Consortium for Computing Sciences in Colleges Northeastern Region
2003 Undergraduate Programming Contest

Sponsored by Microsoft Corporation
Rhode Island College, Providence, RI
Friday, April 25, 2003

Guidelines and notes

1. The contest will begin after the welcoming remarks conclude (some time between 9:00am and 9:30am) and will run continuously until 12:00noon. Breakfast will be available at about 8:00am, and lunch will be available at about 12:00noon.

2. Each team consists of up to three undergraduates currently enrolled at the same institution.

3. Each team will be randomly assigned a single computer in the computer lab of Horace Mann Technology Center. Teams may use C, C++, or Java to write solutions to the problems.

4. The contest will consist of five problems. The team that correctly solves the most problems wins. If more than one team correctly solves the same number of problems, placement will be determined by the total time to complete them, i.e., the sum of the solution times for each solved problem. For example, if the contest begins at 9:30am and a team solves one problem at 10:00am and another problem at 11:00am, its total time is 120 minutes since it took 30 minutes for the first problem and 90 minutes for the second problem.

5. Teams may use any printed materials during the contest but may **not** use any electronic devices (e.g., laptop computers, calculators, or PDAs) or any materials stored electronically.

6. Teams may print out code. Your source code should include a comment at the beginning indicating your school/team's name. **Your printouts will be delivered to you**.

7. Submitted "solutions" will be sent to the judges via email. The judges will mark them as accepted or rejected. The judges may indicate one of the following reasons for rejection: program not found, compilation error, run-time error, running time limit exceeded, incorrect output, or incorrect output format. Teams may submit "solutions" to the same problem multiple times (within reason) without penalty.

8. A public scoreboard will be maintained. To maintain competition and suspense, however, it will not be updated during the last 45 minutes of the contest. It may be removed earlier at the discretion of the judges.

9. Contestants may not communicate with faculty advisors or other teams during the contest. All questions should be directed to contest officials only. For any disputes, the decision of the judges will be final.

10. For each problem, you may assume that the input is both legal and properly formatted. Hence, no error checking is required.

1. Magic Square

A magic square is an *n* x *n* array of integers in which the numbers in each row, each column, and both diagonals add up to the same value. Normally, each integer is unique, but we will ignore that requirement for this problem. Write a program named **ismagic** that takes from standard input a description of a square array and outputs the word **yes** if the array forms a magic square and outputs **no** otherwise. *[Clarification: n can be as large as 100.]*

Input: Your program should accept the square array from standard input. The first line will specify the number of rows (and columns) in the array. The remaining lines will specify each row of the array, with integers separated by single spaces.

Output: Your program should output **`yes`** or **`no`**, depending on whether the square array is a magic square.

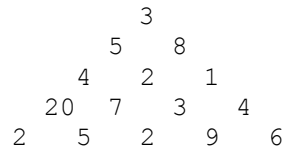
Sample runs:

```
$ ismagic
? 2
? 9 4
? 5 1
no
```

```
$ ismagic
? 3
? 8 1 6
? 3 5 7
? 4 9 2
yes
```

```
$ ismagic
? 3
? 8 1 6
? 3 0 7
? 4 9 2
no
```

2. Triangle Sum



The figure above shows a triangle of numbers. Write a program named **trianglesum** that calculates the maximum sum of the numbers encountered on a path from the top to the bottom. Each path begins at the single value in the top row and ends at one of the values in the bottom row. Each step in the path can go one row down either diagonally to the left or diagonally to the right. For example, one path is 3→5→2→7→5, and another path is 3→8→2→3→9. You can assume that the triangle will contain at least 1 row and up to 200 rows. You can assume that each number in the triangle is an integer between 0 and 100, inclusive.

Input: Your program should accept the triangle representation from standard input. The first line will specify the number of rows in the triangle. The remaining lines will specify each row of the triangle, with integers separated by single spaces.

Output: Your program should output a single number indicating the maximum sum that can be obtained on a path from the top to the bottom of the triangle.

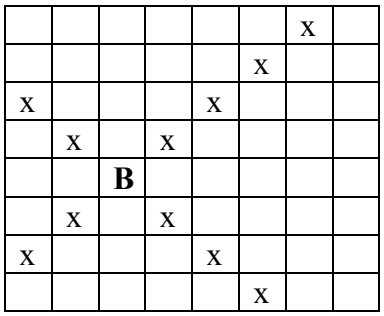
Sample runs:

```
$ trianglesum
? 2
? 1
? 10 20
21
```

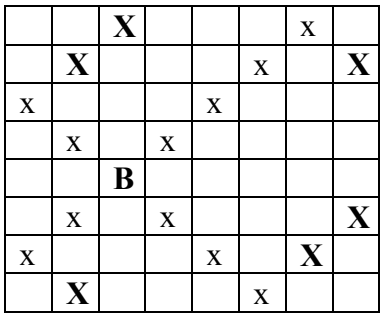
```
$ trianglesum
? 5
? 3
? 5 8
? 4 2 1
? 20 7 3 4
? 2 5 2 9 6
37
```

3. The Bouncing Bishop

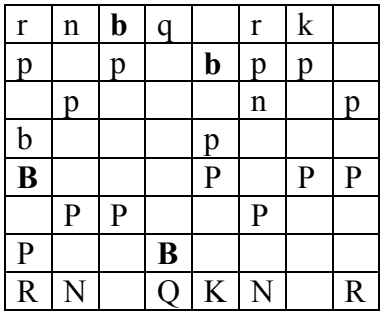
The game of chess is played on an 8 by 8 board by two opposing players, one with black pieces and the other with white pieces. One piece is called the bishop, and it can move diagonally in any direction. For example, in the board to the right, the bishop is indicated by the **B**, and it can move to any of the spots marked with an x. Of course, if another piece occupies a square, the bishop can not move through that piece. However, the bishop could capture it if the piece were an opposing piece.



Consider a new piece, the "bouncing bishop". This piece moves diagonally as the bishop, but it can also bounce off a side once in a move. Its bounce is the reflection of the way it came. For example, in the board to the right, the bouncing bishop is indicated by the **B**, and it can move to any one of the spots marked with an x or **X**. The bold capital **X**'s indicate sample paths after a "bounce" has taken place. You should write a program named **bishops** that determines what pieces the bouncing bishop(s) for the side to move can capture.



Assume that each side may have at most two *[correction: due to an error in the diagram below, we changed this to <u>three</u>]* bouncing bishops and no regular bishops. It is possible to represent a position in any game with eight lines of text. Each line describes one row of the board and uses the letters B, K, N, P, Q, and R to represent a bouncing White bishop, a White King, a White Knight, a White Pawn, a White Queen, and a White Rook, respectively (there are no other piece types). The lower case equivalents represent Black pieces. (The rules of chess and the way the other pieces move are not relevant to this problem.) Numbers are used to indicate consecutive empty places. For example, the position below



can be represented as:

```
rnbq1rk1
p1p1bpp1
1p3n1p
b3p3
B3P1PP
1PP2P2
P2B4
RN1QKN1R
```

(**The Bouncing Bishop** continued)

Input: Your program should accept from standard input 8 lines of text describing a chess position. It should accept a ninth line of text containing the letter **`W`** or the letter **`B`**, which indicates that it is White's turn to move or Black's turn to move, respectively.

Output: Your program should output on one line the letters of all the pieces that the side to move's bishop(s) can capture. If there are no pieces that can be captured, the program must print the word **`none`**. The list may appear in any order. Note that if there are two or more of any piece type that can be captured, the letter appears once for each piece.

Sample Runs:

```
$ bishops
? rnbq1rk1
? p1p1bpp1
? 1p3n1p
? b3p3
? B3P1PP
? 1PP2P2
? P2B4
? RN1QKN1R
? W
pp
```

```
$ bishops
? 1k6
? 6q1
? 1p6
? 8
? 8
? 4B3
? 7R
? 1K6
? W
pq
```

4. Cellular Automata

Consider a bit string of 19 bits that evolves over a sequence of generations. The values of the bits at any generation *i* are determined by the values of the bits at generation *i-1* and a set of transformations, which together are called a rule. We refer to each bit value with $x_{i,j}$, where *i* is the generation number and *j* is the position number, from 18 (leftmost) to 0 (rightmost). The value of $x_{i,j}$ is determined by the 3-bit string from the previous generation created by concatenating $x_{i-1,(j+1) mod 19}$, $x_{i-1,j}$, and $x_{i-1,(j-1) mod 19}$. Since there are 8 different 3-bit strings (111, 110, 101, …, 000), a rule consists of 8 transformations, each of which maps a 3-bit string to either 0 or 1. A rule can be concisely specified by an 8-bit value that simply represents the appropriate outputs for the strings 111, 110, 101, 100, 011, 010, 001, and 000, in that order. For this problem, assume that the bits in generation 1 always consist of nine 0's followed by a 1, followed by nine more 0's. Your job is to write a program named **generations** that accepts from standard input a rule number and a number of generations, and then outputs the state of each generation using that rule.

Rule Example: Suppose the rule is 163, which is equivalent to 10100011 in base 2. This means that the transformations are:

3-bits in previous generation	**what the bit should be in new generation**
111	1
110	0
101	1
100	0
011	0
010	0
001	1
000	1

The first generation is `0000000001000000000`
The second generation is `1111111110011111111`
(000→1,001→1,010→0, and 100→0)
The third generation is `1111111100101111111`
(000→1,110→0,100→0, 001→1, 011→0)

(**Cellular Automata** continued)

Input: Your program should accept from standard input two values, which we'll call **R** and **G**. **R** should be an integer between 0 and 255, inclusive, corresponding to a rule, as described above. **G** should be a positive integer indicating a number of generations.

Output: Your program should output the bit string for each generation on a new line, from generation 1 up to and including generation **G**. Each bit that is on (1) should be displayed with an `X` character, and each bit that is off (0) should be displayed with a `.` (period) character. (Remember, the bits in generation 1 always consist of nine 0's followed by a 1, followed by nine more 0's.)

Sample run:

```
$ generations
? 30
? 8
.........X.........
........XXX........
.......XX..X.......
......XX.XXXX......
.....XX..X...X.....
....XX.XXXX.XXX....
...XX..X....X..X...
..XX.XXXX..XXXXXX..
```

(Here, 30 is 00011110 in binary. Thus, 100, 011, 010, 001 transform to 1, while 111, 110, 101, and 000 transform to 0.)

```
$ generations
? 145
? 8
.........X.........
XXXXXXXX..XXXXXXXXX
XXXXXXX.X..XXXXXXXX
XXXXXX...X..XXXXXXX
XXXXX.XX..X..XXXXXX
XXXX....X..X..XXXXX
XXX.XXX..X..X..XXXX
XX...X.X..X..X..XXX
```

5. Finding the Shortest Path

Write a program named **bestpath** that finds the shortest path (in terms of total distance traveled) between two points, **S** and **E**, on a 2-D grid that has rectangular obstacles. The rectangles will not overlap or touch each other in any way. The path from **S** to **E** may include travel along the edges of the rectangles, but should never pass through any rectangle. You will be provided the start point **S**, the end point **E**, and the points making up each rectangle in the grid. Each point will be specified by a pair of integers representing the *x* and *y* coordinates of that point. Each rectangle will be specified by a list of four integers – the first two represent the *x* and *y* coordinates of the rectangle's upper-left corner, while the next two represent the *x* and *y* coordinates of the rectangle's lower-right corner. You can assume that **S** and **E** are not contained in any rectangle, and that all coordinate values are pairs of integers, each between 0 and 1000, inclusive.

Input: Your program should read from standard input a description of the 2-D scene. The first and second lines will specify the *x* and *y* coordinates of the start point and end point, respectively. The third line will specify the number of rectangles in the scene. Each subsequent line will specify a rectangle with four integers, as described above. All integers will be separated by single spaces.

Output: A sequence of points, each on its own line, that specifies the shortest path from the start point **S** to the end point **E**. If there isn't a unique shortest path, you may specify any one of the shortest paths. *[Clarification: You should print out every point along the solution path that is a corner of one of the rectangles in the scene. You should not print out any point that is not a corner of one of the rectangles in the scene.]*

Sample run (with corresponding diagram):

```
$ bestpath
? 0 9
? 8 2
? 3
? 1 8 4 5
? 2 4 5 1
? 6 9 9 3
0 9
4 8
6 3
8 2
```