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

Project 3 – Game of Life

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

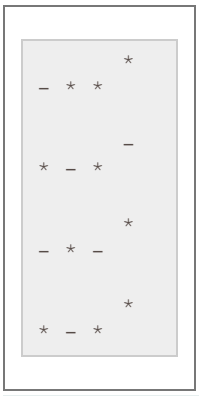
The descriptions are based on the compiler gcc and the program runs on the UNIX system. You can run your program on the PC system.

Problem Statement:

Initially, there is a square region (say of size $m \times m$) of society with m^2 cells which may be dead or alive. Our task is to monitor the evolution of the cells in this society for at several generations (as determined by the user) based on the following four **Game-of-Life Rules**:

1. Any live cell with fewer than two live neighbours dies, as if caused by loneliness.
2. Any live cell with two or three live neighbours lives on to the next generation, as if sustained by friendship.
3. Any live cell with more than three live neighbours dies, as if caused by over-crowdedness.
4. Any dead cell with exactly three live neighbours becomes a live cell in the next generation, as if by reproduction.

For example, given the following demography of the society of a 4-by-4 region (here a dead cell is represented by '-', and a live cell is represented by '*' respectively):



After one evolution, the new generation will have the following demography:

In building the society, you as the society creator can internally represent a live cell by an integer 1, and a dead cell by an integer 0. In fact, it will be very handy for you to manipulate the evolutions using integer encoding.

In receiving and reporting about the liveness of the society however, you **must use** the character '*' to represent a live cell, and the character '-' to represent a dead cell, as shown in the example above.

This task is divided into several levels. Read through all the levels (from first to last, then from last to first) to see how the different levels are related. **You may start from any level. Some useful tips are give at the end of this description.**

♦ ♦ ♦ ♦ ♦

Level 1

Name your program as `life1.c`.

Write a program that builds the square-region society of size $m \times m$, where m is the maximal permissible boundary size provided by the user. You may assume that the m is positive and cannot be bigger than 48. Create the society by entering the liveness of each cell in the society, using '-' to represent dead and '*' to represent live. Besides these two special characters, your input can also include whitespaces. Lastly, print out the demography of the society by using '-' to represent dead and '*' to represent live.

For building the society, your program should call a procedure named `read_soc` to do the job. The prototype of `read_soc` is as follows:

```
void read_soc(int soc[][MAXSIZE], int size);
```

This procedure will read in '-'s and '-'s from the user, ignoring any whitespaces entered, and build the society as a 2D array (filled with 0's and 1's.)

For displaying the demography of the society, your program should call a procedure named `prt_soc` to do the job, which has the following prototype:

```
    void prt_soc(int soc[][MAXSIZE], int size);
```

Note that the demography displayed will be surrounded by a boundary formed by characters '/', '\', 'o' and character '|', as shown in the sample runs below. Furthermore, two adjacent cell values (either '-' or '*') in a row are separated by a blank space.

The following are some sample runs of the program. User input is in **bold** font.

```
$ a.out

Read the boundary size: 6

Build the society.

*   - - - - *

- *   - - * -

- - *       * - -

- *--* *

* - -

- - *

* -   *

* - -

New Life:

/ooooooooooooo\

| * - - - - * |

| - * - - * - |

| - - * * - - |

| - * - - * * |

| * - - - - * |

| * - * * - - |

\ooooooooooooo/
```

```
$ a.out

Read the boundary size: 5

Build the society.

- * * - *

- * - * -

* - * - *

- * * * -

- * - * -

New Life:

/ooooooooooo\

| - * * - * |

| - * - * - |

| * - * - * |

| - * * * - |

| - * - * - |

\ooooooooooo/
```

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

```
cp life1.c life2.c
```

♦ ♦ ♦ ♦ ♦

Level 2

Name your program as `life2.c`.

Write a program that builds the square-region society of size $m \times m$, where m is the maximal permissible boundary size provided by the user. You may assume that the m is positive and cannot be bigger than 48. Create the society by entering the liveness of each cell in the society, using '-' to represent dead and '*' to represent live. Besides these two special

characters, your input can also include whitespaces. Then, print out the demography of the society by using '-' to represent dead and '*' to represent live.

Also provide a statistics indicating the number of live cells in the society.

For building the society, your program should call a procedure named `read_soc` to do the job. The prototype of `read_soc` is as follows:

```
void read_soc(int soc[][MAXSIZE], int size);
```

This procedure will read in '-'s and '-'s from the user, ignoring any whitespaces entered, and build the society as a 2D array (filled with 0's and 1's.)

For displaying the demography of the society, your program should call a procedure named `prt_soc` to do the job, which has the following prototype:

```
void prt_soc(int soc[][MAXSIZE], int size);
```

Note that the demography displayed will be surrounded by a boundary formed by characters '/', '\', 'o' and character '|', as shown in the sample runs below. Furthermore, two adjacent cell values (either '-' or '*') in a row are separated by a blank space.

For computing the live population of the society, your program should call a function named `population` to do the job, which has the following prototype:

```
int population(int soc[][MAXSIZE], int size);
```

The following are some sample runs of the program. User input is in **bold** font.

```
$ a.out

Read the boundary size: 6

Build the society.
*   - - - - *
- *   - - * -
- - *       * - -
- *--* *
* - -
```

```
- - *

* -   *

* - -

New Life:

/ooooooooooooo\

| * - - - - * |

| - * - - * - |

| - - * * - - |

| - * - - * * |

| * - - - - * |

| * - * * - - |

\ooooooooooooo/

The number of inhabitants is: 14.
```

```
$ a.out

Read the boundary size: 15

Build the society.

 -   -   -  * -   -   -  * * - * - * * -

* * *  -   -  * * * * * * * * * -

*  -  *  -   -   -  * * * * *  -   -  * *

* * *  -   -   -  * *  -   -  * * * * *

* *  -   -   -  *  -  *  -  *  -  * * *  -

* * *  -   -  * *  -   -   -  *  -   -   -   -

*  -   -   -   -   -  * *  -  *  -   -   -   -  *

 -  * *  -  *  -   -  * *  -   -  * * * *

 -  *  -   -  * * * * * *  -  * * * *

*  -  * *  -   -  *  -   -   -   -   -  *  -   -

* * *  -   -  * *  -   -  * * * *  -  *

* * *  -  * * * * *  -  * *  -   -  *

*  -  * * * * *  -   -  * * * * * *
```

```
* *  -  *  -  * * *  -   -   -   -   -  * *

 -  *  -  * * *  -   -  *  -   -  * * *  -

New Life:

/ooooooooooooooooooooooooooooooooo\

| - - - * - - - * * - * - * * * - |

| * * * - - * * * * * * * * * * - |

| * - * - - - * * * * * * - - * * |

| * * * - - - * * * - - * * * * * |

| * * - - - * - * - * - * * * * - |

| * * * - - * * - - - * - - - - - |

| * - - - - - - * * - * - - - - * |

| - * * - * - - * * - - * * * * * |

| - * - - * * * * * * - * * * * * |

| * - * * - - * - - - - - * - - - |

| * * * - - * * - - * * * * - * * |

| * * * - * * * * * - * * - - * * |

| * - * * * * * * - - * * * * * * |

| * * - * - * * * - - - - - * * * |

| - * - * * * * - - * - - * * * - |

\ooooooooooooooooooooooooooooooooo/

The number of inhabitants is: 130.
```

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

```
cp life2.c life3.c
```

♦ ♦ ♦ ♦ ♦

Level 3

Name your program as `life3.c`.

Write a program that builds the square-region society of size $m \times m$, where m is the maximal permissible boundary size provided by the user. You may assume that the m is positive and cannot be bigger than 48.

1. Create the society by entering the liveness of each cell in the society, using '-' to represent dead and '*' to represent live. Besides these two special characters, your input can also include whitespaces. Then, print out the demography of the society by using '-' to represent dead and '*' to represent live.
2. Provide a statistics indicating the number of live cells in the society.
3. Use the four **Game-of-Life Rules** described above to evolve the society and generate a new generation of the society.

For building the society, your program should call a procedure named `read_soc` to do the job. The prototype of `read_soc` is as follows:

```
void read_soc(int soc[][MAXSIZE], int size);
```

This procedure will read in '-'s and '-'s from the user, ignoring any whitespaces entered, and build the society as a 2D array (filled with 0's and 1's.)

For displaying the demography of the society, your program should call a procedure named `prt_soc` to do the job, which has the following prototype:

```
void prt_soc(int soc[][MAXSIZE], int size);
```

Note that the demography displayed will be surrounded by a boundary formed by characters '/', '\', 'o' and character '|', as shown in the sample runs below. Furthermore, two adjacent cell values (either '-' or '*') in a row are separated by a blank space.

For computing the live population of the society, your program should call a function named `population` to do the job, which has the following prototype:

```
int population(int soc[][MAXSIZE], int size);
```

For evolving the society, your program should call a **function** named `evolve_soc` to do the job, which has the following prototype:

```
int evolve_soc(int soc[][MAXSIZE], int size);
```

This function will determine the liveness of each cell in the region by adhering to the **Game-of-Life Rules**. It accepts the society as its first parameter, updates all the cells in the society through evolution, and returns the updated society via the first parameter. In addition, it

determines if there is any change in the liveness of any cell in the society during this evolution, and returns (as the returned value of the function) `1` when there is a change in the liveness of any cell, and `0` otherwise.

As this computation for liveness of a cell is done pretty often, it is useful to call a function to do the computation. This function, called `destiny`, will have the following prototype:

```
    int destiny(int soc[][MAXSIZE], int row, int col) ;
```

Note that, here, the second and third parameters (`row` and `col` respectively) indicate the position of the cell in the region which will have its liveness determined; its new liveness (which is an integer of either 0 or 1) is the returned value of the function.

You may wish to have an alternate prototype for the `destiny` function. One such variant is as follows:

```
    int destiny(int soc[][MAXSIZE], int row, int col, int rowSize, int
colSize) ;
```

This variant includes the sizes of the region that the function needs to check in order to determine the liveness of the cell (identified by `row` and `col`).

The following are some sample runs of the program. User input is in **bold** font.

```
$ a.out

Read the boundary size: 5

Build the society.

- * * - *

- * - * -

* - * - *

- * * * -

- * - * -

New Life:

/ooooooooooo\

| - * * - * |
```

```
| - * - * - |

| * - * - * |

| - * * * - |

| - * - * - |

\ooooooooooo/

The number of inhabitants is: 13.


After One Evolution:

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

/ooooooooooo\

| - * * * - |

| * - - - * |

| * - - - * |

| * - - - * |

| - * - * - |

\ooooooooooo/

The number of inhabitants is: 11.
```

```
$ a.out

Read the boundary size: 6

Build the society.

* - - - - *

- * - - * -

- - * * - -

- * - - * *

* - - - - *

* - * * - -

New Life:

/ooooooooooooo\

| * - - - - * |
```

```
| - * - - * - |

| - - * * - - |

| - * - - * * |

| * - - - - * |

| * - * * - - |

\ooooooooooooo/

The number of inhabitants is: 14.


After One Evolution:

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

/ooooooooooooo\

| - - - - - - |

| - * * * * - |

| - * * * - * |

| - * * * * * |

| * - * * - * |

| - * - - - - |

\ooooooooooooo/

The number of inhabitants is: 18.
```

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

```
cp life3.c life4.c
```

♦ ♦ ♦ ♦ ♦

Level 4

Name your program as `life4.c`.

Write a program that builds the square-region society of size $m \times m$, where m is the maximal permissible boundary size provided by the user. You may assume that the m is positive and cannot be bigger than 48.

1. Create the society by entering the liveness of each cell in the society, using '-' to represent dead and '*' to represent live. Besides these two special characters, your input can also include whitespaces. Then, print out the demography of the society by using '-' to represent dead and '*' to represent live.
2. Provide a statistics indicating the number of live cells in the society.
3. Use the four **Game-of-Life Rules** described above to evolve the society and generate a new generation of the society.
4. Treating the evolved society obtained from step 3 above as the first generation of soceity, repeatedly perform evolution on the society for a (positive) number of times, say *n*, as input by the user. Your program will stop evolving the society under one of the following conditions:
    a. You reach the end of the n^{th} evolution of the society and discover that the demography of the evolved society is different from the one before the last evolution (i.e., the result of the $(n-1)^{st}$ evolution.) In this situation, you display the result of the n^{th} evolved society, and print a statement saying "`Evolution continues after` *n* `evolutions.`"
    b. You have yet to reach *n* rounds of evolution of the society, but have discovered that in the i^{th} evolution (for *i* between 1 and *n*), the demography of the society does not change. In this situation, you show the result of the last evolved society, and print a statement saying "`The society stabilizes at` *j* `evolutions.`" (Note that, if the society stabilizes at j^{th} evolution, you can only notice it after you attempt to evolve the society (at $(j+1)^{st}$ evolution) and then realize that it has stabilized. So, the value of *j* can be 0.)

For building the society, your program should call a procedure named `read_soc` to do the job. The prototype of `read_soc` is as follows:

```
void read_soc(int soc[][MAXSIZE], int size);
```

This procedure will read in '-'s and '-'s from the user, ignoring any whitespaces entered, and build the society as a 2D array (filled with 0's and 1's.)

For displaying the demography of the society, your program should call a procedure named `prt_soc` to do the job, which has the following prototype:

```
void prt_soc(int soc[][MAXSIZE], int size);
```

Note that the demography displayed will be surrounded by a boundary formed by characters '/', '\', 'o' and character '|', as shown in the sample

runs below. Furthermore, two adjacent cell values (either '-' or '*') in a row are separated by a blank space.

For computing the live population of the society, your program should call a function named `population` to do the job, which has the following prototype:

```
int population(int soc[][MAXSIZE], int size);
```

For evolving the society, your program should call a **function** named `evolve_soc` to do the job, which has the following prototype:

```
int evolve_soc(int soc[][MAXSIZE], int size);
```

This function will determine the liveness of each cell in the region by adhering to the **Game-of-Life Rules**. It accepts the society as its first parameter, updates all the cells in the society through evolution, and returns the updated society via the first parameter. In addition, it determines if there is any change in the liveness of any cell in the society during this evolution, and returns (as the returned value of the function) `1` when there is a change in the liveness of any cell, and `0` otherwise.

As this computation for liveness of a cell is done pretty often, it is useful to call a function to do the computation. This function, called `destiny`, will have the following prototype:

```
int destiny(int soc[][MAXSIZE], int row, int col) ;
```

Note that, here, the second and third parameters (`row` and `col` respectively) indicate the position of the cell in the region which will have its liveness determined; its new liveness (which is an integer of either 0 or 1) is the returned value of the function.

You may wish to have an alternate prototype for the `destiny` function. One such variant is as follows:

```
int destiny(int soc[][MAXSIZE], int row, int col, int rowSize, int
colSize) ;
```

This variant includes the sizes of the region that the function needs to check in order to determine the liveness of the cell (identified by `row` and `col`).

For evolving the society for a number of generations, your program should call a procedure named `generate` to accomplish the task. Its prototype is povided here:

```
void generate(int soc[][MAXSIZE], int size, int gen) ;
```

This procedure takes in the usual arguments about the society, and receives a positive integer `gen` as its third parameter. The third parameter indicates the number of evolutions to be performed by this procedure. At the end of its execution, the procedure displays a message indicating if the society has stabilizes or the society continues to evolve.

The following are some sample runs of the program. User input is in **bold**.

```
$ a.out

Read the boundary size: 5

Build the society.

- * * - *

- * - * -

* - * - *

- * * * -

- * - * -

New Life:

/ooooooooooo\

| - * * - * |

| - * - * - |

| * - * - * |

| - * * * - |

| - * - * - |

\ooooooooooo/

The number of inhabitants is: 13.


After One Evolution:

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

/ooooooooooo\

| - * * * - |

| * - - - * |

| * - - - * |
```

```
| * - - - * |

| - * - * - |

\oooooooooooo/

The number of inhabitants is: 11.


3

Evolution begins:

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

/oooooooooooo\

| - * * * - |

| - * * * - |

| * * - * * |

| - - - - - |

| - - - - - |

\oooooooooooo/

The number of inhabitants is: 10.

Evolution continues after 3 evolutions.
```

```
$a.out

Read the boundary size: 5

Build the society.

- * - * -

* - * - *

* - * - *

- * - * -

- - * - -

New Life:

/oooooooooooo\

| - * - * - |

| * - * - * |
```

```
| * - * - * |

| - * - * - |

| - - * - - |

\ooooooooooo/

The number of inhabitants is: 11.


After One Evolution:

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

/ooooooooooo\

| - * * * - |

| * - * - * |

| * - * - * |

| - * - * - |

| - - * - - |

\ooooooooooo/

The number of inhabitants is: 12.


3

Evolution begins:

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

/ooooooooooo\

| - * * * - |

| * - - - * |

| * - * - * |

| - * - * - |

| - - * - - |

\ooooooooooo/

The number of inhabitants is: 11.

The society stabilizes at evolution 1.
```

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

```
cp life4.c life5.c
```

♦ ♦ ♦ ♦ ♦

Level 5

Name your program as `life5.c`.

Write a program that builds the square-region society of size $m \times m$, where m is the maximal permissible boundary size provided by the user. You may assume that the m is positive and cannot be bigger than 48.

1. Create the society by entering the liveness of each cell in the society, using '-' to represent dead and '*' to represent live. Besides these two special characters, your input can also include whitespaces. Then, print out the demography of the society by using '-' to represent dead and '*' to represent live.
2. Provide a statistics indicating the number of live cells in the society.
3. Use the four **Game-of-Life Rules** described above to evolve the society and generate a new generation of the society.
4. Treating the evolved society obtained from step 3 above as the first generation of soceity, repeatedly perform evolution on the society for a (positive) number of times, say n, as input by the user. Your program will stop evolving the society under one of the following conditions:
    a. You reach the end of the n^{th} evolution of the society and discover that the demography of the evolved society is different from the one before the last evolution (i.e., the result of the $(n-1)^{st}$ evolution.) In this situation, you display the result of the n^{th} evolved society, and print a statement saying "`Evolution continues after` *n* `evolutions.`"
    b. You have yet to reach n rounds of evolution of the society, but have discovered that in the i^{th} evolution (for i between 1 and n), the demography of the society does not change. In this situation, you show the result of the last evolved society, and print a statement saying "`The society stabilizes at` *j* `evolutions.`" (Note that, if the society stabilizes at j^{th} evolution, you can only notice it after you attempt to evolve the society (at $(j+1)^{st}$ evolution) and then realize that it has stabilized. So, the value of j can be 0.)
5. Perform a population census for this society. That is, for each row in the society, determine how many live cells in that row.

For building the society, your program should call a procedure named `read_soc` to do the job. The prototype of `read_soc` is as follows:

```
void read_soc(int soc[][MAXSIZE], int size);
```

This procedure will read in '-'s and '-'s from the user, ignoring any whitespaces entered, and build the society as a 2D array (filled with 0's and 1's.)

For displaying the demography of the society, your program should call a procedure named `prt_soc` to do the job, which has the following prototype:

```
void prt_soc(int soc[][MAXSIZE], int size);
```

Note that the demography displayed will be surrounded by a boundary formed by characters '/', '\', 'o' and character '|', as shown in the sample runs below. Furthermore, two adjacent cell values (either '-' or '*') in a row are separated by a blank space.

For computing the live population of the society, your program should call a function named `population` to do the job, which has the following prototype:

```
int population(int soc[][MAXSIZE], int size);
```

For evolving the society, your program should call a **function** named `evolve_soc` to do the job, which has the following prototype:

```
int evolve_soc(int soc[][MAXSIZE], int size);
```

This function will determine the liveness of each cell in the region by adhering to the **Game-of-Life Rules**. It accepts the society as its first parameter, updates all the cells in the society through evolution, and returns the updated society via the first parameter. In addition, it determines if there is any change in the liveness of any cell in the society during this evolution, and returns (as the returned value of the function) `1` when there is a change in the liveness of any cell, and `0` otherwise.

As this computation for liveness of a cell is done pretty often, it is useful to call a function to do the computation. This function, called `destiny`, will have the following prototype:

```
int destiny(int soc[][MAXSIZE], int row, int col) ;
```

Note that, here, the second and third parameters (`row` and `col` respectively) indicate the position of the cell in the region which will have its liveness

determined; its new liveness (which is an integer of either 0 or 1) is the returned value of the function.

You may wish to have an alternate prototype for the `destiny` function. One such variant is as follows:

```
    int destiny(int soc[][MAXSIZE], int row, int col, int rowSize, int
colSize) ;
```

This variant includes the sizes of the region that the function needs to check in order to determine the liveness of the cell (identified by `row` and `col`).

For evolving the society for a number of generations, your program should call a procedure named `generate` to accomplish the task. Its prototype is povided here:

```
    void generate(int soc[][MAXSIZE], int size, int gen) ;
```

This procedure takes in the usual arguments about the society, and receives a positive integer `gen` as its third parameter. The third parameter indicates the number of evolutions to be performed by this procedure. At the end of its execution, the procedure displays a message indicating if the society has stabilizes or the society continues to evolve.

To conduct a population census, you write the following procedure:

```
    void census(int soc[][M], int size, int pop[]);
```

This runs through each row in the society to sum up the number of live cells, and records these numbers in the one-dimensional array <\tt>pop.

To display the census result, you may wish to write another procedure to do so.

The following are some sample runs of the program. User input is in **bold**.

```
$ a.out

Read the boundary size: 5

Build the society.

- * - * -

* - * - *
```

```
* - * - *

- * - * -

- - * - -

New Life:

/ooooooooooo\

| - * - * - |

| * - * - * |

| * - * - * |

| - * - * - |

| - - * - - |

\ooooooooooo/

The number of inhabitants is: 11.


After One Evolution:

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

/ooooooooooo\

| - * * * - |

| * - * - * |

| * - * - * |

| - * - * - |

| - - * - - |

\ooooooooooo/

The number of inhabitants is: 12.


3

Evolution begins:

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

/ooooooooooo\

| - * * * - |

| * - - - * |
```

```
| * - * - * |

| - * - * - |

| - - * - - |

\ooooooooooo/

The number of inhabitants is: 11.

The society stabilizes at evolution 1.


Census Result:

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

[ 3 2 3 2 1 ]
```

```
$a.out

Read the boundary size: 6

Build the society.

- - - - - -

- * - * - *

* - * - * -

- * - * - *

- * - * - *

* - * - * -

New Life:

/ooooooooooooo\

| - - - - - - |

| - * - * - * |

| * - * - * - |

| - * - * - * |

| - * - * - * |

| * - * - * - |

\ooooooooooooo/

The number of inhabitants is: 15.
```

```
After One Evolution:
++++++++++++++++++++
/oooooooooooooo\
| - - - - - - |
| - * * * * - |
| * - - - - * |
| * * - * - * |
| * * - * - * |
| - * * * * - |
\oooooooooooooo/
The number of inhabitants is: 18.


5
Evolution begins:
++++++++++++++++
/oooooooooooooo\
| - - - - * - |
| * * - - * * |
| * - - * - * |
| - - - - - * |
| - - - - * - |
| - - - - - - |
\oooooooooooooo/
The number of inhabitants is: 10.
Evolution continues after 5 evolutions.


Census Result:
++++++++++++++
[ 1 4 3 1 1 0 ]
```

To proceed to the next level (say level 6), copy your program by typing the Unix command

```
cp life5.c life6.c
```

♦ ♦ ♦ ♦ ♦

Level 6

Name your program as `life6.c`.

Write a program that builds the square-region society of size $m \times m$, where m is the maximal permissible boundary size provided by the user. You may assume that the m is positive and cannot be bigger than 48.

1. Create the society by entering the liveness of each cell in the society, using '-' to represent dead and '*' to represent live. Besides these two special characters, your input can also include whitespaces. Then, print out the demography of the society by using '-' to represent dead and '*' to represent live.
2. Provide a statistics indicating the number of live cells in the society.
3. Use the four **Game-of-Life Rules** described above to evolve the society and generate a new generation of the society.
4. Treating the evolved society obtained from step 3 above as the first generation of soceity, repeatedly perform evolution on the society for a (positive) number of times, say n, as input by the user. Your program will stop evolving the society under one of the following conditions:
    a. You reach the end of the n^{th} evolution of the society and discover that the demography of the evolved society is different from the one before the last evolution (i.e., the result of the $(n-1)^{st}$ evolution.) In this situation, you display the result of the n^{th} evolved society, and print a statement saying "`Evolution continues after` *n* `evolutions.`"
    b. You have yet to reach n rounds of evolution of the society, but have discovered that in the i^{th} evolution (for i between 1 and n), the demography of the society does not change. In this situation, you show the result of the last evolved society, and print a statement saying "`The society stabilizes at` *j* `evolutions.`" (Note that, if the society stabilizes at j^{th} evolution, you can only notice it after you attempt to evolve the society (at $(j+1)^{st}$ evolution) and then realize that it has stabilized. So, the value of j can be 0.)
5. Perform a population census for this society. That is, for each row in the society, determine how many live cells in that row.

6. Perform a migration on the society based on the result of your population census. Specifically, move those rows with more number of live cells to the top rows of the society, and those with less number of live cells to the bottom. For instance, given the following demography and census result:

Demography	Census Result
- * * * * - - * * - * - * * * -	3 2 2 3

7. We sort the census results in descending order and arrange the rows according to this order. Note that two rows having the same number of live cells must preserve their relative up-down ordering before and after migration. For the case above, the result of the migration is as follows:

Demography after migration	Sorted Census Result
- * * * * * * - * - - * * - * -	3 3 2 2

For building the society, your program should call a procedure named `read_soc` to do the job. The prototype of `read_soc` is as follows:

```
void read_soc(int soc[][MAXSIZE], int size);
```

This procedure will read in '-'s and '-'s from the user, ignoring any whitespaces entered, and build the society as a 2D array (filled with 0's and 1's.)

For displaying the demography of the society, your program should call a procedure named `prt_soc` to do the job, which has the following prototype:

```
void prt_soc(int soc[][MAXSIZE], int size);
```

Note that the demography displayed will be surrounded by a boundary formed by characters '/', '\', 'o' and character '|', as shown in the sample runs below. Furthermore, two adjacent cell values (either '-' or '*') in a row are separated by a blank space.

For computing the live population of the society, your program should call a function named `population` to do the job, which has the following prototype:

```
int population(int soc[][MAXSIZE], int size);
```

For evolving the society, your program should call a **function** named `evolve_soc` to do the job, which has the following prototype:

```
int evolve_soc(int soc[][MAXSIZE], int size);
```

This function will determine the liveness of each cell in the region by adhering to the **Game-of-Life Rules**. It accepts the society as its first parameter, updates all the cells in the society through evolution, and returns the updated society via the first parameter. In addition, it determines if there is any change in the liveness of any cell in the society during this evolution, and returns (as the returned value of the function) `1` when there is a change in the liveness of any cell, and `0` otherwise.

As this computation for liveness of a cell is done pretty often, it is useful to call a function to do the computation. This function, called `destiny`, will have the following prototype:

```
int destiny(int soc[][MAXSIZE], int row, int col) ;
```

Note that, here, the second and third parameters (`row` and `col` respectively) indicate the position of the cell in the region which will have its liveness determined; its new liveness (which is an integer of either 0 or 1) is the returned value of the function.

You may wish to have an alternate prototype for the `destiny` function. One such variant is as follows:

```
int destiny(int soc[][MAXSIZE], int row, int col, int rowSize, int
colSize) ;
```

This variant includes the sizes of the region that the function needs to check in order to determine the liveness of the cell (identified by `row` and `col`).

For evolving the society for a number of generations, your program should call a procedure named `generate` to accomplish the task. Its prototype is povided here:

```
void generate(int soc[][MAXSIZE], int size, int gen) ;
```

This procedure takes in the usual arguments about the society, and receives a positive integer `gen` as its third parameter. The third parameter indicates the number of evolutions to be performed by this procedure. At the end of its execution, the procedure displays a message indicating if the society has stabilizes or the society continues to evolve.

To conduct a population census, you write the following procedure:

```
void census(int soc[][M], int size, int pop[]);
```

This runs through each row in the society to sum up the number of live cells, and records these numbers in the one-dimensional array <\tt>pop.

To display the census result, you may wish to write another procedure to do so.

To perform migration within the society, your program will call the following procedure:

`void migrate(int soc[][M], int size, int pop[]) ;` You may wish to write your own sorting function to sort the census result, and use another 2D-array of similar size (as `soc[][M]`) to help in performing the migration. Another useful tip is given below.

The following are some sample runs of the program. User input is underlined.

The following are some sample runs of the program. User input is in **bold**.

```
$ a.out

Read the boundary size: 5

Build the society.

- * * - *

- * - * -

* - * - *
```

```
- * * * -

- * - * -

New Life:

/ooooooooooo\

| - * * - * |

| - * - * - |

| * - * - * |

| - * * * - |

| - * - * - |

\ooooooooooo/

The number of inhabitants is: 13.


After One Evolution:

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

/ooooooooooo\

| - * * * - |

| * - - - * |

| * - - - * |

| * - - - * |

| - * - * - |

\ooooooooooo/

The number of inhabitants is: 11.


3

Evolution begins:

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

/ooooooooooo\

| - * * * - |

| - * * * - |

| * * - * * |
```

```
| - - - - - |

| - - - - - |

\ooooooooooo/

The number of inhabitants is: 10.

Evolution continues after 3 evolutions.


Census Result:

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

[ 3 3 4 0 0 ]


After migration:

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

/ooooooooooo\

| * * - * * |

| - * * * - |

| - * * * - |

| - - - - - |

| - - - - - |

\ooooooooooo/

The number of inhabitants is: 10.
```

```
$a.out

Read the boundary size: 5

Build the society.

* - - - *

- * - * -

- - * - -

- * - * -

* - - - *

New Life:
```

```
/ooooooooooo\
| * - - - * |
| - * - * - |
| - - * - - |
| - * - * - |
| * - - - * |
\ooooooooooo/
The number of inhabitants is: 9.


After One Evolution:
++++++++++++++++++++
/ooooooooooo\
| - - - - - |
| - * * * - |
| - * - * - |
| - * * * - |
| - - - - - |
\ooooooooooo/
The number of inhabitants is: 8.


5
Evolution begins:
+++++++++++++++++
/ooooooooooo\
| - * * * - |
| * - - - * |
| * - - - * |
| * - - - * |
| - * * * - |
\ooooooooooo/
```

```
The number of inhabitants is: 12.

Evolution continues after 5 evolutions.



Census Result:

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

[ 3 2 2 2 3 ]



After migration:

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

/oooooooooooo\

| - * * * - |

| - * * * - |

| * - - - * |

| * - - - * |

| * - - - * |

\oooooooooooo/

The number of inhabitants is: 12.
```

♦ ♦ ♦ ♦ ♦

Useful tips

1. While determinig the liveness of each cell in the society (of size $m \times m$), it is a common practice to consider three different scenarios, depending on the location of the cell under check: (1) the cell is located at one of the four **corners** of the region: (0,0), (0,*m*-1), (*m*-1,0) and (*m*-1,*m*-1); (2) the cell is located at one of the four **borders** of the region; and (3) the cell is located **inside** the region, neither at the corner nor the border. This will requires many variety of tests in your code, making the code complicated.

   A simple way to reduce such complexity however, is to consider the society to be surrounded by **virtual** rows and columns. That is, instead of representing the society by a matrix of size ($m \times m$), with the first cell appearing at location (0,0), we can consider a **bigger**

virtual society to be a matrix of size ((*m*+2) x (*m*+2)), where *the boundaries are filled with 0, forming virtual boundaries*. The cells that we are interested in are located within the following four corner: (1,1), (1,*m*), (*m*,1) and (*m*,*m*). Now, all the cells in the **real** society are considered **inside** the virtual society, and they can all be treated uniformly under the same and only scenario.

Note that this virtual, larger society exists only in your program code, it is not known to the user using your system. So, when you display the demography of the society, you still show the demography of the real society, not the virtual one.

This is a common technique used in programming, trading memory space for speed/code-complexity.

2. During the sorting of the census results, exchanges of numbers (the counts of number of live cells in each row) are inevitable. However, it will be very expensive if during each exchange of numbers, we also exchange the contents of the corresponding two rows in the society. A much cheaper technique is to create a new and smaller 2D-array that maintains the census results and the corresponding row numbers, and sort the 2D-array. For instance:

Row number	Demography	Census Result	2D-array (census result, row number)
0 1 2 3	- * * * * - - * * - * - * * * -	3 2 2 3	(3,0) (2,1) (2,2) (3,3)

3. Now we can sort the 2D-array quite efficiently according to the first value of each pair, returning the following results:

```
4. (3,0),(3,3),(2,1),(2,2)
```

5. This means that:
   - Row 0 in the migrated society will come from row 0 of the original society;
   - Row 1 in the migrated society will come from row 3 of the original society;
   - Row 2 will come from the original row 1, and

- Row 3 will come from the original row 2.

With the above information, you can now perform the eventual migration.

♦ ♦ ♦ ♦ ♦

THE END