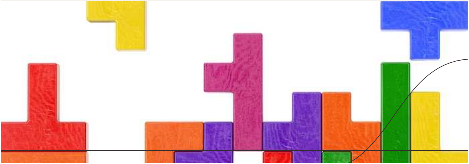
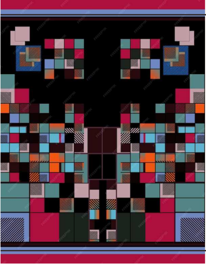
Project 5: Make a square

Welcome to **our** universe, where creativity meets challenge. In this presentation, we will delve into the *magic* of the 4x4 square algorithm and uncover its secrets. Get ready to unlock the code and elevate your game to new heights!



Our Goal

For our project, we want to create a program that will solve various pentomino puzzles by using recursive backtracking to fill a given board. It will attempt to find all possible solutions for the given board.
Problem Definition



Problem Definition

The problem is to make a square with size 4X4 by using 4 or 5 pieces. The pieces can be rotated or flipped, and all pieces should be used to form a square.
There may be more than one possible solution for a set of pieces, and not every arrangement will work even with a set for which a solution can be found.

Algorithm

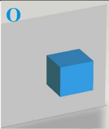
The algorithm used in this program is done by the recursive function solve(). It consists of 6 nested for loops. The first for loop iterates through all pieces, the second for loop iterates through all permutations for the given piece, and the third and fourth for loops iterate through the board's y and x coordinates, respectively.

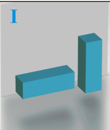
The pseudocode for the algorithm is as follows:

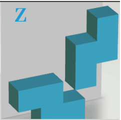
```
solve(board, pieces, solutions):
   for each piece in pieces:
      for each permutation for the piece:
         for each y coordinate of the board:
            for each x coordinate of the board:
               place piece(board[y][x], piece, permutation, board)
               if piece has been placed:
                  remove placed piece from pieces
               if pieces is empty:
                  record solution
               else solve(updated board, updated pieces)
placepiece(board[y][x], piece, permutation, board):
   for each y coordinate of the piece:
      for each x coordinate of the piece:
         if the current square of the piece is filled:
         y = board y + piece y
         x = board x + piece x
```

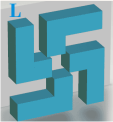
```
if the piece goes out of bounds:
return false
if board[x][y] is not empty:
return false
board[x][y] = piece
return true
```

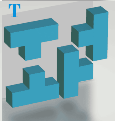
Set pieces

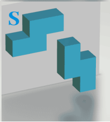

O

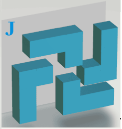

I


Z


L


T


S


J

MasterThread(Input pieces)

The class has several instance variables, including inputPieces, keyThreadLJT, keyThreadSZI, t1, t2, t3, t4, and key0fSolve.The constructor MasterThread(int[] inputPieces) initializes the inputPieces variable with the provided array.The run() method is the main entry point for the thread. It contains the logic for solving the puzzle using multiple threads.Inside the run() method, four HashMaps (pieces1, pieces2, pieces3, pieces4) are created to store puzzle pieces.The board1setup() method is called four times to set up the puzzle boards (usablePieces1, usablePieces2, usablePieces3, usablePieces4) using the pieces1, pieces2, pieces3, pieces4 HashMaps, respectively.The key0fSolve variable is assigned the value of usablePieces1.Four instances of the multiThreading class (m1, m2, m3, m4) are created, each with the corresponding pieces HashMap and usablePieces array.Four threads (t1, t2, t3, t4) are created with the respective multiThreading instances and started.t1.join(), t2.join(), t3.join(), and t4.join() are called to wait for the completion of all four threads.If the solutionsThread list in m1, m2, m3, or m4 is not empty, a PrintSolutionsThreads instance is created with the first solution and a thread (p1) is started to print the solution.If the solutionsThread list is empty for all four multiThreading instances, a default PrintSolutionsThreads instance is created, and a thread (p1) is started to handle the case when no solutions are found.

Board1setup

The code defines various 2D arrays representing different pieces (e.g., pieceZa, pieceZb, pieceIa, etc.). Each piece is represented by a 2D array of integers, where each integer represents a specific shape or color of the piece.ArrayLists are created to store different rotations of each piece. For example, pieceZ is an ArrayList that contains two rotations (pieceZa and pieceZb) of the Z-shaped piece.Similar ArrayLists are created for other pieces, such as pieceI, pieceJ, pieceL, pieceO, pieceS, and pieceT. Each ArrayList contains different rotations of the respective piece.The code initializes a copy of the inputPieces array.A loop iterates over the copy_inputPieces array to determine which pieces need to be placed on the game board. It checks if the value at each index is non-zero and finds the corresponding piece from the ArrayLists created earlier.Based on the value of the piece, a switch statement is used to add the appropriate piece to the "pieces" HashMap. If the value is 1, it adds the single rotation of the piece. If the value is greater than 1, it adds multiple rotations of the piece to the HashMap.After adding the piece to the HashMap, the value in copy_inputPieces is decremented, and the loop continues until all pieces have been placed on the board.Finally, the code creates a copy of the inputPieces array and initializes a Set of keys from the "pieces" HashMap. It converts the set of keys to an array.

pieceKeyList: This method takes a HashMap called pieces and returns an array of integers pieceKeyList. It initializes an integer array pieceKeyList with the same size as the pieces HashMap. Then, it iterates over an array called array0fKey (possibly a typo, should be arrayOfKey) and assigns each element to the corresponding index in pieceKeyList. Finally, it increments keyThreadLJT if it is less than 3 and increments keyThreadSZI if it is less than 1. The method then returns pieceKeyList.

setupcopy: This method takes an ArrayList of 2D integer arrays called piece and an integer i. It creates a new ArrayList called piecescopy to store the copied pieces. It then iterates over each element (piececopy) in the piece ArrayList. For each piececopy, it creates a new 2D integer array newpiece with the same dimensions. It multiplies each value in piececopy by i and assigns it to the corresponding position in newpiece. After copying all the pieces, it adds newpiece to the piecescopy ArrayList. Finally, it returns piecescopy.

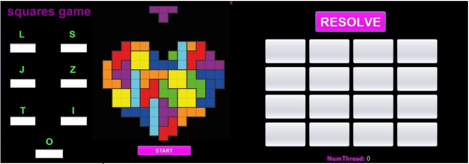
print2DArray: This method takes a 2D integer array called myArray and prints it in a formatted manner. It calculates the length of the array in the y-direction (yLength) and the length in the x-direction (xLength). It then prints a line of hyphens (-) with a length of xLength+2. Next, it iterates over each row in myArray and prints a vertical bar (|) at the beginning of each row. Within each row, it iterates over each column and prints the corresponding element. If the element is less than 10, it adds an extra space before printing it. Finally, it prints a vertical bar at the end of each row and another line of hyphens (-).

MultiThreading

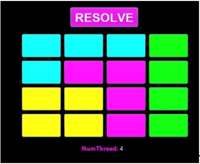
- The **multiThreading** class implements the Runnable interface, indicating that it can be executed in a separate thread. It contains several fields, including boardYdim and boardXdim, representing the dimensions of the puzzle board, pieceYdim and pieceXdim, representing the dimensions of the puzzle pieces, and other fields for storing the puzzle pieces, solutions, and other necessary variables. The multiThreading constructor initializes the piecesThread and usablePieces fields with the provided parameters.
- The **solve** method is the main method for solving the puzzle. It takes the puzzle board, usable pieces, depth, solutions, and pieces as parameters. It iterates over each usable piece and its permutations. For each permutation, it tries to place the piece on each point of the board. If the piece can be placed, it creates a new board and updates the newPieces array to exclude the placed piece. If there are no remaining pieces, it adds the solution to the solutions list. Otherwise, it recursively calls the solve method with the updated board and pieces.

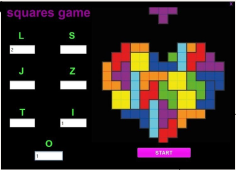
- The **placePiece** method attempts to place a piece on the board. It iterates over each point in the piece and checks if it can be placed at the corresponding position on the board. If all points can be placed, it updates the board accordingly and returns true. Otherwise, it returns false.
- The **doesSolutionExist** method checks if a solution already exists in the solutions list. It compares the provided solution with each solution in the list and returns true if a match is found. Otherwise, it returns false.
- The **run** method overrides the run method of the Runnable interface. It creates a new board, calls the solve method to start solving the puzzle, and prints “Done” when the solving process is complete.

GUI TO I/O PIECES



EXAMPLE





Our team collaborated closely to do this project . We worked together, combining our individual skills and expertise, to develop a multi-threaded solution for the puzzle. Each team member played a crucial role in the process, contributing their knowledge and ideas to create an efficient and effective implementation. Through effective communication and coordination, we were able to divide the tasks, analyze the problem, and design a solution that met our requirements. We leveraged the power of teamwork to overcome challenges, share insights, and continuously iterate on our work. This project stands as a testament to the strength of our collaboration and the collective effort we put forth to achieve our goals.

Thanks!

Farida Ahmed 20210674
Farah Khaled 20210673
Fatma Mohamed 20210667
Nouran Yasser 20211015
Amr Mohamed 20210645
Omar Mohamed 20210625