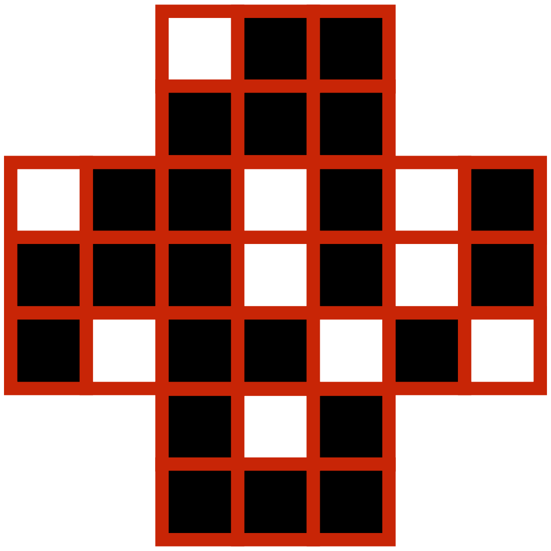
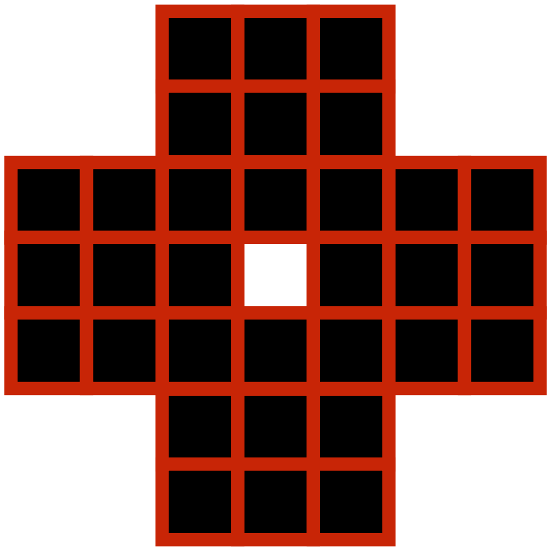
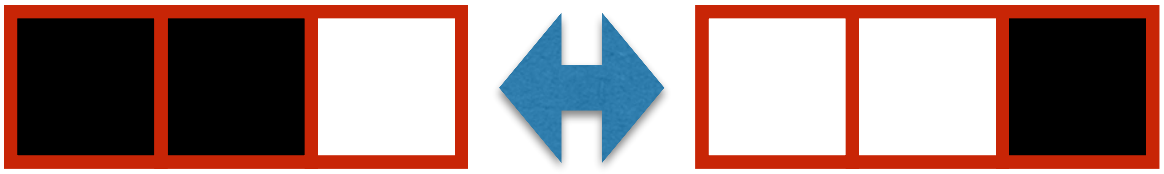
Computer Science COMP-250 Homework #4 *v*4.0
Due Friday April 1st, 2016

A “**HIЯIQ**” (pronounced higher-I.Q.) puzzle is an array of 33 black or white pixels (bits), organized in 7 rows, 4 of which contain 3 pixels each and the middle 3 rows contain 7 pixels each. You are given a particular configuration of these pixels as in (A).


(A)

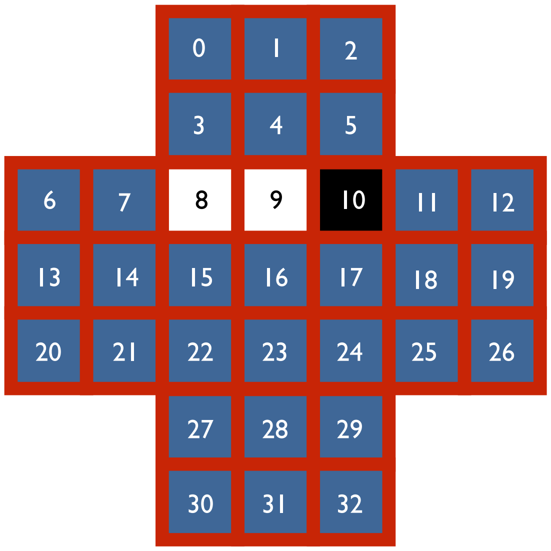

(B)

Your task is to write a JAVA program that will transform this configuration into the *solved* configuration (B). To modify your configuration you are allowed to apply one of two substitution rules whenever the rule is applicable: anywhere in your configuration if one of these patterns is found (horizontally or vertically), you can replace it with the other:



We recommend that you use a tree/graph data structure to represent each reachable configuration of the “**HIЯIQ**” puzzle. The children of a configuration should be the possible configurations that can be reached applying a single substitution rule. Your algorithm should traverse the tree so to locate the *solved* configuration. Once the solution is found, your algorithm should return a **String** containing the sequence of substitutions necessary to solve the puzzle. For convenience we will name the “BBW to WWB” substitution a **B-substitution** and the “WWB to BBW” substitution a **W-substitution**.

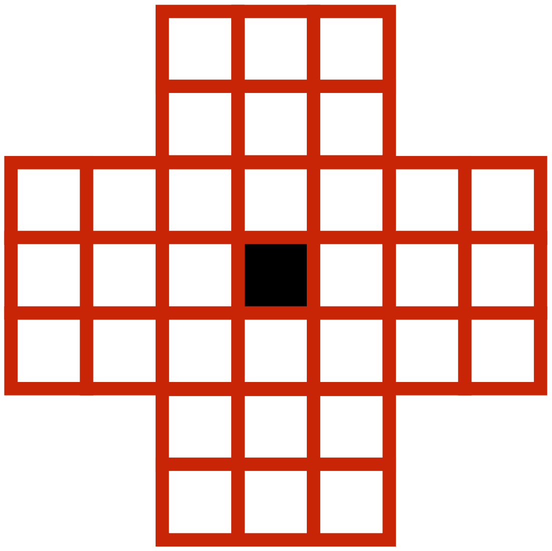
We will use the following notation to represent the locations on the "**HIЯIQ**" board:



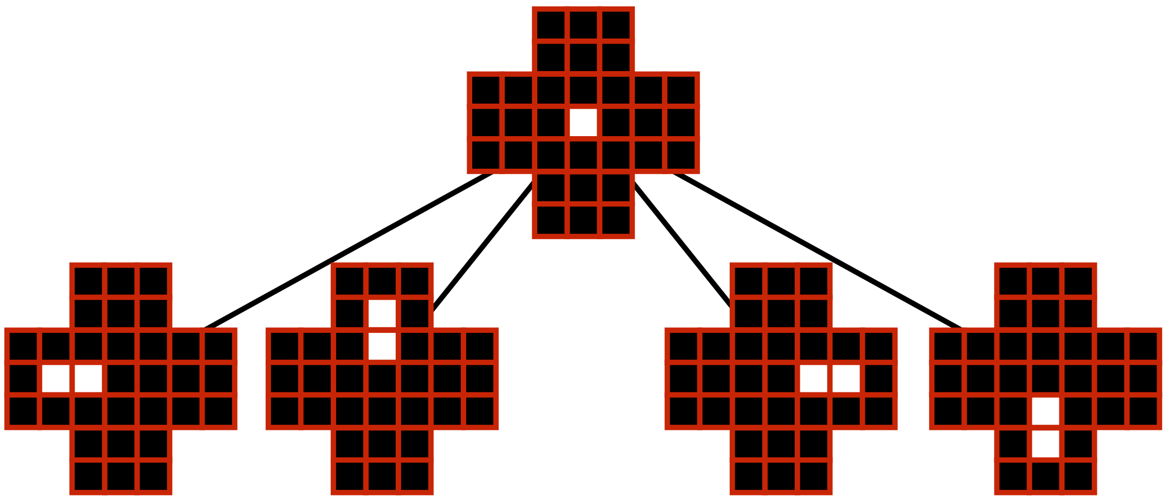
For instance "8W10" means a W-substitution may be applied to pixels at positions 8,9,10. Your output should be something like : "18W16, 5B17,…, 16W28". We will provide you with an Object type **HiRiQ** that you must use for uniformity reasons. This object will contain methods for testing if the puzzled is solved, and outputting a board configuration. This will be available through the course web page very soon.

In order to make your program (space) efficient, you must implement some sort of memory management policy to avoid overflowing the virtual machine memory carelessly. You will be evaluated on the ability of your program to solve various puzzles: maybe your program cannot find a solution from ALL configurations, but maybe it can find a solution for a configuration with only 20 white pixels on it, or only 10 white pixels,… Examples on various configurations to be solved will be provided by us on the course web page as well. The more complex puzzles you can solve, the better grade you will get. **Note:** if you choose to solve this problem *without explicitly exploring a tree*, you must explain where a *virtual* tree is being explored and *how* it is explored (DFS, BFS, etc).

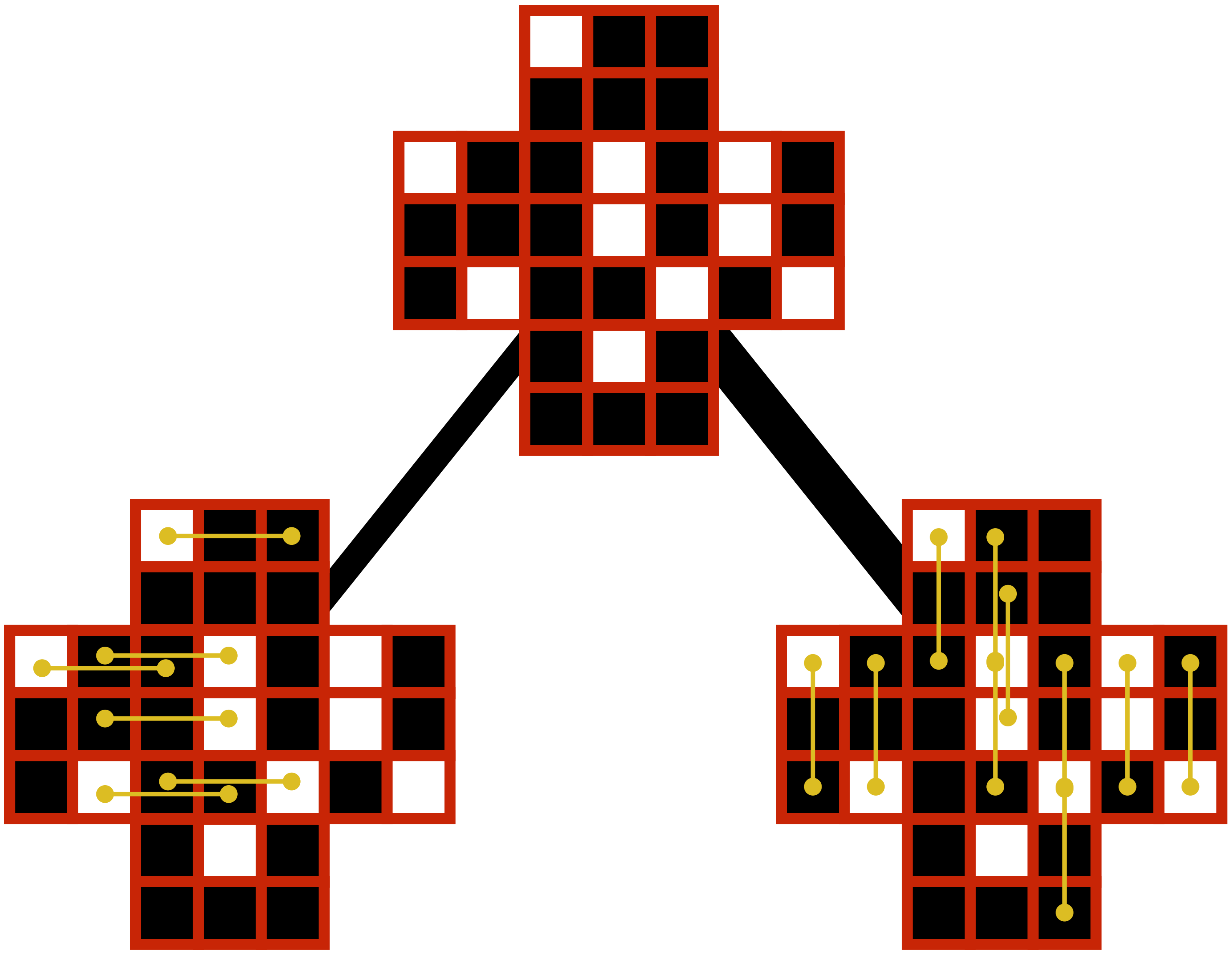
The standard "HI-Q™" puzzle has a unique start-configuration (with 32 white pixels) *opposite* to the *solved* state and only allows W-substitutions:



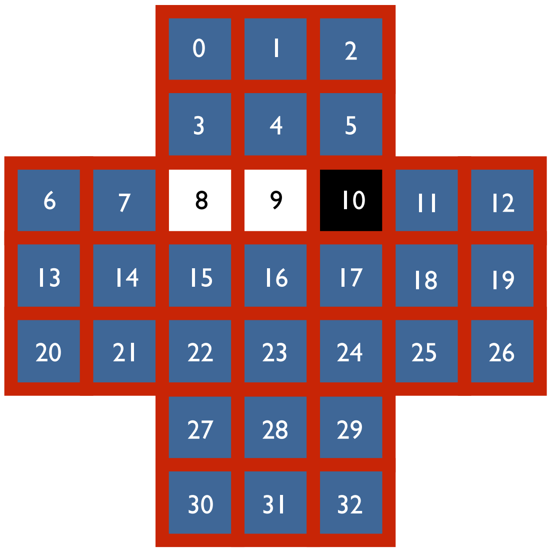
In a first example, we have the *solved* state with its four children :



In a second example, we have the configuration (A) from above with the horizontal substitutions to get 6 of its children and the vertical substitutions to get its other 10 :



List of 38 triplets:



In general, when looking for the children of a given configuration, the following 38 triplets should be considered for applying the substitution rules:

[0 , 1 , 2]
[3 , 4 , 5]

[6 , 7 , 8]
[7, 8, 9]
[8 , 9 ,10]
[9 ,10,11]
[10,11,12]

[13,14,15]
[14,15,16]
[15,16,17]
[16,17,18]
[17,18,19]

[20,21,22]
[21,22,23]
[22,23,24]
[23,24,25]
[24,25,26]

[27,28,29]
[30,31,32]

[12,19,26]
[11,18,25]

[2 , 5 ,10]
[5 ,10,17]
[10,17,24]
[17,24,29]
[24,29,32]

[1 , 4 , 9]
[4 , 9 ,16]
[9 ,16,23]
[16,23,28]
[23,28,31]

[0 , 3 , 8]
[3 , 8 ,15]
[8 ,15,22]
[15,22,27]
[22,27,30]

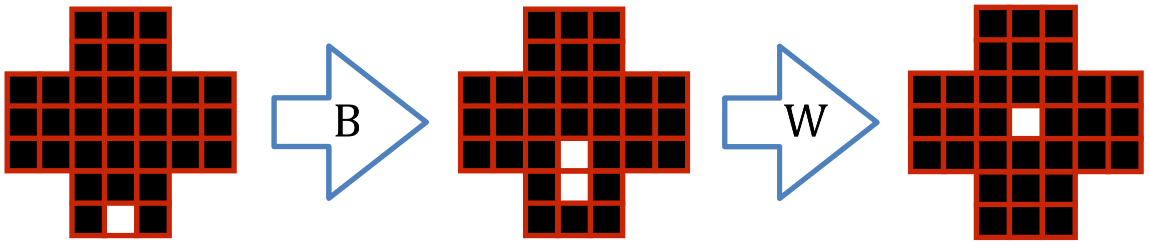
[7 ,14,21]
[6 ,13,20]

Description of a general solution:

The general approach to this problem is to locally generate the neighbourhood of the given configuration of type **HiRiQ**. To that effect, enumerate all 38 places where a substitution rule might be applied and create new nodes for configurations reachable by a single substitution. Some configurations have as little as zero neighbours and some configurations have as many as 38 neighbours.

Just for fun: find all configurations with zero neighbours (I believe there are 16 of those) and all configurations with 38 neighbours (I believe there are 16 of those).

Once you have generated the neighbourhood of the original configuration you will start moving around in these nodes looking for the *solved* configuration. Notice that the B-substitution decreases the number of black pixels whereas the W-substitution increases the number of black pixels. So in some sense, the W-substitution gets you closer to your goal whereas the B-substitution gets you further from your goal. Some configurations will force you to use the B-substitution to increase your number of white pixels before you can reach the *solved* configuration using W-substitutions. Here is an example:



For this assignment, sky is the limit. All sorts of improvements can be considered. You must submit a working solution to get some points. Once you have a basic working solution, it is up to you to improve it as much as you like. Here are a few ideas about possible improvements:

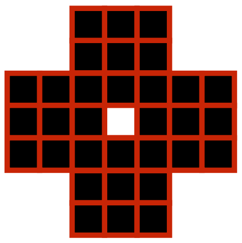
- To increase the number of solvable puzzles, first build a solution table of ALL configurations with only a few white pixels. Use that table as soon as you reach a configuration with few enough white pixels.

- Are all configurations (except for the 16 with zero neighbours) reachable from the *solved* state ? If not, can you rapidly identify those unsolvable configurations ??

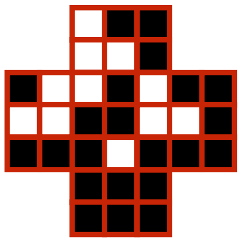
- If possible avoid using B-substitutions to converge faster towards the goal.

Some helper code is found below. 1) HiRiQ(n) creates a HiRiQ object with 4 possible values for n=0,1,2,3. For an object *conf* of type HiRiQ, 2) *conf*.IsSolved() decides if it is solved, 3) *conf*.store(B) stores an array B of booleans in *conf*, while 4) *conf*.load(B) returns B filled with booleans. Finally, 5) *conf*.print() prints *conf*'s formatted contents.

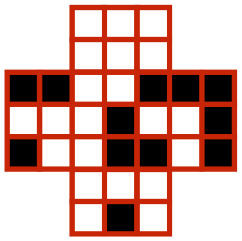
```
public class tester {
        public static void main(String[] args) {

        class HiRiQ
        {
        //int is used to reduce storage to a minimum...
        public int config;
        public byte weight;

        //initialize the configuration to one of 4 START setups n=0,1,2,3
        HiRiQ(byte n)
        {
        if (n==0){config=65536/2;weight=1;}
        else if (n==1){config=4403916;weight=11;}
             else if (n==2){config=-1026781599; weight=21;}
                  else{config=-2147450879; weight=32;}
        }
```


(n=0)

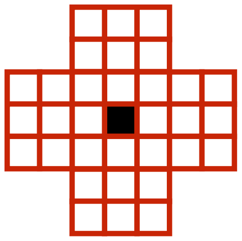
```
        //Boolean saying whether it is solved
        boolean IsSolved()
        {return( (config==65536/2) && (weight==1) );}

        //transforms the array of 33 booleans to an (int) config and a (byte) weight.
        public void store(boolean[] B)
        {int a=1; config=0; weight=(byte) 0;
         if (B[0]) {weight++;}
         for (int i=1; i<32; i++)
          {if (B[i]) {config=config+a;weight++;}
          a=2*a;}
         if (B[32]) {config=-config;weight++;}
        }
```


(n=1)

```
        //transforms the int representation to an array of 33 booleans.
        //the weight (byte) is necessary because only 32 bits are memorized
        //and so the 33rd is decided based on the fact that the config has the correct weight or not.
        public boolean[] load(boolean[] B)
        {
         byte count=0;
         int fig=config;
         B[32]=fig<0;
         if (B[32]) {fig=-fig;count++;}
         int a=2;
         for (int i=1; i<32; i++)
         {
          B[i]= fig%a>0;
          if (B[i]) {fig=fig-a/2;count++;}
          a=2*a;
         }
         B[0]= count<weight;
         return(B);
        }
```

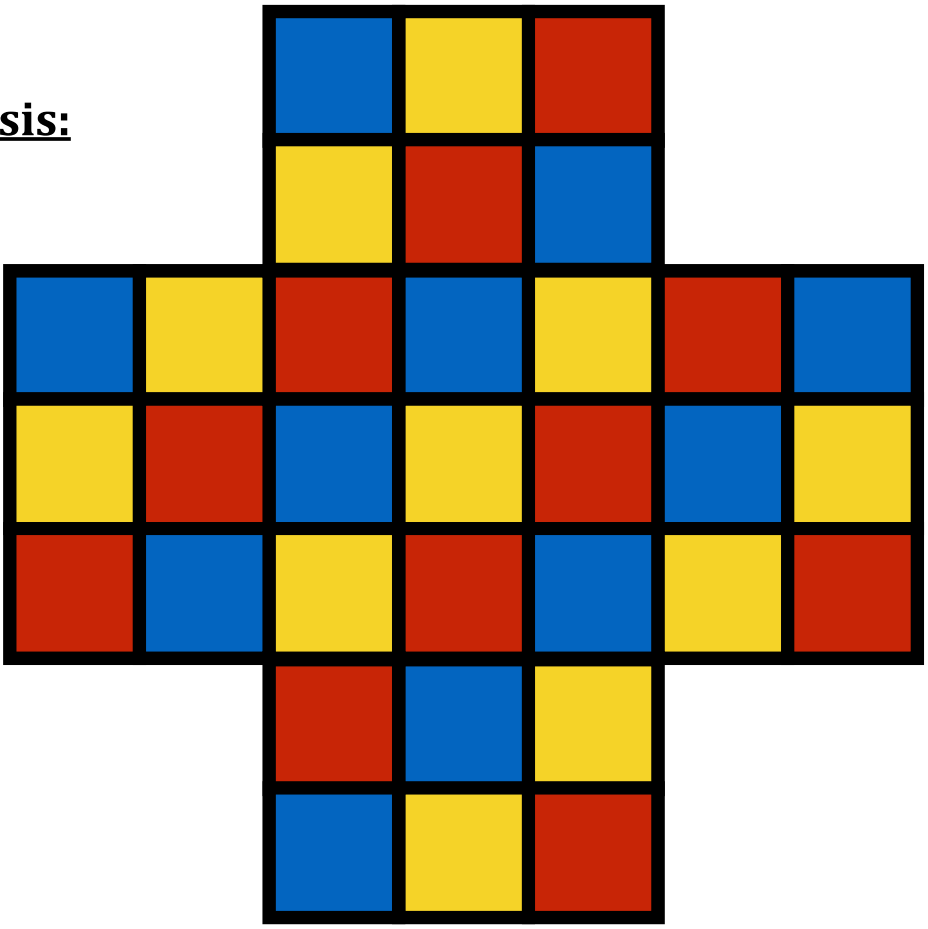

(n=2)

```
        //prints the int representation of an array of booleans.
        public void printB(boolean Z)
        {if (Z) {System.out.print("[ ]");} else {System.out.print("[@]");}}

        public void print()
        {
        byte count=0; int fig=config; boolean next,last=fig<0;
        if (last) {fig=-fig;count++;}
        int a=2;
        for (int i=1; i<32; i++)
        {
        next= fig%a>0; if (next) {fig=fig-a/2;count++;}
        a=2*a;
        }
        next= count<weight; count=0; fig=config;
        if (last) {fig=-fig;count++;}
        a=2;
        System.out.print("      ") ; printB(next);
        for (int i=1; i<32; i++)
        {
        next= fig%a>0;
        if (next) {fig=fig-a/2;count++;}
        a=2*a;
        printB(next);
        if (i==2 || i==5 || i==12 || i==19 || i==26 || i==29) {System.out.println() ;}
        if (i==2 || i==26 || i==29) {System.out.print("      ") ;};
        }
        printB(last); System.out.println() ;
        }
        }
```

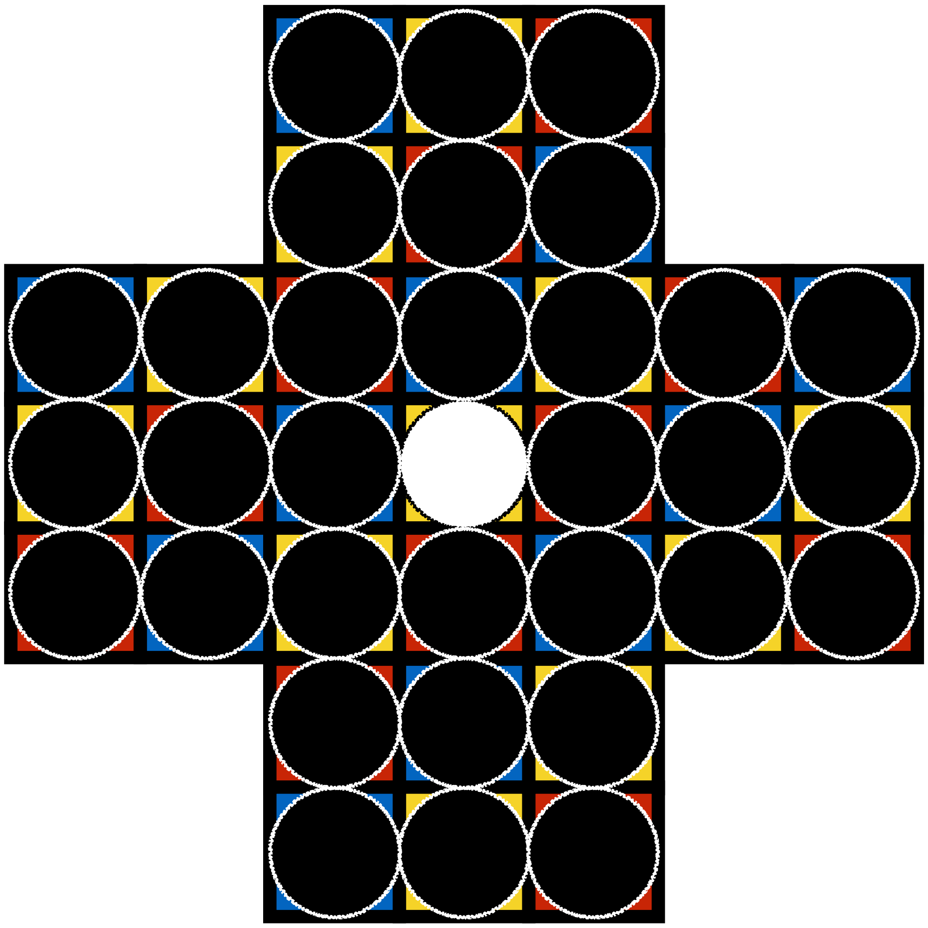

(n=3)

```
        HiRiQ W=new HiRiQ((byte) 0) ; W.print(); System.out.println(W.IsSolved());
        HiRiQ X=new HiRiQ((byte) 1) ; X.print(); System.out.println(X.IsSolved());
        HiRiQ Y=new HiRiQ((byte) 2) ; Y.print(); System.out.println(Y.IsSolved());
        HiRiQ Z=new HiRiQ((byte) 3) ; Z.print(); System.out.println(Z.IsSolved());
        boolean[] B=new boolean[33]; B=Z.load(B);
        for (int i=0; i<33; i++){B[i]= !B[i];};
        Z.store(B); Z.print(); System.out.println(Z.IsSolved());
        }
}
```

Some Analysis:



Consider the above 3-colouring of the puzzle cells. Every cell is either blue, yellow or red. No two adjacent cells are of the same colour. It contains exactly 11 blue cells, 11 yellow cells and 11 red cells. All the allowed substitutions will flip exactly one pixel on a blue cell, one pixel on a yellow cell and one pixel on a red cell. A-L-W-A-Y-S…

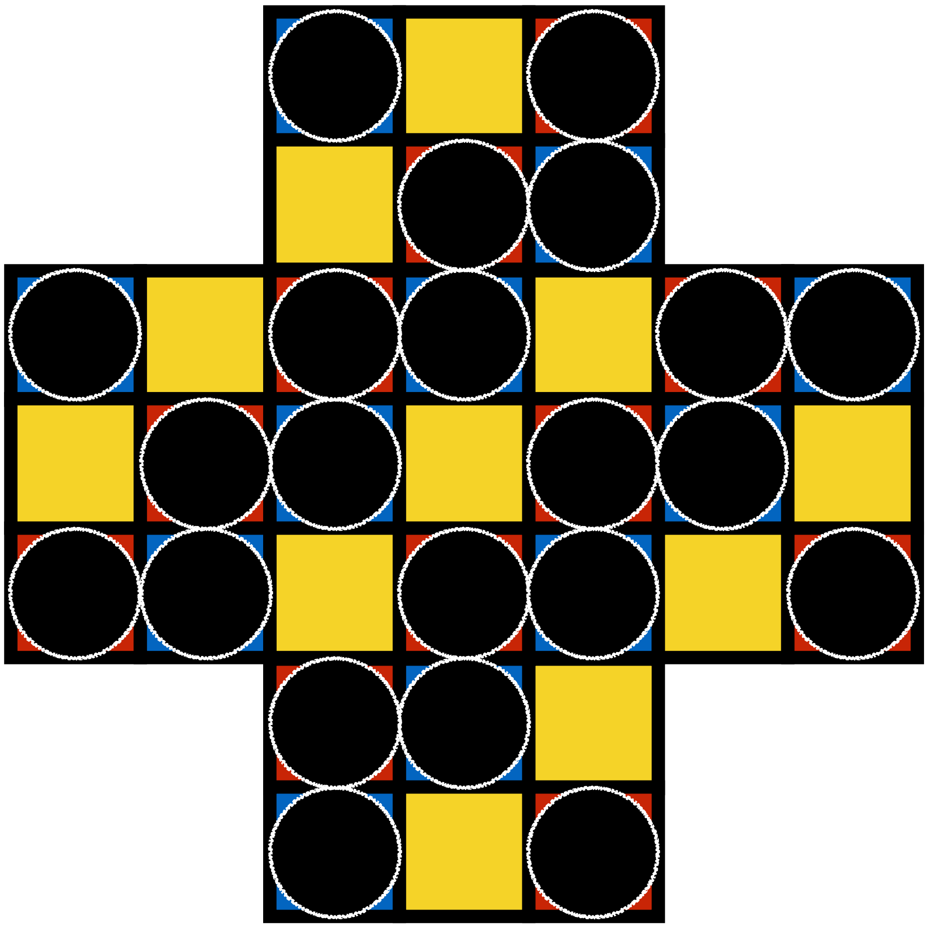


Notice that the *solved* configuration contains one white pixel on a yellow cell, zero white pixel on blue or red cells. We can write this as a triplet: (0,1,0). Whatever substitution we apply will switch the number of white pixels on each of the three colours from even to odd or the opposite: the parity (even / odd) of the number of white pixels on each type of coloured cells will always be the same for blue and red and always opposite for yellow. For example, starting from the *solved* state (0,1,0) it may be changed up to (1,0,1) and then back down to (0,1,0) or up to (2,1,0) or (0,1,2). The parities of blue and red will always be the same and yellow the opposite. Therefore any configuration with colour distribution (B,Y,R) with 0≤B≤11 white pixels on blue cells, 0≤Y≤11 white pixels on yellow cells and 0≤R≤11 white pixels on red cells must satisfy

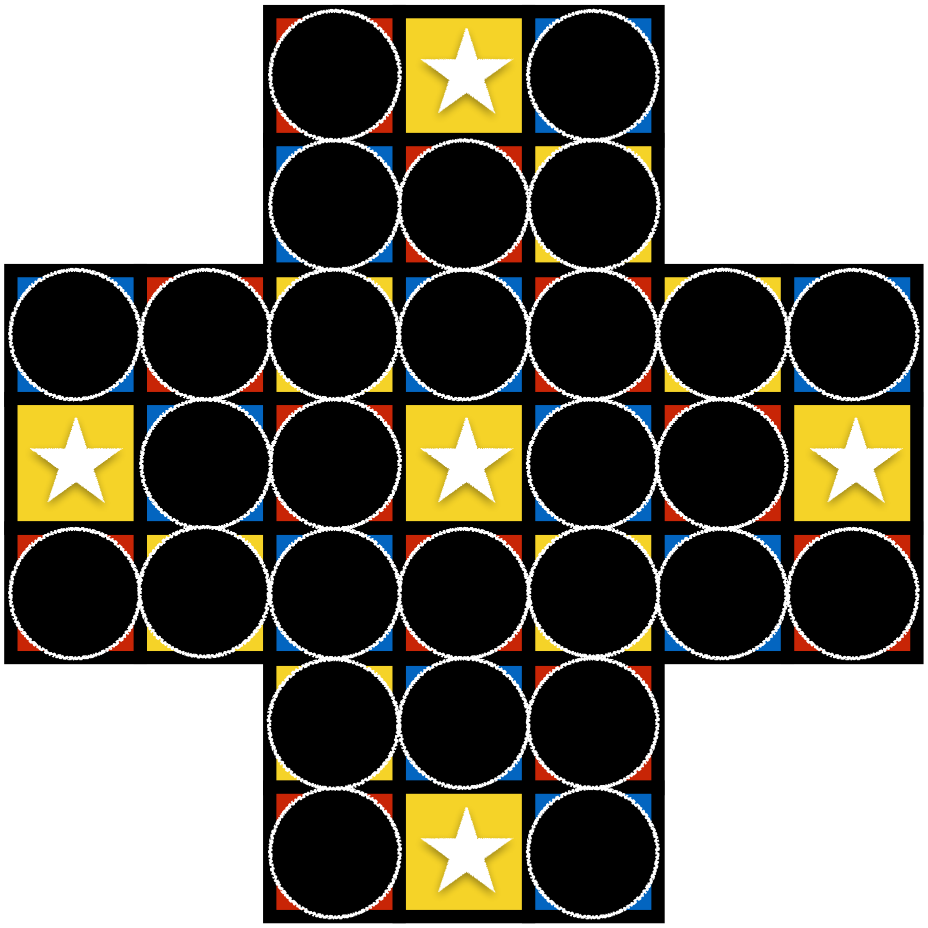
$$\text{Parity}(B) = B\%2 = R\%2 = \text{Parity}(R) \neq \text{Parity}(Y) = Y\%2$$

to be reachable from (0,1,0). *<u>However, it is a necessary but not a sufficient condition.</u>* A number of conclusions can be deduced from this observation:

- Any configuration with a single white pixel must be of the form (0,1,0). Thus a single white pixel must always be on a yellow cell to be reachable :

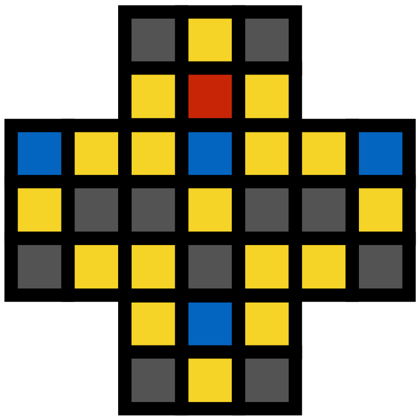
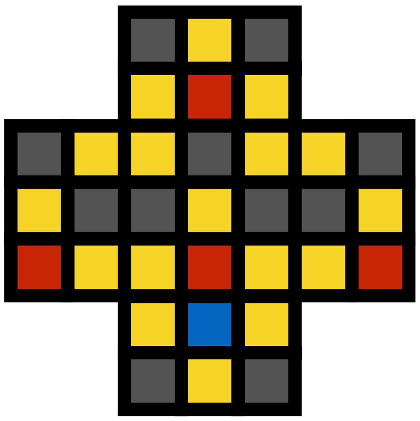


But by symmetry, the same can be argued about the (horizontally or vertically) flipped 3-coloured board. So single white pixels must also be on a yellow cell of the flipped board:



Only 5 cells (marked by stars) are yellow in both the original colouring and the flipped one. These are the only 5 configurations with a single white pixel that are reachable from the *solved* state.

- By the same argument, any configuration with 32 white pixels must be of the form (11, 10, 11). Thus the single black pixel must be on a starred yellow cell as well. All 5 such configurations are reachable and no more.
- Any state with 2 white (/ 2 black) pixels must be of type (1, 0, 1) (/ (10, 11, 10)). Thus the 2 white/black pixels must be on a blue and a red cell. This is also true if we flip the board vertically, horizontally or both. This is possible in only a few places. The yellow cells below indicate cells that are yellow in one of the (flipped) boards.

#white #white on blue,yellow,red (*** necessary but not sufficient condition ***)

1	(0,1,0)				
2	(1,0,1)				
3	(0,1,2)	(0,3,0)			
	(2,1,0)				
4	(1,0,3)	(1,2,1)			
	(3,0,1)				
5	(0,1,4)	(0,3,2)	(0,5,0)-(*** no such configuration reachable ***)		
	(2,1,2)	(2,3,0)			
	(4,1,0)				
6	(1,0,5)	(1,2,3)	(1,4,1)		
	(3,0,3)	(3,2,1)			
	(5,0,1)				
7	(0,1,6)	(0,3,4)	(0,5,2)	(0,7,0)	
	(2,1,4)	(2,3,2)	(2,5,0)		
	(4,1,2)	(4,3,0)			
	(6,1,0)				
8	(1,0,7)	(1,2,5)	(1,4,3)	(1,6,1)	
	(3,0,5)	(3,2,3)	(3,4,1)		
	(5,0,3)	(5,2,1)			
	(7,0,1)				
9	(0,1,8)	(0,3,6)	(0,5,4)	(0,7,2)	(0,9,0)
	(2,1,6)	(2,3,4)	(2,5,2)	(2,7,1)	
	(4,1,4)	(4,3,2)	(4,5,0)		
	(6,1,2)	(6,3,0)			
	(8,1,0)				
…	…				
16	(1,4,11)	(1,6,9)	(1,8,7)	(1,10,5)	
	(3,2,11)	(3,4,9)	(3,6,7)	(3,8,5)	(3,10,3)
	(5,0,11)	(5,2,9)	(5,4,7)	(5,6,5)	(5,8,3)
					(5,10,1)
	(7,0,9)	(7,2,7)	(7,4,5)	(7,6,3)	(7,8,1)
	(9,0,7)	(9,2,5)	(9,4,3)	(9,6,1)	
	(11,0,5)	(11,2,3)	(11,4,1)		
…	…				
28	(7,10,11)				
	(9,8,11)	(9,10,9)			
	(11,6,11)	(11,8,9)	(11,10,7)		
29	(8,11,10)				
	(10,9,10)	(10,11,8)			
30	(9,10,11)				
	(11,8,11)	(11,10,9)			
31	(10,11,10)				
32	(11,10,11)				