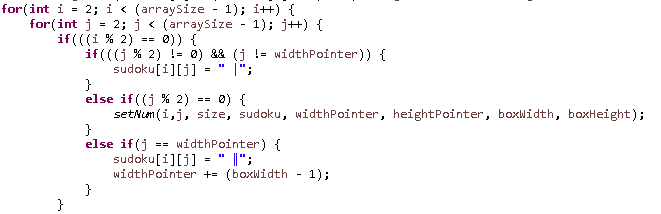
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size of Puzzle | Total amount Horizontal and Vertical “Hashes” | Number of Boxes (Row x Columns) | Size of Each Individual Boxes | Total 2D-Array Size |
| Standard 9 x 9 | 2 down and 2 across | 3 x 3 | 3 x 3 | 20 x 20 |
| 16x16 | 3, 3 | 4 x 4 | 4 x 4 | 34 x 34 |
| 25 x 25 | 4, 4 | 5 x 5 | 5 x 5 | 52 x 52 |
| 10 x 10 | 1, 4 | 5 x 2 | 2 x 5 | 22 x 22 |
| 15 x 15 | 2, 4 | 5 x 3 | 3 x 5 | 32 x 32 |
| 20 x 20 | 3, 4 | 5 x 4 | 4 x 5 | 42 x 42 |

**Sudoku Puzzle Program Planning Part 1 (Java Program)**

* For any N x N size of puzzle, if N is a perfect sqrt…
  + Only, sqrt and not any other roots, because only sqrt will give us a N x N
  + The total amount of horizontal and vertical hashes is (SQRT(N) – 1)
  + The number of boxes and its size is SQRT(N) x SQRT(N)
  + The total 2d-array size is array [(N + (N + 1))] x [(N + (N + 1))]
    - If individual boxes do not have boarders (will be hard to trace), then the total 2d-array size is [(N + (N) + 1 + 1))] x [(N + (N) + 1 + 1))]
      * The extra + 1 is for boarder for user reference. This can be removed if converted into a GUI program.
* For any N x N size of puzzle, if N is NOT a perfect sqrt (possible, but ugly) …
  + The total amount of horizontal and vertical hashes is (columns – 1, row – 1)
  + The total number to boxes is size of each box inverted [(r, c) 🡪 (c, r)]
  + To find the total size of each individual boxes…
    - If the remainder of N divided by the floor of SQRT(N) is 0
      * Size of box is (floor of SQRT(N), N / floor of SQRT(N))
    - Else, apply the previous concept, but with (SQRT(N) – 1), so that we have a smaller number by a larger number
  + The total 2d-array size is array [(N + (N + 1)) + 1] x [(N + (N + 1)) + 1]
    - The extra + 1 is for boarder for user reference. This can be removed if converted into a GUI program.
  + The height of a box is how many boxes there are in one column; the width of a box is how many boxes there are in one row

**Methods**

1. Main
   1. Get size
   2. Make puzzle
   3. Take answers
   4. Ask user if they want to play again
2. Make puzzle
   1. Form grid system
      1. Two while loops to create the outer two boarders
      2. Use table above and picture below to help make grid system
   2. Fill in grid
   3. Take out sections
   4. Print puzzle (how we would print out a regular 2D-array)
3. Fill in grid
   1. Put all possible numbers into an array (attempt I):

**private** **static** **void** setNum(**int** row, **int** col, **int** size, String[][] sudoku, **int** wPointer, **int** hPointer, **int** boxW, **int** boxH) {

List <Integer> possibleAns = **new** ArrayList<>();

**for**(**int** i = 0; i < size; i++) { possibleAns.add(i + 1); }

**int** [] boxNums = *getBoxNumber*(row, col, sudoku, wPointer, hPointer, boxW, boxH);

**for**(**int** a: boxNums) {

**if**(possibleAns.contains(a)) { possibleAns.remove(possibleAns.indexOf(a)); }

}

**for**(**int** i = 2; i < sudoku[1].length; i += 2) {

**if**(!sudoku[row][i].equals(" ") && (possibleAns.contains(Integer.*parseInt*(sudoku[row][i]))))

possibleAns.remove(possibleAns.indexOf(Integer.*parseInt*(sudoku[row][i])));

**if**(!sudoku[i][col].equals(" ") && (possibleAns.contains(Integer.*parseInt*(sudoku[i][col]))))

possibleAns.remove(possibleAns.indexOf(Integer.*parseInt*(sudoku[i][col])));

}

//(Math.random() \* ((max - min) + 1)) + min)

**int** chosenIndex = (**int**)((Math.*random*() \* (((possibleAns.size() - 1) - 0) + 1)) + 0);

**if**((possibleAns.size() != 0) && (possibleAns.get(chosenIndex) < 10))

sudoku[row][col] = "0" + String.*valueOf*(possibleAns.get(chosenIndex));

**else** **if**((possibleAns.size() != 0))

sudoku[row][col] = String.*valueOf*(possibleAns.get(chosenIndex));

}

**private** **static** **int**[] getBoxNumber(**int** r, **int** c, String[][] sudoku, **int** w, **int** h, **int** boxW, **int** boxH) {

**int** [] temp = **new** **int**[(((boxW - 1) / 2) \* ((boxH - 1) / 2))];

**int** count = 0;

// if(r < w) {

r = ((h - boxH) + 2);

// }

// else{

// w += (boxW - 1);

// }

// if(c < h) {

c = ((w - boxW) + 2);

// }

// else{

// h += (boxH - 1);

// }

**for**(**int** i = r; i < h; i += 2) {

**for**(**int** j = c; j < w; j += 2) {

**if** (!sudoku[i][j].equals(" "))

temp[count++] = Integer.*parseInt*(sudoku[i][j]);

}

}

**return** temp;

}

* + 1. Read in the column, row, and box number
    2. Compare these numbers to the one in the array
    3. Remove the same numbers (no repeats)
    4. To find box numbers, close in on the target
    5. Pick a random index of the array, and that number will be the chosen one
    6. This method is an idea, but contains deadlocks
  1. Put all possible numbers into an array (attempt II):
     1. Start with 1 and then all the way to max
     2. List out available rows and column
     3. Set row and column limit pointers (this will also limit to a certain box)
     4. Pick a random row and column and check if there is already a number there, and if there is, pick a new point
     5. Place number
     6. Make sure there are no repeating numbers in any column, row, or box
        1. Can assure this by removing the available row and column as we go through the program
     7. This will also end in deadlock when the chosen row in the box is all full
  2. Ways to deal with deadlocks:
     1. Try moving the number in the box above the current box to the column that is having trouble
     2. Clear the whole row of boxes of that number and redo the number for that row of boxes
     3. Clear the whole puzzle of that number and redo the numbering
     4. Clear the whole puzzle

1. Take out numbers
   1. Each box can have between 44% to 55% of its number (this percentage is subject to change as more testing will be done)
   2. Deep copy 2D array to later compare and see if user completed puzzle correctly
   3. Randomly replace points in copied array with “ “ while following rule in point a
2. Take in answers from user
   1. Format: [row, col, answer]
   2. Putting answer at coordinate where answer already exists will override old answer
   3. Restrict user from overwriting/removing preexisted numbers
   4. Check for the validity of user’s answer
   5. Keep record of user answers, because it’s impossible to differentiate between user’s answers and the original puzzle
   6. User can pass and get a new puzzle if they think this puzzle is too hard
   7. \*\*\*Hints\*\*\*
      1. User can ask for hints
      2. Can give user a set of possible numbers or just the answer
      3. Hints are limited depending on the size of the puzzle
   8. Press enter to submit the puzzle (with confirmation)
3. Check answer
   1. Once user submits the puzzle, compare it with original array
   2. For coordinates with different numbers, replace it with ‘<>’
   3. Ask user if they want to correct their puzzle

\*\*\*ALL BOARDER DESIGNS ARE SUBJECT TO CHANGES\*\*\*

