CS 21 Machine Problem 1: Tetrisito

Department of Computer Science
College of Engineering
University of the Philippines - Diliman

1 Problem Statement

1.1 Tetris

Tetris is a puzzle video game created by Soviet software engineer Alexey Pajitnov in 1984. It has been published by several companies for multiple platforms, most prominently during a dispute over the appropriation of the rights in the late 1980s. After a significant period of publication by Nintendo, the rights reverted to Pajitnov in 1996, who co-founded the Tetris Company with Henk Rogers to manage licensing.

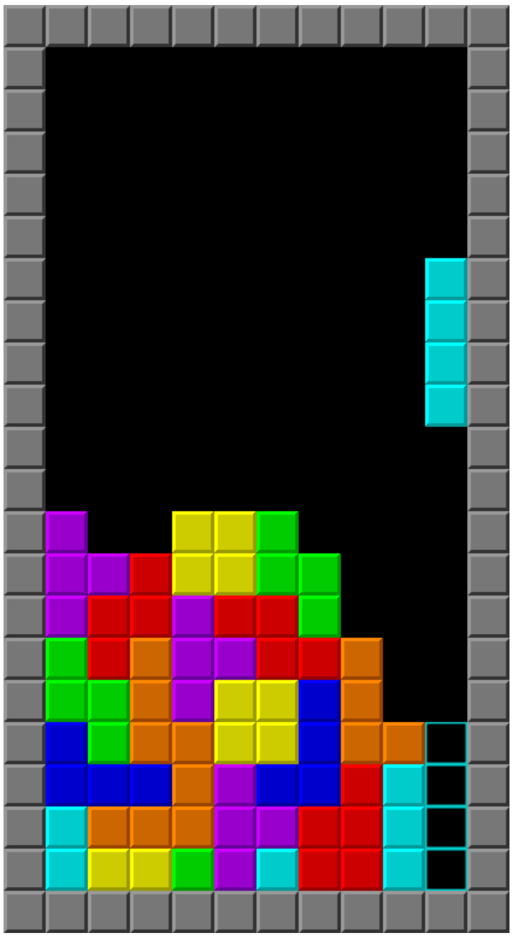
Figure 1: Classical Tetris screen

In Tetris, players complete lines by moving differently shaped pieces (tetrominoes), which descend onto the playing field. During this descent, the player can move the pieces laterally and rotate them until they touch the bottom of the field or land on a piece that had been placed before it. The completed lines disappear and grant the player points, and the player can proceed to fill the vacated spaces.

The game ends when the uncleared lines reach the top of the playing field. The longer the player can delay this outcome, the higher their score will be. In multiplayer games, players must last longer than their opponents; in certain versions, players can inflict penalties on opponents by completing a significant number of lines. Some versions add variations on the rules, such as three-dimensional displays or a system for reserving pieces[1].

1.2 Tetrisito

1.2.1 Overview

We will be exploring a custom puzzle game variant of Tetris that we made, called **Tetrisito**. Tetrisito will use a smaller 6x6 square grid of cells as its space. The grid will have an initial configuration; it could be completely empty or have some occupied cells. Then a target grid will be given, which will contain a goal configuration or shape. Lastly, one to five pieces will be provided, which can be "dropped" into the working grid. For each piece, we can choose their initial horizontal position before dropping, but while the pieces are falling, lateral movement and rotation will not be allowed (in contrast to original Tetris). The goal of the puzzle is to figure out if we can achieve the goal grid/shape by using the pieces provided.

1.2.2 Working grid

The working 6x6 grid will be described using ASCII characters. A dot (`.`) will be used to indicate an empty cell, and a pound sign/hash (`#`) will be used to indicated an occupied cell.

Here are examples for an empty grid and a grid with some occupied cells, respectively.

```
......
......
......
......
......
......
```

```
......
......
......
......
...#..
.###..
```

1.2.3 Pieces

One to five (not necessarily unique) pieces will be given for the puzzle. The pieces will be given as 4x4 grids of ASCII characters, also described using dots (`.`) and hashes (`#`). For Tetrisito, we will be using only the square, the reverse L-piece, the I-piece, and the T-piece, restricted to the following configurations:

[1]https://en.wikipedia.org/wiki/Tetris

-
  ```
  ....
  ....
  ##..
  ##..
  ```

-
  ```
  ....
  .#..
  .#..
  ##..
  ```

-
  ```
  ....
  ....
  #...
  ###.
  ```

-
  ```
  ....
  ##..
  #...
  #...
  ```

-
  ```
  ....
  ....
  ###.
  ..#.
  ```

-
  ```
  #...
  #...
  #...
  #...
  ```

-
  ```
  ....
  ....
  ....
  ####
  ```

-
  ```
  ....
  ....
  .#..
  ###.
  ```

-
  ```
  ....
  #...
  ##..
  #...
  ```

-
  ```
  ....
  ```

```
....
###.
.#..
```

-
  ```
  ....
  .#..
  ##..
  .#..
  ```

Pieces will drop from above the 6x6 grid area. Pieces can also be dropped from any initial horizontal position, as long as the piece is still completely within the grid. To illustrate:

```
....     ....     ....     ...      ..
#...     #...     #...     #..      #.
##..     ##..     ##..     ##.      ##
#...     #...     #...     #..      #.
......  ......  ......  ......  ......
......  ......  ......  ......  ......
......  ......  ......  ......  ......
......  ......  ......  ......  ......
......  ......  ......  ......  ......
......  ......  ......  ......  ......


......  ......  ......  ......  ......
......  ......  ......  ......  ......
......  ......  ......  ......  ......
#.....  .#....  ..#...  ...#..  ....#.
##....  .##...  ..##..  ...##.  ....##
#.....  .#....  ..#...  ...#..  ....#.
```

1.2.4 Initial grid configuration, goal configuration, and available pieces

A 6x6 initial configuration will be given, as well as the goal configuration. Here is an example.

```
Initial configuration

......
......
......
......
......
......

Goal configuration

......
......
......
.#.#..
```

```
.###..
.###..
```

Suppose we have the following pieces, in order from left to right:

```
....  ....
.#..  #...
.#..  ##..
##..  #...
```

We can see that we are capable of achieving the goal configuration, through the following steps:

```
......    ......     ......
......    ......     ......
...... -> ......  -> ......
......    ...#..     .#.#..
......    ...#..     .###..
......    ..##..     .###..
```

1.2.5 Additional differences from Tetris

Line clearing will not be required (see bonus section). Lateral movement and rotation for falling pieces is also not allowed.

If dropping a piece will cause it to stick out of the 6x6 grid (exceed the limits of the grid), that move will NOT be allowed.

If a piece can no longer be added to the grid, that piece can be skipped and the next piece can be considered.

1.2.6 Task

Your task for this project is to write a MIPS assembly program that takes as input, the initial grid configuration, the goal configuration, the number of pieces, then the pieces themselves, and determine whether the goal configuration can be reached from the initial configuration, using the pieces provided.

The intended solution for this problem is **exhaustive search/recursive backtracking**.

2 Input

Input will be given via standard input. The input will be given in the following order: (1) initial grid, (2) goal grid, (3) number of pieces, (4) pieces.

The grids and pieces will be given as ASCII characters, and the number of pieces will be given as an integer.

Here is the sample input for the scenario described in the previous section.

```
......
......
......
......
......
......
......
```

```
......
......
.#.#..
.###..
.###..
2
....
.#..
.#..
##..
....
#...
##..
#...
```

For checking purposes, input files will be used. For example, the input file to be used is named `input.in`. With your `.asm` file in the same directory (along with additional files such as `macros.asm`), say `cs21project1.asm`, the command

```
java -jar Mars4_5.jar p cs21project1.asm < input.in
```

will be used to check your code.

3 Output

The output of your program will be a single string, which is either `YES` or `NO`, indicating whether it is possible or not to reach the goal configuration from the initial configuration, using the pieces provided.

Output will be via the MARS console (standard output).

4 Scoring

Scoring will be based on partial or full implementations. The perfect score for this machine problem is **100 points**. Some points will come from the documentation (check the relevant section towards the end of the document).

You may submit more than one implementation, however only your **highest scoring submission** will be credited.

4.1 A: Only one piece to fall

For this partial implementation, **only one piece** will be given.

4.1.1 Sample input 1

```
......
......
......
......
.###..
..#...
......
......
..#...
```

```
.###..
.###..
..#...
1
....
....
.#..
###.
```

4.1.2 Sample output 1

```
YES
```

4.1.3 Sample input 2

```
......
......
......
......
.###..
..#...
......
......
..#...
.###..
.###..
..#...
1
....
....
....
####
```

4.1.4 Sample output 2

```
NO
```

This implementation will earn you **40 points**.

4.2 B: Several pieces fall, in fixed order

For this partial implementation, one to five pieces can be given. The order in which they are given will be the order they are dropped into the grid.

4.2.1 Sample input 1

```
......
......
......
......
......
......
......
......
```

```
......
.#.#..
.###..
.###..
2
....
.#..
.#..
##..
....
#...
##..
#...
```

4.2.2 Sample output 1

```
YES
```

4.2.3 Sample input 2

```
......
......
......
......
......
......
......
..#...
..##..
...#..
..##..
..##..
2
....
.#..
.#..
##..
....
#...
##..
#...
```

4.2.4 Sample output 2

```
NO
```

This implementation will earn you **75 points**.

4.3 C: Several pieces fall, in any order

This is the full implementation. One to six pieces can be given, and the order in which the pieces are dropped can be altered.

4.3.1 Sample input 1

```
......
......
......
......
......
......
......
......
......
.#.#..
.###..
.###..
2
....
#...
##..
#...
....
.#..
.#..
##..
```

4.3.2 Sample output 1

```
YES
```

4.3.3 Sample input 2

```
......
......
......
......
......
......
......
..#...
..##..
...#..
..##..
..##..
2
....
.#..
.#..
##..
....
#...
##..
#...
```

4.3.4 Sample output 2

```
NO
```

This implementation will earn you **90 points**.

4.4 BONUS 1: Lateral movement is allowed

Lateral movement will be allowed while the piece/s is/are falling.

You may include this feature in any of the partial or full implementations described above. If your implementation passes the test cases for lateral movement, you will get **20 additional points**.

4.5 BONUS 2: Lines are completed and cleared

Once a piece settles into place in the grid, if a row (or line) is fully occupied, that line gets cleared, and any blocks previously on top of the cleared line will descend until they hit other blocks or the bottom of the grid.

You may include this feature in any of the partial or full implementations described above. If your implementation passes the test cases for line clearing, you will get **20 additional points**.

5 Implementation

Your assembly programs must follow MIPS conventions and our own CS21 conventions. These include:

1. header comments including your name and section
2. 4-column format for the source code
3. proper initialization and cleanup of functions (`lw` and `sw`, `$sp` decrement/increment, storing `$ra`)
4. proper usage of registers for passing arguments and return values
5. proper usage of preserved and unpreserved registers for function calls
6. using `$gp` for global variables

You may opt to implement auxiliary functions. Indicate the presence of these auxiliary functions in the documentation. You may also use macros, and any concept you have learned from the MIPS-related lab exercises so far.

6 Documentation

The documentation is worth **10 points**, and is necessary to earn any implementation points (no documentation, no implementation points). Note that no points will be given for documentation that does not come with a functional program (documentation-only submissions are banned).

The documentation shall contain a description of the implementation chosen, and a short narrative describing the implementation approach. The documentation should also contain relevant code snippets and short narratives explaining the purpose of each code snippet in the execution of the program. This includes input/output code, function definitions, usage of registers, and parts of the algorithm proper.

7 Submission and Testing

Submission of the documentation and `.asm` source code/s is via UVLE. Name your assembly source code as **cs21project1`letter`.asm**, where `letter` denotes the implementation that you chose (`A`, `B`, or `C`). Refer to the UVLe submission bin for the deadline. Please submit your implementation in the corresponding submission bin. If you have additional files, compress all of your files together into a `.zip` archive.

Sample test cases for the different implementations will be provided on UVLe along with this specification. Upon submission of the MP, your programs will be tested on the sample test cases and hidden test cases. The test cases will follow the limits given in the specifications. Your programs will be checked in an **all-or-nothing** manner; please test your program/s diligently.

All forms of academic dishonesty will not be tolerated. We will not hesitate to file necessary academic dishonesty charges against students that would be involved in the copying of another person's code.