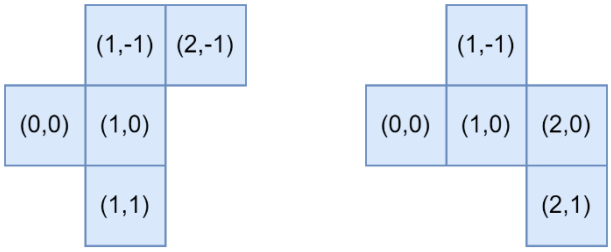
Introduction

The key to reducing the pentomino problem is to get it into a format that can utilize Knuth's dancing links algorithm. In an exact cover problem, the goal is to select rows in a matrix such that each column has coverage exactly once. With this in mind, what are the "columns" I need exact coverage of for our pentomino problem? I need to utilize each of the 12 shapes (F, I, L, N, P, T, U, V, W, X, Y, Z) exactly once, so I know I will need 12 columns for these. Additionally, these pieces need to fill the container they're placed in completely, so there will be (12 x 5 = 60) cells that will need columns of their own. Tying this together, I will need a matrix that has 12 columns for the pieces and 60 columns for the cells, totaling 72 columns. The order of these does not matter, they could be randomly distributed if I wished, but for simplicity's sake I will designate the first 12 as the columns set aside for pentomino pieces. The number of rows that are needed will vary based on what the dimensions are for the container used. Because some of the pentominoes require 3 cells vertically and 3 cells horizontally, our smallest dimension can be a 3x20. Other valid configurations for this problem include 4x15, 5x12, and 6x10. Each of these configurations will have their own number of rows that will represent valid placements for each piece.

Calculating Valid Placements

For this step, I find the different orientations for each shape by using a coordinate system focused on their top-left most square. To remediate trivial solutions, I normalized the "F" shape to only two positions – its standard shape and rotated 90 degrees.



Once I have these coordinates defined for each shape, to calculate valid placements, I simply iterate through each cell of the container and place the shape's (0,0) block there. By adding the cell's coordinates to each block coordinate for the shape, I can see if the shape will fit by seeing if it extends past the container boundaries.

(0,0)	(1,0)	...																...	(19,0)
(0,1)	(1,1)	...																...	(19,1)
(0,2)	(1,2)	...																...	(19,2)

The grid for a 3x20 container

Below, you see an example of an invalid placement. I'm testing if the "F" piece fits on the (0,0) cell. Visually, that is easy to confirm. But mathematically, it is also simple.

	(1,-1)	(2,-1)																
(0,0)	(1,0)	...															...	(19,0)
(0,1)	(1,1)	...															...	(19,1)
(0,2)	(1,2)	...															...	(19,2)

An invalid piece placement

(0,0) + (0,0) = (0,0)
(0,0) + (1,0) = (1,0)
(0,0) + (1,1) = (1,1)
(0,0) + (1,-1) = (1,-1)
(0,0) + (2,-1) = (2,-1)

The offending numbers are highlighted above. When the calculation is done, I compare each X and Y value to the bounds of the container. In the case of the 3x20 configuration, those bounds are:

X: [0,20)
Y: [0,3)

Therefore, if either number is negative, or is greater than or equal to the number of rows or columns, the placement is invalid. See below for an example of a proper placement.

(0,0)	(1,0)	...															(1,-1)	(2,-1)
(0,1)	(1,1)	...														(0,0)	(1,0)	(19,1)
(0,2)	(1,2)	...															(1,1)	(19,2)

A valid piece placement

(17,1) + (0,0) = (17,1)
(17,1) + (1,0) = (18,1)
(17,1) + (1,1) = (18,2)
(17,1) + (1,-1) = (18,0)
(17,1) + (2,-1) = (19,0)

All of these X,Y coordinates fall within the bounds of the given container, so the placement is valid. As a row, this placement is represented by a 1 in the column designated for the piece "F", then a 1 in the column for each of the 19, 20, 38, 39, and 59 cells, with a 0 in every other column.

Encoding the Matrix

For each of these valid placements found, I add a new row to the configuration's matrix – representing each cell as a column from left to right, top to bottom. In the example shown here, you can see some valid placements for the "F" shape, which is indicated by the 1 in the first index of the row.

The highlighted section represents the first cell of the container

After all valid placements have been calculated, you can see how this has been properly reduced to an exact cover problem. I want each column covered exactly once. Doing so in the first 12 columns will ensure I use each pentomino but never reuse one and doing so in the last 60 columns will ensure every space in the container is covered by only a single piece.

Algorithm

To implement the algorithm, I follow Knuth's guidance on pg. 5 of his article:

If $R[h] = h$, print the current solution (see below) and return.
Otherwise choose a column object c (see below).
Cover column c (see below).
For each $r \leftarrow D[c], D[D[c]], \ldots,$ while $r \neq c$,
    set $O_k \leftarrow r$;
    for each $j \leftarrow R[r], R[R[r]], \ldots,$ while $j \neq r$,
        cover column j (see below);
    search$(k+1)$;
    set $r \leftarrow O_k$ and $c \leftarrow C[r]$;
    for each $j \leftarrow L[r], L[L[r]], \ldots,$ while $j \neq r$,
        uncover column j (see below).
Uncover column c (see below) and return.

```
private void search(int k) {
    if(head.R == head) {
        solutions++;
//          printSolution();
        return;
    }
    ColumnNode c = selectColumn();
    c.cover();
    for(DataNode r = c.D; r != c; r = r.D) {
        if(solution.size() < (k + 1)) solution.add(r);
        else solution.set(k, r);
        for(DataNode j = r.R; j != r; j = j.R) {j.C.cover();}
        search(k + 1);
        r = solution.get(k);
        c = r.C;
        for(DataNode j = r.L; j != r; j = j.L) {j.C.uncover();}
    }
    c.uncover();
}
```

In the base case, I have a slight variation due to the nature of our problem. First, I increment a solutions variable that keeps track of how many solutions have been found. Additionally, I have commented out the printSolution() method as I expect thousands of results for all configurations except 3x20.

Finally, changes had to be made to the buildDancingLinks() method in the DLX class we were provided. The original was placing all the DataNodes into the first two ColumnNodes instead of their proper location, this was due to it using a ForEach loop to pull out the value (which can only be 0 or 1) and then it used that value as the index of the ColumnNode to attach it to. To correct this, I created the betterBuildDancingLinks() method that made slight changes to this loop and iterated through each column index instead of their values.

Results

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) throws Exception {
        ArrayList<ArrayList<Integer>> matrix3x20 = PositionEncoder.loadShapes(3);
        DLX dlx3x20 = new DLX(matrix3x20.get(0).size(), matrix3x20);
        dlx3x20.run();
        System.out.println("(3x20) # of Solutions: " + dlx3x20.getNumberOfSolutions());

        ArrayList<ArrayList<Integer>> matrix4x15 = PositionEncoder.loadShapes(4);
        DLX dlx4x15 = new DLX(matrix4x15.get(0).size(), matrix4x15);
        dlx4x15.run();
        System.out.println("(4x15) # of Solutions: " + dlx4x15.getNumberOfSolutions());

        ArrayList<ArrayList<Integer>> matrix5x12 = PositionEncoder.loadShapes(5);
        DLX dlx5x12 = new DLX(matrix5x12.get(0).size(), matrix5x12);
        dlx5x12.run();
        System.out.println("(5x12) # of Solutions: " + dlx5x12.getNumberOfSolutions());

        ArrayList<ArrayList<Integer>> matrix6x10 = PositionEncoder.loadShapes(6);
        DLX dlx6x10 = new DLX(matrix6x10.get(0).size(), matrix6x10);
        dlx6x10.run();
        System.out.println("(6x10) # of Solutions: " + dlx6x10.getNumberOfSolutions());
    }
}
```

```
Run: Main
"C:\Program Files\Java\jdk-16.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\I
(3x20) # of Solutions: 2
(4x15) # of Solutions: 368
(5x12) # of Solutions: 1010
(6x10) # of Solutions: 2339

Process finished with exit code 0
```

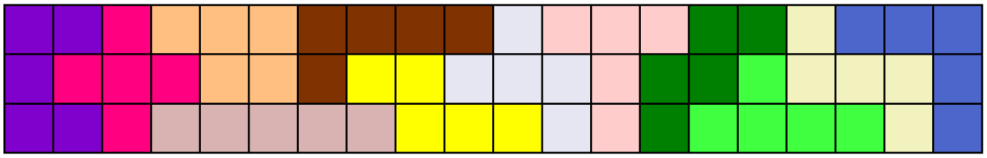
As you can see, I get the expected number of solutions that are given to us in the assignment.

For each of our four configurations, I call upon a PositionEncoder class that first creates a new TXT file containing the exact cover matrix, and then loads that matrix back into an ArrayList of ArrayList<Integer> so that it can be processed. After this data has been loaded, all I need to do is run the DLX that I associated with the matrix, this will run the dancing links algorithm on the data and find all the solutions.

Below, I uncomment the printSolution() method and run this process again for only 3x20 so that you can see some real solutions.

Note: Each of the numbers are 11 larger than their real cell number to account for the columns I set aside for each pentomino piece (0-11). So cell 14 relates to cell 3 in the visual representation.

```
X 14 33 34 35 54
U 12 13 32 52 53
P 15 16 17 36 37
I 55 56 57 58 59
L 18 19 20 21 38
N 39 40 60 61 62
F 22 41 42 43 63
T 23 24 25 44 64
W 26 27 45 46 65
Y 47 66 67 68 69
V 29 30 31 51 71
Z 28 48 49 50 70
```



```
X 29 48 49 50 69
U 30 31 51 70 71
P 46 47 66 67 68
I 24 25 26 27 28
Z 23 43 44 45 65
Y 42 61 62 63 64
W 21 22 40 41 60
T 18 19 20 39 59
F 17 36 37 38 58
N 34 35 55 56 57
L 13 14 15 16 33
V 12 32 52 53 54
```