You can form a project group of 2 to 3 persons to solve the problem in project.

Project 5 - Map

(Credit: The following questions come from the School of Computing)

The descriptions are based on the compiler gcc and the program runs on the UNIX system. You can run your program on the PC system.

Problem Statement

Given an *m*x*n* map, perform several kinds of operations on the map, including map-filling operation, map-touching operations and drawing of borders for certain values in the map.

A **map** is defined as an *m*x*n* integer array. Following is a 6x6 map, with headers along each row and each column.

```
    0  1  2  3  4  5*
 +------------------*
0|  0  3  9  9  9  9*
1|  0  3  9  5  5  9*
2|  7  7  7  7  7  9*
3|  7  7  7  7  7  9*
4|  7  7  7  7  7  9*
5|  0  3  3  5  5  3*
```

(A character '*' at the end of each line indicates the last character in that line during printing.) A **coordinate** is a pair of integers indicating a location in the map. In the above map, at location (5,0), the cell value is 0; at location (0,5), the cell value is 9. A coordinate is defined as a structure of type:

```
typedef struct {
     int x;
     int y;
} coord_t;
```

Thus, location (5,0) has the coordinate data with data members x = 5 and y = 0, and location (0,5) has the coordinate data with data members x = 0 and y = 5.

Tasks:

Accept the skeleton program map_skeleton.c as the base for constructing your own program. **Do not overwrite this skeleton program**. Otherwise, you can get yourself into deep trouble if you need to re-develop your program from start due to some unforeseen circumstances.

Study the skeleton program carefully. It contains an **operation loop**, in which you can repetitively enter some specific number to perform map (and related) operations, mainly over the most recently updated map.

Compile the skeleton program, and execute it. Follow the sample runs below to see the results. (user input are in **blue and in bold**.) Series of input can be gathered into an "input file", which is presented here for your convenience. You can also execute your program by redirecting the input file to your executable code, as follows:

```
a.out < input_file_name
```

Sample Run 1

Input file	Run
-2	$ a.out Enter operation number [0 to 4]; other integer to get out: **-2** The End.

Sample Run 2

Input file	Run
0 1 2 3 4 5	$ a.out Enter operation number [0 to 4]; other integer to get out: **0** Print the current map: ++++++++++++++++++++ Enter operation number [0 to 4]; other integer to get out: **1** Find corners.

```
++++++++++++

Enter operation number [0 to 4]; other integer to get out: 2

Map Filling Operation.

++++++++++++++++++++

Enter operation number [0 to 4]; other integer to get out: 3

Display map borders.

++++++++++++++++++

Enter operation number [0 to 4]; other integer to get out: 4

Map Touching Operation.

++++++++++++++++++++

Enter operation number [0 to 4]; other integer to get out: 5

The End.
```

Important Advice

1. You are advised to **develop your program from level 1 up to level 5, in that order**, and not jump randomly from one level to another. The test data at level *k* are designed such that you need to pass those levels lower than *k* in order to test the current level.

2. You are also advised **not to modify the main function of your program**, which you obtain from the skeleton program. In testing your code, we will strip off your main function, and tie our in-house main function with the rest of your code. So, any changes made to the main function will be ineffective.

To proceed to the next level (say level 1), copy your program by typing the Unix command

```
cp map_skeleton.c map1.c
```

♦ ♦ ♦ ♦ ♦

This level has two sub-tasks:

1. Modify `map1.c` to construct a map. The map is constructed before the operation number is read. Reading data to construct a map is performed by `readMap`, which has the following prototype.

```
2.            void readMap(int map[][N], int *m, int *n);
```

It will read two positive integers *m* and *n* -- which are at most 50. They form the maximum number of rows and columns of the map respectively. It then reads in *m*x*n* number of non-negative integers (of range between 0 and 99 both inclusive) to fill in the map contents. Lastly, it calls a procedure `prtMap` to display the map on the screen. All these are done before the operation number is read.

The procedure to print the map is:

```
    void prtMap(int map[][N], int m, int n);
```

3. In addition, complete the task for operation number 0, which simply prints out the current map.

Following is a sample run (user input are in **blue and in bold**).Take note the printing format of the map.

Sample Run 1

Input file	Run
3 3 1 2 3 1 2 3 1 2 3 8	$ a.out Read in number of rows: **3** Read in number of columns: **3** Read in the map of size 3 X 3. **1 2 3** **1 2 3** **1 2 3**

```
     0  1  2*
  +---------*
 0|  1  2  3*
 1|  1  2  3*
 2|  1  2  3*
Enter operation number [0 to 4]; other integer to get out: 8
The End.
```

Sample Run 2

Input file	Run

```
3
3
1 2 3
1 2 3
1 2 3
0
8
```

```
$ a.out
Read in number of rows: 3
Read in number of columns: 3
Read in the map of size 3 X 3.
1 2 3
1 2 3
1 2 3
     0  1  2*
  +---------*
 0|  1  2  3*
 1|  1  2  3*
 2|  1  2  3*
Enter operation number [0 to 4]; other integer to get out: 0
Print the current map:
++++++++++++++++++++++
     0  1  2*
  +---------*
 0|  1  2  3*
 1|  1  2  3*
 2|  1  2  3*
```

	Enter operation number [0 to 4]; other integer to get out: 8 The End.

Click here to submit to CodeCrunch.

A sample test case is provided for you to test for format correctness using the `diff` command:

```
./a.out < map1.in | diff - map1.out
```

To proceed to the next level (say level 2), copy your program by typing the Unix command

```
cp map1.c map2.c
```

♦ ♦ ♦ ♦ ♦

In this level, we complete the task for operation number **1** which is to determine the coordinates of the four corners of a rectangular region in the map.

At this juncture, your program has already read in a map (which is the task of Level 1). It now reads in two coordinates, (x1,y1) and (x2,y2) which represent the two **diagonally opposite corners** of a rectangular region within the map. With these two coordinates, your program will compute and display the four corners of the rectangular region, ordered as such: top-left, top-right, bottom-left and bottom-right.

For instance, consider the following 5x5 map.

```
    0  1  2  3  4
  +---------------
0|  1  3  5  7  9
 |
1|  2  4  6  8  0
 |
2|  1  3  5  7  9
 |
3|  2  4  6  8  0
 |
4|  1  3  5  7  9
```

If the two diagonally opposite coordinates given are (1,1) and (3,4), they define the following region (enclosed by dotted lines, with the two corresponding corners displayed in red and in bold).

```
    0  1  2  3  4
  +---------------
0|  1  3  5  7  9
 |    *.........*
1|  2 :4  6  8  0:
 |    :          :
2|  1 :3  5  7  9:
 |    :          :
3|  2 :4  6  8  0:
 |    *.........*
4|  1  3  5  7  9
```

For this region, the top-left corner has coordinate (1,1), the top-right corner is (1,4), bottom-left corner is (3,1), and bottom-right, (3,4).

The same region can be specified by their diagonally opposite coordinates in four different ways:

- the pair (1,1) and (3,4)
- the pair (3,4) and (1,1)
- the pair (3,1) and (1,4)
- the pair (1,4) and (3,1)

As such, your program must be able to take in whichever one of these four pairs of coordinates and return the corresponding four corners of the specified rectangular region.

In addition, if the two input coordinates share either the `x` or `y` data elements, then the specified region degenerates into either part of a row or a column.

Moreover, if the two input coordinates are the same (n,n), then it refers to a degenerated rectangular region containing just one element at coordinate (n,n).

One last point, you can **safely assume that the coordinates entered to form the corners are always within the boundary of the map**, and there is thus no need to perform out-of-bounds checks.

Write a procedure `read2Corners` to read two coordinates defining the rectangular region.

```
void read2Corners(coord_t *c1, coord_t *c2);
```

The two coordinates read will be stored in `c1` and `c2` respectively.

Next, write a procedure `findCorners` to compute all four corners of the rectangular region defined by two input coordinates. It has the following prototype.

```
void findCorners(coord_t c1, coord_t c2, coord_t corners[4]);
```

The four corners together are represented by an array of four coordinates (`coord_t corner[4]`), with the top-left corner having array index 0, top-right corner has index 1, bottom-left: 2, and bottom-right: 3. These corners will then be displayed in the output. Please refer to the sample run for details.

The following partial `printf` statement shows the format specifier for displaying the top-left corner:

```
    printf("Top-left Corner: %d, %d.n",  ...);
```

Following is a sample run (user input are in **blue and in bold**):

Sample Run 1	Perform operation number 1 two times
Input file	Run

Input file:

```
5

5

1 2 3 4 5

9 8 7 6 5

8 7 6 5 4

6 7 8 9 1

1 2 3 4 5

1

4 1

1 4

1

1 0

1 4

7
```

Run:

```
$ a.out

Read in number of rows: 5

Read in number of columns: 5

Read in the map of size 5 X 5.

1 2 3 4 5

9 8 7 6 5

8 7 6 5 4

6 7 8 9 1

1 2 3 4 5

     0  1  2  3  4*

  +---------------*

 0|  1  2  3  4  5*

 1|  9  8  7  6  5*

 2|  8  7  6  5  4*

 3|  6  7  8  9  1*

 4|  1  2  3  4  5*

Enter operation number [0 to 4]; other integer to get out: 1

Find corners.

++++++++++++

Please enter the first corner: 4 1

Please enter the corresponding diagonal corner: 1 4
```

```
Top-left Corner: 1, 1.

Top-right Corner: 1, 4.

Bottom-left Corner: 4, 1.

Bottom-right Corner: 4, 4.

Enter operation number [0 to 4]; other integer to get out: 1

Find corners.

++++++++++++

Please enter the first corner: 1 0

Please enter the corresponding diagonal corner: 1 4

Top-left Corner: 1, 0.

Top-right Corner: 1, 4.

Bottom-left Corner: 1, 0.

Bottom-right Corner: 1, 4.

Enter operation number [0 to 4]; other integer to get out: 7

The End.
```

Click here to submit to CodeCrunch.

A sample test case is provided for you to test for format correctness using the `diff` command:

```
./a.out < map2.in | diff - map2.out
```

To proceed to the next level (say level 3), copy your program by typing the Unix command

```
cp map2.c map3.c
```

♦ ♦ ♦ ♦ ♦

At this level, we complete the task for operation number **2**, which performs a Map-filling Operation. The steps involved are as follows:

Read in two coordinates to define a rectangular region in the map. Then, read in a number `n` such that *0* ≤ `n` ≤ *99*, and fill up the rectangular region with `n`.

As an example, suppose the current map is as follows:

```
     0  1  2  3  4
   +---------------
  0|  1  3  5  7  9
   |
  1|  2  4  6  8  0
   |
  2|  1  3  5  7  9
   |
  3|  2  4  6  8  0
   |
  4|  1  3  5  7  9
```

your input can be the following (comments about the input are provided to the right):

```
Input:        2        // operation number
     3 3      // first coordinate defining the region
     1 1      // second coordinate defining the region
        8     // number to be filled in
```

The updated map will be as follows (where changes have been highlighted in red):

```
      0  1  2  3  4
   +---------------
 0|  1  3  5  7  9
  |
 1|  2  8  8  8  0
  |
 2|  1  8  8  8  0
  |
 3|  2  8  8  8  0
  |
 4|  1  3  5  7  9
```

Write a procedure `fillSpace` with the following prototype to perform this map-filling operation.

```
void fillSpace(int map[][N], int rows, int cols);
```

This procedure will read in the necessary input and perform the map-filling task. However, it will **not** be displaying the operation result. As you will observe in the sample run below, we will delegate the map printing task to operation number 0.

Following are some sample runs (user input are in **blue and in bold**).

Perform map-filling operation two times

Sample Run 1

Input file	Run
5 5 0 2 4 6 8 1 3 5 7 9 2 4 6 8 0 5 7 9 1 3	$ a.out Read in number of rows: **5** Read in number of columns: **5** Read in the map of size 5 X 5. **0 2 4 6 8** **1 3 5 7 9**

```
0 1 2 5 0

2

2 1

4 3

6

0

-1
```

```
2 4 6 8 0

5 7 9 1 3

0 1 2 5 0

     0  1  2  3  4*

  +---------------*

 0|  0  2  4  6  8*

 1|  1  3  5  7  9*

 2|  2  4  6  8  0*

 3|  5  7  9  1  3*

 4|  0  1  2  5  0*

Enter operation number [0 to 4]; other integer to get out: 2

Map Filling Operation.

++++++++++++++++++++

Please enter the first corner: 2 1

Please enter the corresponding diagonal corner: 4 3

Top-left Corner: 2, 1.

Top-right Corner: 2, 3.

Bottom-left Corner: 4, 1.

Bottom-right Corner: 4, 3.

Enter number to be filled in: 6

Enter operation number [0 to 4]; other integer to get out: 0

Print the current map:

++++++++++++++++++++

     0  1  2  3  4*

  +---------------*

 0|  0  2  4  6  8*

 1|  1  3  5  7  9*

 2|  2  6  6  6  0*

 3|  5  6  6  6  3*

 4|  0  6  6  6  0*
```

```
Enter operation number [0 to 4]; other integer to get out: -1

The End.
```

Sample Run 2

Input file	Run

```
5

5

0 2 4 6 8

1 3 5 7 9

2 4 6 8 0

5 7 9 1 3

0 1 2 5 0

2

2 1

4 3

2

4 2

0 2

6

0

-1
```

```
$ a.out

Read in number of rows: 5

Read in number of columns: 5

Read in the map of size 5 X 5.

0 2 4 6 8

1 3 5 7 9

2 4 6 8 0

5 7 9 1 3

0 1 2 5 0

     0  1  2  3  4*

  +--------------*

 0|  0  2  4  6  8*

 1|  1  3  5  7  9*

 2|  2  4  6  8  0*

 3|  5  7  9  1  3*

 4|  0  1  2  5  0*

Enter operation number [0 to 4]; other integer to get out: 2

Map Filling Operation.

++++++++++++++++++++

Please enter the first corner: 2 1

Please enter the corresponding diagonal corner: 4 3

Top-left Corner: 2, 1.

Top-right Corner: 2, 3.

Bottom-left Corner: 4, 1.

Bottom-right Corner: 4, 3.
```

```
Enter number to be filled in: 6
Enter operation number [0 to 4]; other integer to get out: 2
Map Filling Operation.
++++++++++++++++++++
Please enter the first corner: 4 2
Please enter the corresponding diagonal corner: 0 2
Top-left Corner: 0, 2.
Top-right Corner: 0, 2.
Bottom-left Corner: 4, 2.
Bottom-right Corner: 4, 2.
Enter number to be filled in: 6
Enter operation number [0 to 4]; other integer to get out: 0
Print the current map:
++++++++++++++++++++
     0  1  2  3  4*
  +---------------*
 0|  0  2  6  6  8*
 1|  1  3  6  7  9*
 2|  2  6  6  6  0*
 3|  5  6  6  6  3*
 4|  0  6  6  6  0*
Enter operation number [0 to 4]; other integer to get out: -1
The End.
```

Click here to submit to CodeCrunch.

A sample test case is provided for you to test for format correctness using the `diff` command:

```
./a.out < map3.in | diff - map3.out
```

To proceed to the next level (say level 4), copy your program by typing the Unix command

```
cp map3.c map4.c
```

♦ ♦ ♦ ♦ ♦

At this level, we complete the task for operation number **3**, which performs **map-value boundary drawing**.

Given a map and a number *n*, the task here is to identify the occurrences of *n* in the map, and draw boundaries on these occurrences such that they are separated from other numbers occurring in the map. For instance, given the following map:

```
   0  1  2  3  4*
  +---------------*
 0|  1  2  3  4  5*
 1|  7  8  9  0  1*
 2|  2  3  4  1  5*
 3|  3  4  1  5  6*
 4|  6  1  1  8  9*
```

Next, given a number, **1** say, your program will draw boundary along the occurrences of **1**, as follows:

```
 --            *
| 1| 2  3  4  5 *
 --         -- *
 7  8  9  0| 1|*
         -- -- *
 2  3  4| 1| 5 *
       -- --   *
 3  4| 1| 5  6 *
   --          *
 6| 1  1| 8  9 *
   -- --       *
```

Note that in displaying a map with borders, we need more rows (ie., lines) and columns than the original map has in order to display the borders as well. Generally, for an $m_X n$ map, we will need *2m+1* rows (lines). If each map value occupies two spaces, we will need *3n+2* spaces to display a line (including the last '*' character).

Write a procedure `drawBorders` which takes in a map and draw the borders to separate the 0s and the 1s. Its prototype is as follows:

```
void drawBorders(int map[][N], int rows, int cols);
```

Following are some sample runs (user input are in **blue and in bold**).

Sample Run 1	Check how borders at corners and edges of map are drawn.
Input file	Run

```
5

5

1 2 3 4 5

7 8 9 0 1

2 3 4 1 5

3 4 1 5 6
```

```
$ a.out

Read in number of rows: 5

Read in number of columns: 5

Read in the map of size 5 X 5.

1 2 3 4 5

7 8 9 0 1
```

```
6 1 7 8 9

3

1

-1
```

```
2 3 4 1 5

3 4 1 5 6

6 1 7 8 9

     0  1  2  3  4*

  +---------------*

 0|  1  2  3  4  5*

 1|  7  8  9  0  1*

 2|  2  3  4  1  5*

 3|  3  4  1  5  6*

 4|  6  1  7  8  9*

Enter operation number [0 to 4]; other integer to get out: 3

Display map borders.

++++++++++++++++++++

Read in a number: 1

 --             *

| 1| 2  3  4  5 *

 --          -- *

  7  8  9  0| 1|*

          -- -- *

  2  3  4| 1| 5 *

       -- --    *

  3  4| 1| 5  6 *

    -- --       *

  6| 1| 7  8  9 *

    --          *

Enter operation number [0 to 4]; other integer to get out: -1

The End.
```

Sample Run 2	No border drawing for non-existing number
Input file	Run

```
5
5
1 2 3 4 5
7 8 9 0 1
2 3 4 1 5
3 4 1 5 6
6 1 1 8 9
3
10
-1
```

```
$ a.out
Read in number of rows: 5
Read in number of columns: 5
Read in the map of size 5 X 5.
1 2 3 4 5
7 8 9 0 1
2 3 4 1 5
3 4 1 5 6
6 1 1 8 9
     0  1  2  3  4*
  +--------------*
 0|  1  2  3  4  5*
 1|  7  8  9  0  1*
 2|  2  3  4  1  5*
 3|  3  4  1  5  6*
 4|  6  1  1  8  9*
Enter operation number [0 to 4]; other integer to get out: 3
Display map borders.
++++++++++++++++++++
Read in a number: 10
                *
  1  2  3  4  5 *
                *
  7  8  9  0  1 *
                *
  2  3  4  1  5 *
                *
  3  4  1  5  6 *
                *
  6  1  1  8  9 *
```

```
                    *

Enter operation number [0 to 4]; other integer to get out: -1

The End.
```

Sample Run 3	Perform map-filling and then border drawing
Input file	Run

```
5

5

1 2 3 4 5

7 8 9 0 1

2 3 4 1 5

3 4 1 5 6

6 1 1 8 9

2

1 0

3 2

5

0

3

5

-1
```

```
$ a.out

Read in number of rows: 5

Read in number of columns: 5

Read in the map of size 5 X 5.

1 2 3 4 5

7 8 9 0 1

2 3 4 1 5

3 4 1 5 6

6 1 1 8 9

     0  1  2  3  4*

  +---------------*

 0|  1  2  3  4  5*

 1|  7  8  9  0  1*

 2|  2  3  4  1  5*

 3|  3  4  1  5  6*

 4|  6  1  1  8  9*

Enter operation number [0 to 4]; other integer to get out: 2

Map Filling Operation.

++++++++++++++++++++

Please enter the first corner: 1 0

Please enter the corresponding diagonal corner: 3 2

Top-left Corner: 1, 0.

Top-right Corner: 1, 2.

Bottom-left Corner: 3, 0.
```

```
Bottom-right Corner: 3, 2.

Enter number to be filled in: 5

Enter operation number [0 to 4]; other integer to get out: 0

Print the current map:

++++++++++++++++++++

     0  1  2  3  4*

  +---------------*

 0|  1  2  3  4  5*

 1|  5  5  5  0  1*

 2|  5  5  5  1  5*

 3|  5  5  5  5  6*

 4|  6  1  1  8  9*

Enter operation number [0 to 4]; other integer to get out: 3

Display map borders.

++++++++++++++++++++

Read in a number: 5

             -- *

  1  2  3  4| 5|*

 -- -- --    -- *

| 5  5  5| 0  1 *

             -- *

| 5  5  5| 1| 5|*

          -- -- *

| 5  5  5  5| 6 *

 -- -- -- --    *

  6  1  1  8  9 *

               *

Enter operation number [0 to 4]; other integer to get out: -1

The End.
```

Click here to submit to CodeCrunch.

A sample test case is provided for you to test for format correctness using the `diff` command:

```
./a.out < map4.in | diff - map4.out
```

To proceed to the next level (say level 5), copy your program by typing the Unix command

```
cp map4.c map5.c
```

♦ ♦ ♦ ♦ ♦

At this level, we complete the task for operation number **4**, which performs **map-touching operation**.

Generally, this operation lets the user touch a location in the map, and specify a number, *v* say. Suppose the current value at that location in the map is *u*, the operation will spread the value *v* to the ***u*-connected cells** within a rectangular region in the map, replacing those cells' value by *v*.

Specifically, this operation is performed when the user enters **4** as the operation number. It first reads in two (diagonal) coordinates from the user to define a rectangular region. Then it takes in two more pieces of information: (a) a coordinate *(i,j)*, assumed to always be within the defined rectangular region, and (b) a number *v*. Here, *(i,j)* is the coordinate of the point where the map is touched with the number *v*. Given that *(i,j)* currently holds a value *u*, the result of this touch is to replace the value at *(i,j)* by *v*, as well as replacing the values by *v* of all those ***u*-connected cells within the rectangular region**.

A cell is said to be ***u*-connected** to *(i,j)* if it satisfies two conditions:

1. It must hold the same value as that for *(i,j)*; ie., value *u*, and
2. It is directly or indirectly adjacent to *(i,j)*.

A location *(a,b)* is directly adjacent to *(i,j)* if it shares an edge border with *(i,j)*. That is, *(a,b)* can be one of the following locations: *(i+1,j)*, *(i-1,j)*, *(i,j+1)* and *(i,j-1)*.

A location *(a,b)* is indirectly adjacent to *(i,j)* if it is directly adjacent to some location which is directly or indirectly adjacent to *(i,j)*.

As an example, consider the following map:

```
    0  1  2  3  4*
 +---------------*
0|  0  1  1  2  1*
1|  1  3  3  0  1*
2|  1  3  1  1  1*
3|  1  1  1  3  3*
4|  2  2  0  2  1*
```

Select the coordinate `(3,1)` to be the location *(i,j)* of interest, notice that `(3,1)` currently has value **1**, then the ***1*-connected cells** in this map are highlighted in red and in bold below.

```
    0  1  2  3  4*
 +---------------*
0|  0  1  1  2  1*
1|  1  3  3  0  1*
2|  1  3  1  1  1*
3|  1  1  1  3  3*
4|  2  2  0  2  1*
```

If now we want to change the value at location `(3,1)` from **1** to **2**, and if the map-touching operation takes effect on the entire map, then the resulting map will be as follows:

```
    0  1  2  3  4*
 +---------------*
0|  0  1  1  2  2*
1|  2  3  3  0  2*
2|  2  3  2  2  2*
3|  2  2  2  3  3*
4|  2  2  0  2  1*
```

However, the map-touching operation only works on a rectangular region. Suppose the region is bounded by `(0,0)` and `(3,3)`, then the ultimate operation result is:

```
    0  1  2  3  4*
 +---------------*
0|  0  1  1  2  1*
1|  2  3  3  0  1*
2|  2  3  2  2  1*
3|  2  2  2  3  3*
4|  2  2  0  2  1*
```

Develop a procedure `touch` with the following prototype to perform this map-touching operation:

```
void touch(int map[][N], int rows, int cols);
```

Following are some sample runs (user input are in **blue and in bold**).

Sample Run 1

Input file	Run
5 5 0 1 1 0 0 1 0 0 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 1 4 5 0	$ a.out Read in number of rows: **5** Read in number of columns: **5** Read in the map of size 5 X 5. **0 1 1 0 0** **1 0 0 0 1** **1 1 1 1 1** **0 0 1 0 0** **0 0 0 0 1** 0 1 2 3 4*

-1	+---------------* 0| 0 1 1 0 0* 1| 1 0 0 0 1* 2| 1 1 1 1 1* 3| 0 0 1 0 0* 4| 0 0 0 0 1* Enter operation number [0 to 4]; other integer to get out: **4** Map Touching Operation. ++++++++++++++++++++++ Please enter the first corner: **0 0** Please enter the corresponding diagonal corner: **4 4** Top-left Corner: 0, 0. Top-right Corner: 0, 4. Bottom-left Corner: 4, 0. Bottom-right Corner: 4, 4. Enter the coordinate of a location to touch: **2 1** Choose a number to spread: **5** Enter operation number [0 to 4]; other integer to get out: **0** Print the current map: ++++++++++++++++++++++ 0 1 2 3 4* +---------------* 0| 0 1 1 0 0* 1| 5 0 0 0 5* 2| 5 5 5 5 5* 3| 0 0 5 0 0* 4| 0 0 0 0 1* Enter operation number [0 to 4]; other integer to get out: **-1** The End.

Sample Run 2	Perform a map-touching operation followed by border drawing
Input file	Run

```
5
5
0 1 1 2 1
1 3 3 0 1
1 3 1 1 1
1 1 1 3 3
2 2 0 2 1
4
0 0
3 3
3 1
2
0
3
2
-1
```

```
$ a.out
Read in number of rows: 5
Read in number of columns: 5
Read in the map of size 5 X 5.
0 1 1 2 1
1 3 3 0 1
1 3 1 1 1
1 1 1 3 3
2 2 0 2 1
     0  1  2  3  4*
  +---------------*
 0|  0  1  1  2  1*
 1|  1  3  3  0  1*
 2|  1  3  1  1  1*
 3|  1  1  1  3  3*
 4|  2  2  0  2  1*
Enter operation number [0 to 4]; other integer to get out: 4
Map Touching Operation.
++++++++++++++++++++++
Please enter the first corner: 0 0
Please enter the corresponding diagonal corner: 3 3
Top-left Corner: 0, 0.
Top-right Corner: 0, 3.
Bottom-left Corner: 3, 0.
Bottom-right Corner: 3, 3.
Enter the coordinate of a location to touch: 3 1
Choose a number to spread: 2
Enter operation number [0 to 4]; other integer to get out: 0
```

```
Print the current map:

++++++++++++++++++++

     0  1  2  3  4*

  +---------------*

 0|  0  1  1  2  1*

 1|  2  3  3  0  1*

 2|  2  3  2  2  1*

 3|  2  2  2  3  3*

 4|  2  2  0  2  1*

Enter operation number [0 to 4]; other integer to get out: 3

Display map borders.

++++++++++++++++++

Read in a number: 2

          --    *

  0  1  1| 2| 1 *

 --       --    *

| 2| 3  3  0  1 *

       -- --    *

| 2| 3| 2  2| 1 *

    --    --    *

| 2  2  2| 3  3 *

       -- --    *

| 2  2| 0| 2| 1 *

 -- --    --    *

Enter operation number [0 to 4]; other integer to get out: -1

The End.
```

Click here to submit to CodeCrunch.

A sample test case is provided for you to test for format correctness using the `diff` command:

```
./a.out < map5.in | diff - map5.out
```

♦ ♦ ♦ ♦ ♦

THE END