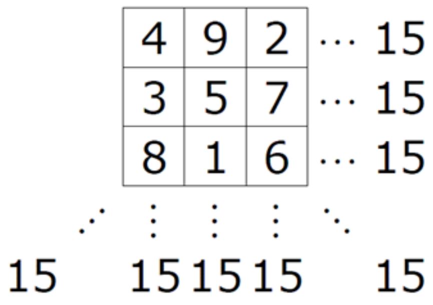
Program Set #1
Total Points: 30

Three problems must be implemented for full credit. The required (starred) problem marked in the set must be implemented by everyone. See the 2436 Grading and Submission Guide Sheets for additional grading/submission information. Partial credit will be given. (10 points each)

Section One- Choose two problems.

1. A magic square is a 3x3 square where the numbers 1-9 fill up each tile, and each row, column, and diagonal add up to 15. The table below shows a magic square.



Notice, if you take any three numbers that make up a row, column, or diagonal, the sum is always 15. Unfortunately, the pattern shown above is the only way to make it perfect. Write a C++ program to count how many rows, columns, and diagonals that need to be fixed before it can be a magic square. The user will enter from the keyboard the nine values in the magic square. Assume proper size. Output to the screen the magic square and the row (R), column (C) and diagonal (D) locations missing to make the input a magic square. If none are missing, output "NONE". Assume the row and columns subscripts start at 1. Finally, the program should ask if the user wants to run the program again (Check case). Refer to the sample output below.

Sample Run:

```
Enter the 3 by 3 magic square: 5 6 4 7 8 9 3 1 2

Magic Square:

5 6 4
7 8 9
3 1 2

Missing: R2 R3

Run Again (Y/N): y

Enter the 3 by 3 magic square: 4 1 7 3 9 6 8 5 2

Magic Square:
```

```
4 1 7
3 9 6
8 5 2

Missing: R1 R2 D2

Run Again (Y/N): N
```

Name the program: `CheckMagicXX.cpp`, where XX are your initials.

2. The game Diffy is based upon Ducci sequences. To play, draw a square and write down four integers at its corners. Then, on the middle of each side, write the positive difference of the two numbers at the corners and connect them to form a smaller square. This process continues until a square is produced with value 0 at all four corners. Begin with four integers at the corners where the largest value on any corner has absolute value `M`, then one can arrive at a box where every corner is 0 in at most `M + 3` steps.

Write a C++ program that counts the number of steps it takes to arrive at a square with 0 at each corner. Input a from the keyboard a single line consisting of four space-separated integers which are the corner starting numbers in clockwise order. The absolute value of each integer is at most 100,000. Assume proper input. Output to the screen the values of the corners after each step and the number of steps it takes to get from the original square to a square where every corner is 0. Finally, the program should ask if the user wants to run the program again (Check case). Refer to the sample output below.

Sample Run:

```
Enter 4 integer numbers clockwise: 6 -2 3 5

The values for each of the corners:

8 5 2 1
3 3 1 7
0 2 6 4
2 4 2 4
2 2 2 2
0 0 0 0

Number of steps: 6

Run Again (Y/N): n
```

Name the program: `DiffyGameXX.cpp`, where XX are your initials.

3. Write a C++ program given a full scorecard calculates the final score of a bowling match. The program reads values for each frame and then prints out all 10 scored frames in a tabular format. In frames (1 to

9) the bowler will roll once or twice, trying to knock down all 10 pins. When all the pins are knocked down in the first throw the frame will store an 'X' and the bowler will not throw a second ball. When pins are still standing after the first throw, the bowler will throw a second time. If no pins are knocked down in a throw a '-' is added to the frame, and when a bowler knocks down some of the standing pins in a throw, the number of pins knocked down by that throw is added to the frame. If all the pins are knocked down by the second throw a '/' is added to that frame. Some examples:

Frame Data	**First Throw**	**Second Throw**
`--`	0 pins	0 pins
`X`	10 pins	
`6-`	6 pins	0 pins
`-5`	0 pins	5 pins
`-/`	0 pins	10 pins
`4/`	4 pins	6 pins

The final frame in bowling is different than the rest. If a bowler gets a strike on their first throw or a spare on their second throw, they get a total of three throws in the final frame. After each throw, if there are no pins standing, all the pins are stood back up for the next throw. Also strikes and spares in frame do not receive bonus points for future throws. Some examples:

Frame Data	**First Throw**	**Second Throw**	**Third Throw**
`--`	0 pins	0 pins	0 pins
`X--`	10 pins	0 pins	0 pins
`6-`	6 pins	0 pins	
`-/5`	0 pins	10 pins	5 pints
`XXX`	10 pins	10 pins	10 pins
`X6/`	10 pins	6 pins	4 pins
`X-8`	10 pins	0 pins	8 pins

The score of each frame will be the previous frame's score plus the current frame's value. The value of the current frame is the number of pins knocked down in the frame plus any bonuses to the frame. When a spare is made the frame's value is increased by the number of pins knocked down on the next throw. When a strike is made the frame's value is increased by the number of pins knocked down on the next two throws.

The input from a data file starts with a line containing the number of test games as integer `N` between `[1,10]`. The following `N` lines consist of a single player's full score card from a complete game of bowling. An `X` represents a strike, a / represents a spare, and a – represents no pins knocked down. All inputs will be valid complete games. For each test case, output to the screen the final score, complete with pin count and in a cumulative graphical ASCII format. Use any appropriate data structure. Let the user enter the file name from the keyboard. Refer to the sample output below.

Sample File:

```
4
9/X9/X9/X9/X9/9/9
XXXXXXXXXXXX
9-9-9-9-9-9-9-9-9-9-
81-92/X637-52X-62/X
```

Sample Run:

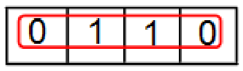
```
Enter the file name: bowling.txt

Game        1    2    3    4    5    6   7    8    9     10
-------------------------------------------------------------
|       | 9|/|  |X| 9|/|  |X| 9|/|  |X| 9|/|  |X| 9|/| 9|/|9|
|       |----|----|----|----|----|----|----|----|----|------|
|Game 1 |  20|  40|  60|  80| 100| 120| 140| 160| 179|   198|
|-----------------------------------------------------------|
|       |  |X|  |X|  |X|  |X|  |X|  |X|  |X|  |X|  |X| X|X|X|
|       |----|----|----|----|----|----|----|----|----|------|
|Game 2 |  30|  60|  90| 120| 150| 180| 210| 240| 270|   300|
|-----------------------------------------------------------|
|       | 9|-| 9|-| 9|-| 9|-| 9|-| 9|-| 9|-| 9|-| 9|-| 9|-| |
|       |----|----|----|----|----|----|----|----|----|------|
|Game 3 |   9|  18|  27|  36|  45|  54|  63|  72|  81|    90|
|-----------------------------------------------------------|
|       | 8|1| -|9| 2|/|  |X| 6|3| 7|-| 5|2|  |X| -|6| 2|/|X|
|       |----|----|----|----|----|----|----|----|----|------|
|Game 4 |   9|  18|  38|  57|  66|  73|  80|  96| 102|   122|
-------------------------------------------------------------
```

Name the program: `BowlingXX.cpp`, where XX are your initials.

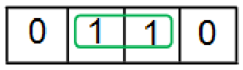
Required Problem- Comprehensive.

4 (**). Write a C++ program that finds the minimal number of groupings in a matrix. The user will enter the size of the rows and columns in the range `[2,5]` from the keyboard. Error check input size. The program then will randomly fill the numbers 0 and 1 into the matrix and output to the screen the matrix and the minimum number of groupings that cover all the 1 values in the matrix. The size of a grouping is defined by the number of 1's in it. Here are the rules in forming groups of ones:

- Each group should contain the largest number of 'ones' and no blank cells.



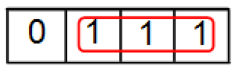
Incorrect



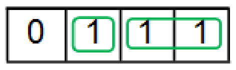
Correct

- The number of 'ones' in a group must be a power of 2 i.e. a group can contain:

$$16(=2^4)\ or\ 8(=2^3)\ or\ 4(=2^2)\ or\ 2(=2^1)\ or\ 1(=2^0)\ cells$$

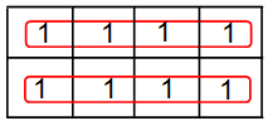


Incorrect

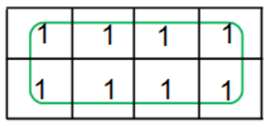


Correct

- Grouping is carried-on in decreasing order meaning, one must try to group for 8 (octet) first, then for 4 (quad), followed by 2 and lastly for 1 (isolated 'ones').

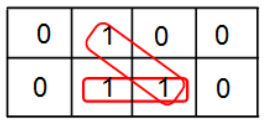


Incorrect

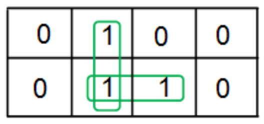


Correct

- Grouping is done either horizontally or vertically or in terms of rectangles/squares. Diagonal grouping of 'ones' is not permitted.

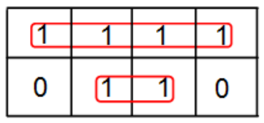


Incorrect

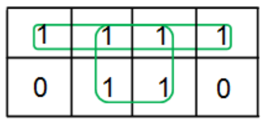


Correct

- The same element(s) may repeat in multiple groups only if this increases the size of the group.

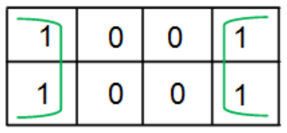


Incorrect



Correct

- The elements around the edges of the matrix, including the four corners, are adjacent and can be grouped together.



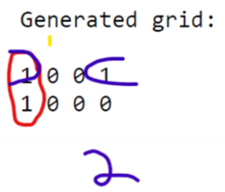
In review:

- No zeros allowed.
- No diagonals.
- Only power of 2 number of cells in each group.
- Groups should be as large as possible.
- Every 1 must be in at least one group.
- Overlapping and wrapping around allowed.
- Get the fewest number of groups possible.

Finally, the program should ask if the user wants to run the program again (Check case). Refer to the sample output below.

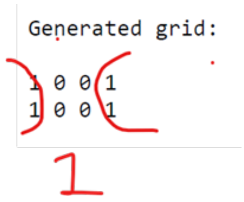
Sample Run:

```
Enter the number of rows (2-5): 2
Enter the number of cols (2-5): 4

 Generated grid:

 1 0 0 1
 1 0 0 0

 The least number of rectangles/squares formed is 2
```



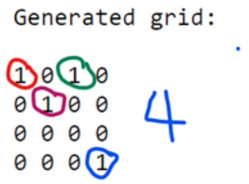
```
Generated grid:

1 0 0 1
1 0 0 0
```

```
Run Again (Y/N): y

Enter the number of rows (2-5): 2
Enter the number of cols (2-5): 4

 Generated grid:

 1 0 0 1
 1 0 0 1

 The least number of rectangles/squares formed is 1
```



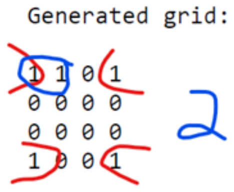
```
Generated grid:

1 0 0 1
1 0 0 1
```

```
Run Again (Y/N): Y
Enter the number of rows (2-5): 4
Enter the number of cols (2-5): 4


 Generated grid:

 1 0 1 0
 0 1 0 0
 0 0 0 0
 0 0 0 1

 The least number of rectangles/squares formed is 4
```



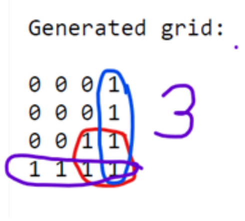
```
Generated grid:

1 0 1 0
0 1 0 0
0 0 0 0
0 0 0 1
```

```
Run Again (Y/N): y

Enter the number of rows (2-5): 4
Enter the number of cols (2-5): 4

 Generated grid:

 1 1 0 1
 0 0 0 0
 0 0 0 0
 1 0 0 1

 The least number of rectangles/squares formed is 2
```



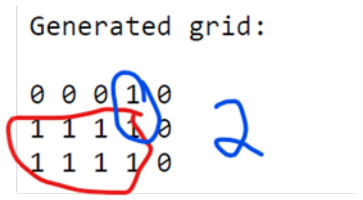
```
Generated grid:

1 1 0 1
0 0 0 0
0 0 0 0
1 0 0 1
```

```
Run Again (Y/N): Y

Enter the number of rows (2-5): 4
Enter the number of cols (2-5): 4

 Generated grid:

 0 0 0 1
 0 0 0 1
 0 0 1 1
 1 1 1 1

 The least number of rectangles/squares formed is 3
```



```
Generated grid:

0 0 0 1
0 0 0 1
0 0 1 1
1 1 1 1
```

```
Run Again (Y/N): y

Enter the number of rows (2-5): 3
Enter the number of cols (2-5): 5

 Generated grid:

 0 0 0 1 0
 1 1 1 1 0
 1 1 1 1 0

 The least number of rectangles/squares formed is 2
```



```
Generated grid:

0 0 0 1 0
1 1 1 1 0
1 1 1 1 0
```

```
Run Again (Y/N): N
```

Name the program: `MinRectanglesXX.cpp`, where XX are your initials.

Extra Credit: Implement the following problem. See the 2436 Grading and Submission Guide Sheets for additional grading/submission information. Partial credit will be given. (10 points)

You have been commissioned to build a fence around a community of houses. The three stipulations are: the fence should be small as possible, it should be rectangular, and it should enclose all houses. Write a C++ program to find the smallest fence size possible using the above stipulations. Input will be from a data file. The first line contains an integer N, the number of cases. Each case will first consist of a line with two integers; the first defines the number of rows of the input map, and the second defines the number of columns. The input map will consist of the '.' character representing an empty space and the 'X' character representing a house. Each character map will have at least one X. For each labeled case, output to the screen the number of rows and columns of the smallest rectangle that encloses all the houses. The number of rows should be first, then the number of columns, separated by a space. Let the user enter the file name from the keyboard. Use any appropriate data structure. Refer to the sample output below.

Sample File:

```
3
4 6
......
.X.X..
....X.
..XX..
8 8
........
...XX...
..X...X.
.....X..
.X.XX...
..XX.X..
...X....
........
1 1
X
```

Sample Run:

```
Enter the file name: fence.txt

Case 1: 3 4
Case 2: 6 6
Case 3: 1 1
```

Name the program: `FencingXX.cpp`, where XX are your initials.