TÉCNICO LISBOA

Fundamentals of Programming

School year 2014-15

Second Project[1]

2048

2048 is a mathematical brain teaser in which the player slides numbered panels in a 4 × 4. The game was created in March 2014 by Gabriele Cirulli, an italian *web* programmer, and is available for free on the *web*[2] and as an *app* for various mobile platforms.

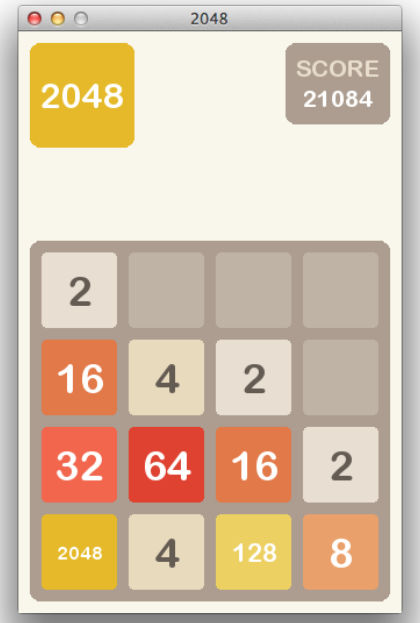


Figure 1: Example of a 2048 board with the 2048 block.

The goal of the game is to combine panels with the same number in order to create a panel with the number 2048 (see Fig. 1). In order to do that, the player can "drag" in each of four directions—up (*N*), down (*S*), left (*W*) and right (*E*). When an action is applied to the board, this affects *all* the panels with room to move. For example, the initial configuration of Fig. 2 could originate one of the four configurations shown in Fig. 3, according to the action applied.

[1]Estimated time to resolution per person, post study: 30 hours.
[2]Available in: `http://gabrielecirulli.github.io/2048/`, last accessed in 29/10/2014.

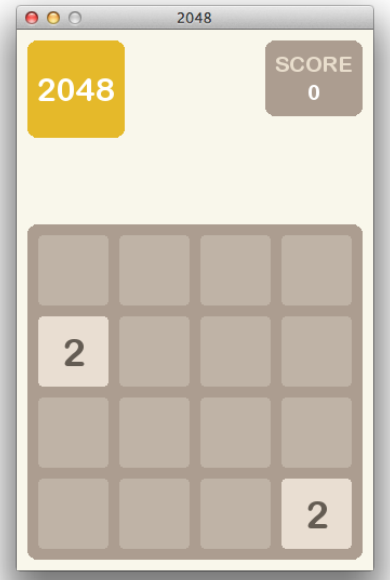
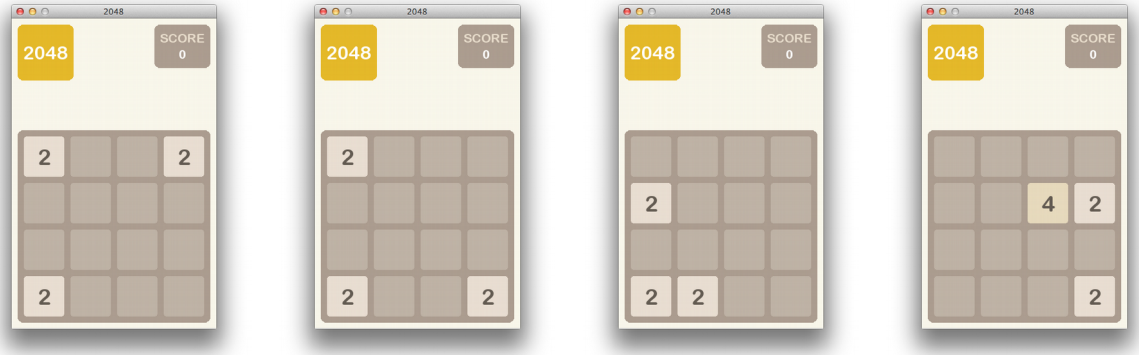


Figure 2: Example of an initial 2048 board.



(a) Action: *N*. (b) Action: *S*. (c) Action: *W*. (d) Action: *E*.

Figure 3: Fig. 2's board after each of the four possible actions.

After each movement, a new panel appears in an empty position of the board, chosen randomly. This new panel has, with a probability of 0.8, the number 2, or, with a probability of 0.2, the number 4.

1 Assignment

With this project, it is intended that a program is written using the Python programming language that defines a set of data types that should be used to manipulate the necessary information during the game, as well as a set of additional

helper functions that will allow the user to play the game itself.

1.1 Abstract Data Types (ADT)

***Coordinate* ADT (4 pts.)**

The coordinate ADT will be used to index the board's multiple positions. Each position of the board is indexed through the respective row (an integer between 1 and 4) and the respective column (an integer between 1 and 4), where position (1, 1) corresponds to the top left corner of the board. The *coordinate* ADT should then be an **immutable** that stores two integers corresponding to a row and a column of the board.

The basic operations associated with this ADT are:

1. *create_coordinate* : *integer* × *integer* → *coordinate*

   This function is the *coordinate* type's constructor. It takes two arguments of the type integer, the first of which corresponds to a row *r* (an integer between 1 and 4) and the second to a column *c* (an integer between 1 and 4), and should return and element of the type *coordinate* corresponding to the position *(r, c)*. The function should check the validity of the arguments, raising a `ValueError` with the message "`create_coordinate: invalid arguments`" in case any of the given arguments is invalid.

2. *coordinate_row* : *coordinate* → *integer*

   This accessor takes an argument of the type *coordinate* and returns the respective row.

3. *coordinate_col* : *coordinate* → *integer*

   This accessor takes an argument of the type *coordinate* and returns the respective column.

4. *is_coordinate* : *universal* → *bool*

   This recognizer takes a single argument and returns `True`,if that argument is of the type *coordinate*, and `False`, otherwise.

5. *coordinates_equal* : *coordinate* × *coordinate* → *bool*

   This test takes as arguments two elements of the type coordinate and returns True in case those arguments correspond to the same position (l, c) on the of the board, and False, otherwise.

Interaction example:

```
>>> C1 = create_coordinate(-1, 1)
[...]
builtins.ValueError: create_coordinate: invalid arguments
>>> C1 = create_coordinate(1, 5)
[...]
builtins.ValueError: create_coordinate: invalid arguments
>>> C1 = create_coordinate(0, 0)
[...]
builtins.ValueError: create_coordinate: invalid arguments
>>> C1 = create_coordinate(1, 2)
>>> coordinate_row(C1)
1
>>> coordinate_col(C1)
2
>>> is_coordinate(0)
False
>>> is_coordinate(C1)
True
>>> C2 = create_coordinate(2, 1)
>>> coordinates_equal(C1, C2)
False
```

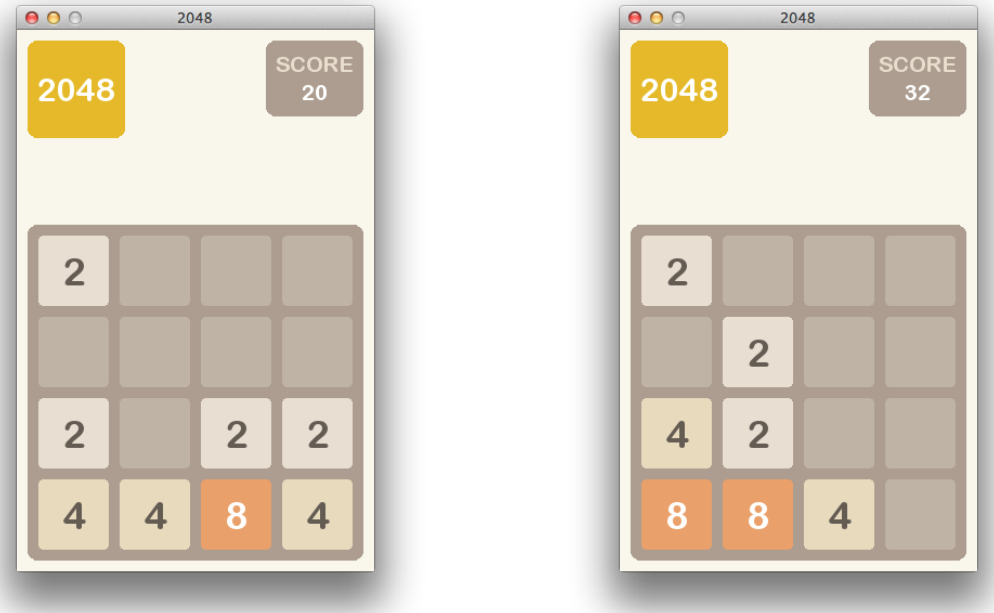
Board ADT (6 val.)

The *board* ADT will be used to represent the board. This ADT should allow to (i) represent a 2048 board (a 4 × 4 position grid) and respective score; (ii) access each of the board's positions; (iii) access the board's score; (iv) modify the content of each of the positions; (v) update the board's score; e (vi) perform the "reduction", i.e., move the pieces in the grid in a given direction and combine consecutive pieces with the same number.

Fig. 4 shows an example of a "reduction" resulting from the *W* action. After a move, each panel performs the largest displacement possible in the corresponding direction until it "collides" with another panel or one of the board's sides. The order by which the panes are displaced is inverse to the play's direction. For example, for the *W* direction, the panels are displaced in each row from right to left, *starting with the leftmost panel and ending in the rightmost panel*.

When a panel collides with another with the same number, the two are combined in a single panel and the resulting value is the sum of the original panels' values. For example, in Fig. 4's board, the panel with the value 4 in position (4, 2) collides with the panel in position (4, 1), originating a panel with the value 8 in position (4, 1).

A panel that's a result of a combination can not be combined with another panel in the same move. For example, in Fig. 4's board, the panel with the value 8 in position (4, 3) collides with the panel with value 8 in position (4, 1). However, since the latter was a result of a combination in this move, the combination isn't possible.



(a) Initial board. (b) After action *W*.

Figure 4: "Reduction" of the board, a result of the W action. The panels with value 4 in the 3rd row are combined into a panel with the value 8.

The value of each panel that's a result of a combination is added to the current score.

The basic operations associated with this ADT are:

6. *create_board*: {} → *board*

   This function corresponds to the constructor of the type *board*. It doesn't take any arguments, and should return an element of the type *board* according to the chosen internal representation. The board should be returned empty (i.e., every position should contain the value 0 and the score should be 0).

7. *board_position* : *board* × *coordinate* → *integer*

   This accessor takes as arguments an element *b* of the type *board* and an element *c* of the type *coordinate* and returns an element of the type integer, which corresponds to the value in the position in the board at coordinate *c*. In case the position at coordinate *c* is empty, the value 0 should be returned. The function should verify if the second argument is a valid coordinate, and raise a `ValueError` with the message "`board_position: invalid arguments`" in case it isn't.

8. *board_score* : *board* → *integer*

   This accessor takes as an argument an element *b* of the type *board* and returns the current score for board *b*.

9. *board_empty_positions* : *board* → *list*

This accessor takes as an argument an element *b* of the type *board* and returns a list containing the coordinates of every position of board *b* that is empty.

10. *board_fill_position* : *board* × *coordinate* × *integer* → *board*

    This modifier takes as arguments an element *b* of the type *board*, an element *c* of the type *coordinate* and an integer *v*, e *modifies b*, placing the value *v* in the position corresponding to the coordinate *c*. The function should return the modified board. In case any of the given arguments is invalid, it should raise a `ValueError` with the message "`board_fill_position: invalid arguments`".

11. *board_update_score* : *board* × *integer* → *board*

    This modifier takes as arguments an element *b* of the type *board* and an integer *v*, non-negative and multiple of 4. It *modifies b*, adding to the value of the respective score *v* points. The function should return the modified board. It should still verify if *v* is a non-negative multiple of 4 integer. In case any of the given arguments is invalid, it should raise a `ValueError` with the message "`board_update_score: invalid arguments`".

12. *board_reduce* : *board* × *string* → *board*

    This modifier should take as arguments an element *b* of the type *board* and one of the 4 possible actions. Particularly, *d* should be one of `'N'`, `'S'`, `'W'`, `'E'`. The function should modify *b*, reducing it in the direction *d* according to the rules of 2048. It should return the modified board *b*, including an update to the score. It should also verify if *d* is a valid move (i.e., one of the four previously indicated strings) and raise a `ValueError` with the message "`board_reduce: invalid arguments`" in case it isn't.

13. *is_board* : *universal* → *bool*

    This recognizer takes a single argument, returning `True` if its argument isn't of the type *board*, and `False` otherwise. There's no need to verify if every element in the board is a power of two.

14. *board_finished* : *board* → *bool*

    This recognizer takes as an argument an element *b* of the type *board* and returns `True` in case the board *b* is finished, i.e., in case it is full and there are no possible moves left, and `False` otherwise.

15. *boards_equal* : *board* × *board* → *bool*

    This test takes as arguments two element b_1 and b_2 of the type *board* and returns `True` in case b_1 and b_2 are two boards with the same configuration and score,

and `False` otherwise.

The following output transformer must also be implemented:

16. *print_board* : *board* → {}

    The *print_board* function takes as an argument and element *b* of the type *board* and prints to the screen an external representation of a 2048 board, as shown in the example below. It should still verify if *b* is a valid board and, in case any of the given arguments is invalid, it should raise a `ValueError` with the message "`print_board: invalid arguments`".

Interaction example:

```
>>> b = create_board(0)
>>> print_board(b)
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
Score: 0
>>> c = create_coordinate(2, 2)
>>> print_board(board_fill_position(b, c, 2))
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 2 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
Score: 0
>>> print_board(b)
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 2 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
Score: 0
>>> board_position(b, 0)
Traceback (most recent call last):
[...]
builtins.ValueError: board_position: invalid arguments
>>> board_position(b, create_coordinate(2, 4))
0
>>> board_position(b, create_coordinate(2, 2))
2
>>> c = create_coordinate(2, 4)
>>> print_board(board_fill_position(b, c, 2))
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 2 ] [ 0 ] [ 2 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
Score: 0
>>> create_coordinate(1, 1) in board_empty_positions(b)
True
>>> create_coordinate(2, 2) in board_empty_positions(b)
False
```

```
>>> is_board(b)
True
>>> is_board('a')
False
>>> print_board(board_reduce(b, 'E'))
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 4 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
Score: 4
>>> print_board(b)
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 4 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
[ 0 ] [ 0 ] [ 0 ] [ 0 ]
Score: 4
```

1.2 Additional functions

1. *request_move* : {} → *string* **(1 val.)**

   This function doesn't take any arguments, consisting only in querying the player for a direction (*N*, *S*, *E* or *W*). In case the player's input is invalid, the function should query the player again for a new move. The function returns a string corresponding to the direction picked by the player.

   Example:

   ```
   >>> j = request_move()
   Pick a move (N, S, E, W): 1
   Invalid move.
   Pick a move (N, S, E, W): a
   Invalid move.
   Pick a move (N, S, E, W): n
   Invalid move.
   Pick a move (N, S, E, W): N
   >>> j
   'N'
   ```

2. *game_2048* : {} → {} **(3 val.)**

   This function is the game's main function. It doesn't take a single argument and allows the player to play a full game of 2048.

   Before every move, the function should print the board to the screen and ask the player to pick a new move. In case the move is valid, the board should be updated, repeating this process until the game ends. Otherwise, it should print

the "`Invalid move.`" indication to the screen and query the user for a new move.

Suggestion: It might still be useful to implement two additional functions, namely `copy_board` and `fill_random_position`. The former takes as an argument an element of the type *board* and returns a copy. The latter takes as an argument an element of the type *board* and fills in a randomly chosen empty position with the numbers 2 or 4, according to their respective probability previously indicated. These aren't obligatory.

1.3 Suggestions

1. Read the entire statement, searching to understand the goal of the various functions that are requested to be implemented. When in doubt, contact the faculty by attending a Q&A schedule to clarify your situation.
2. In the project's development process, begin by implementing the ADTs in the order they're presented in the statement, following the respective methodology. Particularly, start by picking an internal representation before starting to implement the basic operations. Only then should the development on the game's functions start, also following the order in which they were presented. When developing each of the required functions, start by understanding if you can use any of the previously defined ones.
3. In order to verify your functions' functionality, use the examples as test cases.
4. Take good care when reproducing error messages and the remaining *outputs* rigorously, just like they're shown in the various examples.

2 Aspects to Avoid

The following aspects are suggestions on bad working habits that should be avoided (and, consequently, poor project grades):

1. Don't think the project is doable in the last few days before the deadline. If you start working in that period of time, Murphy's Law will come into action (every problem is more difficult than they seem; everything lasts more time than we think; and if something can go wrong, it will go wrong, in the worst of every possible occasion).
2. Don't think that a single member of the group's elements will do all of the work. This is a group project and should be done in strict collaboration (communication and control) among the group's members, each of which with their own responsibilities. Either a possible oral discussion or questions in exams about the project are used to outwit these situations.

3. Discussing ideas with other colleagues from other groups about implementation is allowed, but do not copy bits or chunks of code, even if they're slightly modified. You should understand your project as a whole, which becomes difficult when parts of the code are copied from (or inspired by) code written by someone else.
4. *Do not duplicate code*. If two functions are similar, it is only natural that these can be merged into a single one, eventually with more parameters.
5. Don't forget that excessively big functions are penalized regarding programming style.
6. The “I'm just gonna make it work for now and care about style later” attitude is completely erroneous.
7. When the program produces an error, worry about finding out the cause of the error. “Hardcoding” tends to distort your code even more.

3 Grading

The execution’s evaluation will be performed by *Mooshak*, an automatic project delivery system. There are plenty of tests configured in the system. The execution time of each test is limited, as well as the memory used in each one. Submissions are limited, only being able to submit at least 15 minutes after the previous one. Only a maximum of 10 submissions are allowed simultaneously, which means a submission could be refused if this limit is exceeded. In that case, try submitting at a later time.

The tests considered for evaluation purposes may or may not include the examples available in this statement, paired with a set of additional tests. The fact that a project completes the given examples successfully doesn’t imply, of course, that it is completely correct, since the examples that were given aren’t too exhaustive. It’s every group’s responsibility to guarantee that the code that was produced is correct.

No kind of information about the test cases used by the automatic evaluation system will be made available. The test files used in the project’s evaluation will be available in the course’s subject’s page after the delivery date.

The project's grade will be based on the following aspects::

1. Correct execution (70%).

   This part of the evaluation is done resorting to the *Mooshak* system that suggests a grade according to the various aspects that were considered.

2. Readability, namely procedural abstraction, data abstraction, well-chosen names, comment quality (and not quantity) and functions' size (25%). Your comments should include, among other things, a description of the internal representation adopted in each of the ADTs.

3. Programming style (5%).

4 Fulfillment Conditions and Deadlines

The 2nd project's delivery will be made exclusively by electronic means. The project should be submitted through the *Mooshak* system, by **11:59 PM** of **December 10th, 2014**. Tardy submissions will not be accepted, whatever the excuse may be.[3]

A single file with the `.py` extension should be submitted with all of the project's code. The file should contain a comment, on the first line, indicating the group's students' names and numbers, as well as the group number.

The source file shouldn't contain accented characters or any non-ASCII character. This includes comments and strings. Non-compliant programs will result in a score penalization of 3 points.

Two weeks before the deadline, instructions on how to submit code on Mooshak *will be published in the course's subject's page. Only then will it be possible to submit by electronic means. By then, the credentials needed to access the system will be supplied to every group. Up until the due date, multiple submissions can be made at will, being the last submission used for evaluation purposes. Therefore, a careful verification of the last submission should be done, guaranteeing it's the version of the project that is pretended to be evaluated. No exceptions will be made.*

There's the possibility of an oral discussion of the project and/or a demonstration of the project's functionality (will be decided individually).

Copies, or very identical projects, will be penalized with failure to the subject. The faculty will be the only judge of what is considered or not to be a copy.

[3] Note that the 10 simultaneous submission limit on the *Mooshak* system implies that, in the case of a high number of submission attempts on the due date, some groups might not be able to submit and will, therefore, be incapable of submitting the correct version of the code in due time.