CS 351 – Advanced Data Structures Practicum

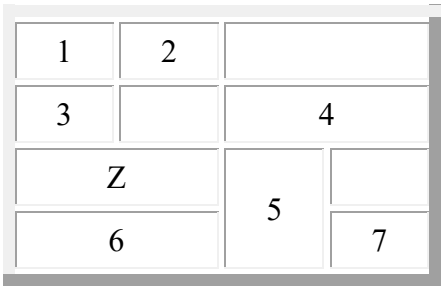
Sliding Block Puzzles

Due: Wednesday, March 29, 2023 at 11:59 pm

A sliding block puzzle consists of a number of pieces that fit into a confined area. The goal is to move one of the pieces to a specific position. This piece will be called the "goal piece". The goal can only be achieved by moving all of the pieces in a certain specified order of moves. Each piece may be restricted in the direction that it can move. Such games or puzzles can be found under the names of Traffic Jam, Rush Hour, Parking Lot, Blocked, Unblock Me, etc. Check out:

- https://www.mathplayground.com/slidingblock.html
- https://en.wikipedia.org/wiki/Klotski
- https://en.wikipedia.org/wiki/Rush_Hour_(puzzle)

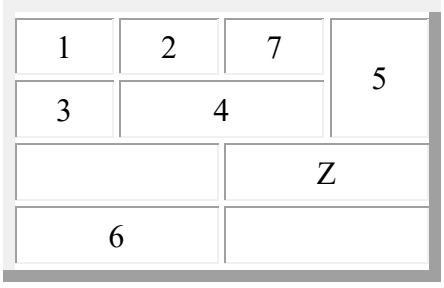
Consider the following puzzle:



The puzzle contains 8 pieces. The piece labeled "Z" is the "goal piece". This piece must be moved to the right hand edge of the puzzle. In this puzzle all pieces can move any direction we wish (left/right or up/down). One solution is to:

1. move piece 4 left one space,
2. then move piece 7 up 3 spaces and left 1 space,
3. then move piece 5 right one space then up 2 spaces,
4. then finally move piece Z right 2 spaces.

The result looks like this:



Your program is to find the shortest solution for these types of puzzles. The solution will always have the goal piece move to the right hand side of the puzzle grid (i.e. the goal piece moves into the last column). Your program is take a command line argument that will contain the name of a data file containing the needed

information about the puzzle. The output of the puzzle will be a sequence of moves that give the shortest solutions.

The solution is to be given as an ordered list of moves as shown below. Each move is to show the Piece moved, the number of space(s) moved, and the direction (up, down, left or right) the piece moved. The solution for the above puzzle is as follows (note there is one space character between output values):

```
1. Piece 4 moves 1 space(s) left
2. Piece 7 moves 3 space(s) up
3. Piece 5 moves 1 space(s) right
4. Piece 7 moves 1 space(s) right
5. Piece 5 moves 2 space(s) up
6. Piece Z moves 2 space(s) right
```

The shortest solution will have the fewest number of moves. Note: that there could be multiple shortest solutions for a puzzle. The above solution could have reversed moves 3 & 4 and still have been the shortest solution. To make verifying output easier, we want to find a specific solution. Thus to make things uniform, you are to attempt to move pieces in order in which they were given in the input (see below for more information) and we are to attempt to move pieces in the following order for the four directions: up, down, left, right.

Your program is to find the fewest number of moves for the solution. Note that it is to treat the following as 1 move:

```
2. Piece 7 moves 3 space(s) up
```

while the following is considered 3 moves:

```
2. Piece 7 moves 1 space(s) up
3. Piece 7 moves 1 space(s) up
4. Piece 7 moves 1 space(s) up
```

Note that the following two moves **COULD NOT** be combined into a single move since the direction that the piece is moving changes. This piece movement is from the very first description of the solution of this puzzle.

```
2. Piece 7 moves 3 space(s) up
3. Piece 7 moves 1 space(s) right
```

This is the same for the following two moves. They also COULD NOT be combined:

```
4. Piece 5 moves 1 space(s) right
5. Piece 5 moves 2 space(s) up
```

Input Format

The input for a puzzle will always come from a file.

- The first line of the file will contain 2 integers, the number of rows and the number of columns of the puzzle "grid". If either of these values is zero or less, print an error message and end the program. These values will be separated by one or more white space characters. The puzzle will always use a rectangular grid. Error messages are to be:
  - **`Error: Number of rows must be greater than zero`**
  - **`Error: Number of columns must be greater than zero`**
- The second line of the file will contain the starting position of the goal piece.
- The remaining lines of the file will contain the starting position of the other pieces in the puzzle.
- Once the end of the file is read, then all of the pieces have been read in.

Each piece will always have a rectangular shape (while such puzzles with non-rectangular shape do exist, it adds a complexity we don't need to deal with here). Each piece's starting position is given by 4 integer values and one character value. These values will be separated by one or more white space characters.

- The first integer will be the starting row position.
- The second integer will be the starting column position.
- The third integer will be the width in columns.
- The fourth integer will be the height in rows.
- The character value will specify the direction of movement the piece can have. This character can be either an "h" for horizontal movement (left or right), a "v" for vertical movement (up or down), a "b" for both horizontal and vertical movement, or a "n" for no movement (the piece cannot move, it must stay in that space).

If a piece would fall outside of the puzzle grid, have an invalid direction of movement, or overlap with another piece, an appropriate error message should be printed and the piece should be discarded from the puzzle (i.e. don't quit the program). Error message should be as follows, where R,C is replaced with the starting row position and starting column position given in the input file:

- `Warning: Piece with starting position of R,C falls outside of grid`
- `Warning: Piece with starting position of R,C has invalid movement`
- `Warning: Piece with starting position of R,C overlaps with other piece`

If the goal piece is listed incorrectly, the first correctly listed piece becomes the goal piece. The upper left corner of the grid has row = 1 and column = 1. The input for the above puzzle would be as follows:

```
4  4
3  1  2  1  b
1  1  1  1  b
1  2  1  1  b
2  1  1  1  b
2  3  2  1  b
3  3  1  2  b
4  1  2  1  b
4  4  1  1  b
```

Note that the names of the pieces are not specified in the input file. The goal piece will always have the name of "Z". The next nine pieces will have the names from "1" to "9". The next 26 pieces will be given names using the lower case letters from "a" to "z". The next 25 pieces will be given names using the upper case letters from "A" to "Y" (since "Z" is already in use). This will give us 61 pieces. We will use this as the limit of possible

pieces. If the file has more than 61 pieces, come up with some appropriate error message and only use the first 61 valid pieces for the rest of the execution of the program. (We do not plan to test this, but you should.)

Sample Input

Some sample input data files are:

- proj3a.data (The one listed above)
- proj3b.data
- proj3c.data This one has no solution.
- proj3d.data
- proj3e.data
- proj3f.data
- proj3g.data Error in input
- proj3h.data Error in input
- proj3i.data Error in input
- proj3j.data Error in input
- proj3k.data A 3 piece puzzle
- proj3l.data Error in input
- proj3m.data
- proj3n.data

Program Output

Your output from the program should be written to standard output and should consist of four parts:

1. Any error messages generated by invalid input.

2. A listing of the grid as the start of the puzzle. This will be a simple ASCII graphic as shown below:

```
******
*12..*
*3.44*
*ZZ5.*
*6657*
******
```

   Note the use of asterisks around the outside of the output and the use of periods instead of spaces to indicate empty locations. It is hoped that these make the output a bit more readable. This should work well for most puzzles that have a size that are easily displayed. Of course, this grid is less readable if the grid so large that it causes line wrap (but this is not the programmer's problem).

3. If the puzzle is solvable, Print:
   a. the number of steps in your solution (were N is the replaced the number of step needed),
      - `This puzzle is solvable in N steps`
   b. then list out the moves that solve the puzzle. Refer to information at top of write-up

   If the puzzle is not solvable, print a message stating the puzzle is not solvable. (It is possible that a puzzle may have no solution.)
      - `This puzzle has no solution`

4. If a solution does exist (the puzzle is solvable), then print a list of the grid showing the solution, if one exists. For example:

```
******
*1275*
*3445*
*..ZZ*
*66..*
******
```

Use of the C++ Standard Template Library

Your program is allowed to take full advantage of the C++ Standard Library. The following gives some suggestions on library elements that might be useful.

To store your data, you will need to create a class to hold each piece. The puzzle will be a collection of these pieces. Since the number of pieces is unknown, this should be dynamic. The use of the Standard Template Library (STL) classes of **vector** or **list** would be a good choice here. When trying to find the shortest number of moves, the breadth-first search algorithm will work. Since the breadth-first search uses a queue, the use of the STL class of **queue** is good choice when implementing a breadth-first search.

Algorithm Ideas and Hints

A good algorithm to solve this problem is to create a class that will hold a "snapshot" of the puzzle. A snapshot is to contain all needed information about the "current" state of the puzzle. The current state is the current position of all pieces in the puzzle and what moves it took to reach the current state from the initial state. So the snapshot needs two main sets of data: the pieces with their positions and a list of moves. The initial state/snapshot is the position of the pieces as given in the input file and zero moves (an empty move list). Then all of the snapshots that could be created from moving a single piece from the initial snapshot are added onto a queue (be sure to add the move information to the list of moves). If moving the piece causes piece Z to move to the right-most column of the puzzle, the puzzle is solved and the move list which contains the solution is printed. Then the first snapshot is removed from the queue and all of the snapshots that could be created from moving a single piece from this snapshot are added onto the queue. If we attempt to remove a snapshot from the queue and the queue is empty, the puzzle has no solution.

Of course, we have to make sure that arrangement of pieces only is added onto the queue one time; otherwise, we may get into an infinite loop. One way to do this is to create a simplified version of the current layout of the pieces and to store and compare this simplified version using an STL **set**. One way to create a simplified version is to create a string from the puzzle. The string would have (# of rows)x(# of column) characters and the first (row-length)-th character would have the piece names from the first row (using a space character for an empty position in the puzzle. , the second (row-length)-th characters would have the piece names from the second row... For example the initial puzzle from above would have the following 16 character string:

```
"12  3 44ZZ5 6657"
```

and the solution of the puzzle would have the following 16 character string:

```
"12753445  ZZ66  "
```

This idea is nice, since comparisons are already defined for strings, we don't have to make up some complicated algorithm to determine if two snapshots have all of their pieces in the same positions.

Identifying Piece Moves

What are the possible first moves from the initial puzzle shown above? The first piece you should check would be the Z piece, since moving that piece determines if the puzzle is solved and if a solution is found, the remaining pieces don't have to be moved. The following is a piece by piece listing of the first moves from the initial puzzle. These moves are given in:

1. the order the pieces were entered in the input file;
2. if a piece has multiple possible moves in multiple directions, the moves are done in order of: up, down, left, right; and
3. if a piece can move multiple spaces in a given direction, those are ordered from smallest movement to largest movement.

- Piece Z
  - Can not be moved
- Piece 1
  - Can not be moved
- Piece 2
  - Move down 1 space
  - Move right 1 space
  - Move right 2 spaces
- Piece 3
  - Move right 1 space
- Piece 4
  - Move up 1 space
  - Move left 1 space
- Piece 5
  - Can not be moved
- Piece 6
  - Can not be moved
- Piece 7
  - Move up 1 space

Therefore, after the first initial snapshot is taken care of, the breadth first search queue should have 7 snapshots on it. One for each of the moves mentioned above.

Multiple Classes

Your program is to include classes for the following 3 ideas in this program:

- Grid - containing the information needed to represent the current layout of pieces
- Piece - containing the information needed for each piece on the grid
- Movement - containing the information needed to specify a moved piece

Exactly what is in each of these classes is left up to you. However, these classes are to follow the concepts of proper object oriented design. These classes must not have any data members be publicly accessible. These classes must be written in their own .h file

In addition to using and submitting a makefile, this program will also require a 1-2 page write up of the data structures used in the program and the logical division of your program into multiple source code files (i.e. which routines are where). Remember that this write-up is to be written in ASCII format and is to be electronically turned in with your program. The name of this file should be "readme.txt".

Programming Style

Your program must be written in good programming style. This includes (but is not limited to) meaningful identifier names, a file header at the beginning of each source code file, a function header at the beginning of the function, proper use of blank lines and indentation to aide in the reading of your code, explanatory "value-added" in-line comments, etc.

Optional: You may work on this project with one other student currently taking CS 351. If you work on this project with a partner, you are to submit only one submission for the both of you. Gradescope will allow you to select your partner when submitting. It is your responsibility to properly do this.

The code submitted by you is to be the work created by the members of your group (or just by yourself if you are not working with another student).

There is some starter code and data files available in replit.com Teams for CS 351.

- https://replit.com/@CS351Spring2023/Sliding-Block-Puzzle