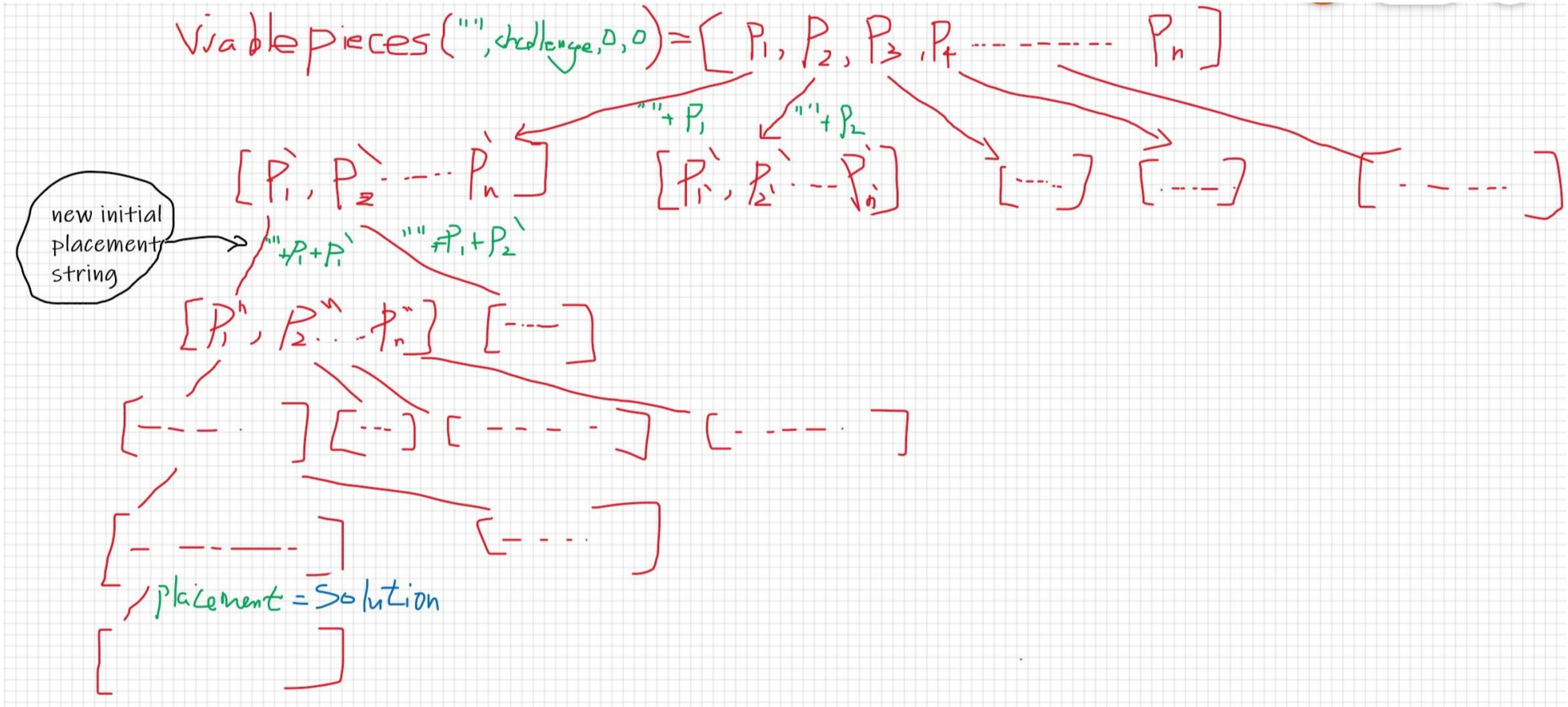
We optimize this program in two aspects.

- **1. curtail locations**: No piece length is longer than 4 so if the distance between the location we want to cover and the location in the loop is longer than 4, it will be skipped.
- **2. Types extraction** : If the types have already appeared in the initially given placement string, then it will be eliminated

```
for (int y:Yrange){                    //for all the x on the board try to find a piece
    if (Math.abs(y - row) > 3)
        continue;
    for (int x:Xrange){                //for all the y on the board try to find a piece
        if (x == 0 && y== 4)
            continue;
        if (Math.abs(x - col) > 3)
            continue;
        for (String type:refineTypes(placement)){   //for all the types on the board try to find a piece
            for (String orientation:orientations){
                if (isPlacementStringValid(placement + type + "" + x + "" + y + "" + orientation) && isOccupyGrid(col,row,new
```

Task 9 optimisation and approach

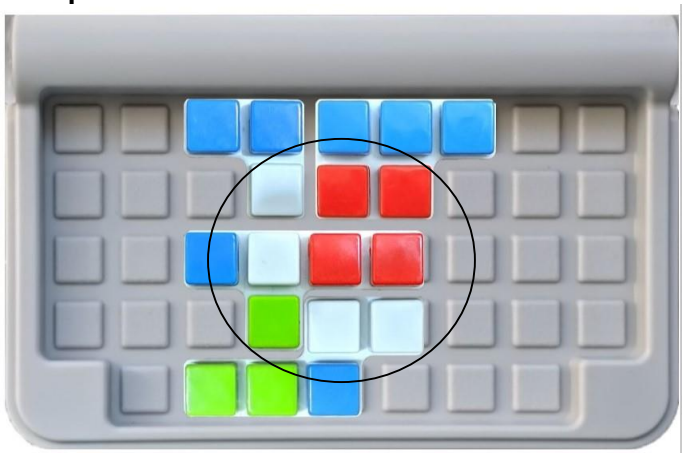
- **Early idea**: Start from the first point (0,0), and call the function of task 6 to find viable pieces and then Recursively loop through each piece and reuse the task 6 function until finding the correct one.



- **Result**: too slow!

Optimized program of task 9

- Still, go through each location but this time we first find a set of placement string that occupies the central nine locations and satisfies the challenge. Pass each placement string as an initial placement
- E.g. central placement string



- Then Based on this set, do the recursion

- **How did we come up with that**: There's a big difference in speed when we use just one piece or more pieces as an initial placement compared without any pieces as initial placement string.
- **Reason:** And it will reduce the time of searching because it prunes off useless branches of all possible placement strings.

Other small optimisation

- **1**. if it is already placed on the board, skip it and Jump to the next point until the current location is not occupied.

```
while (pieces[y][x] != null){   //if the loication is already occupied by some pieces than it will be skipped
    Location nextP1 = nextPoint(x,y);
    x = nextP1.getX();y = nextP1.getY();
}
```

  - It will optimize the program since it can never go through some points which are already occupied

- **2.** Once find the dead cell (dead cell means an empty cell surrounded by occupied cell), jump out of the loop!
- E.g.

```
if (viablePieces == null){   // it is null means it is a dead cell and this a optimisation
    pieces = new Piece[5][9];
    return "";
}
```

  - viablePieces == null means No viable piece can occupy this location so it is a dead cell.
  - **Explanation**: skipping a dead cell can cut off the useless branches of that loop. So that is the main reason why we use it.

Interesting aspects of our game

- **2.** Ingenious and interesting way of storing the piece location after snapping the piece to the board

```
/* it will store the message of each piece which is already placed to the board e.g. a001*/
private String[] pieceState = new String[10];
```

- (after we drag a piece into the board the information of piece will be stored in this array)

Notice that if we combine all the element in the array, we will get a placement string then we can do interesting thing

- **3.** Another interesting aspect is that we call the function of task 5 to judge whether a location is occupied or not.

```
private boolean validPiece() {
    String placement = "";
    for (String p:pieceState){  //set up all the placement as a placement string
        placement = placement + p;
    }
    if (placement.equals(""))
        return true;
    return FocusGame.isPlacementStringValid(placement);     //use already placement string and add it to new piece
}
```

(we can use that array to create a placement string and pass it to task 5 function to test whether it is valid or not.)

If valid allow to rotate

Otherwise snap to hone

- Reusing the function can make our program coherent and space-saving.

Thank you!