UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

School of Computer Science & Applied Mathematics

COMS3005A Assignment
Part 2: Move Generation

Simplifications

To make life slightly easier going forward, we will be simplifying the game slightly. In particular, you may ignore crocodiles, monkeys and elephants going forward. **This does not apply to part 1, so please complete that assuming that these pieces may exist.** From now on, you may assume that the game starts with none of these pieces on the board. The new starting position will be

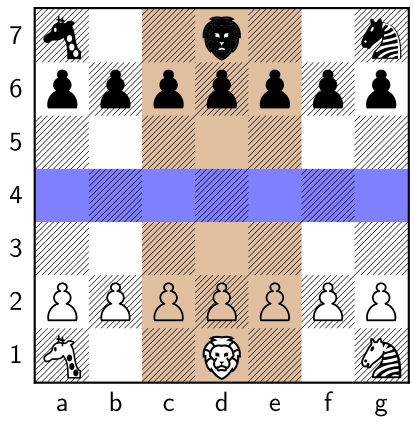


Figure 1: The starting position for White and Black excluding elephants, monkeys and crocodiles.

1 Introduction

In this section, we will need to generate the valid moves available to a player given the current state of the game. We will need to write code that accepts a game state as a FEN string and sets up the board (see Part 1 if you have no already done so). Once this has been done, the next step is to generate the moves available at the current position. This will depend on whether it is white or black to play.

2 Move Representations

For every submission, we will represent a move as a string `<start_square><end_square>` specifying the starting location of the piece to move and then square the piece ends up on. For example, the move `e3e4` represents a piece moving from `e3` to `e4`.

3 Input Hint

Hint: a reminder not to be careful when mixing `cin` with `getline`. An example of doing so is below:

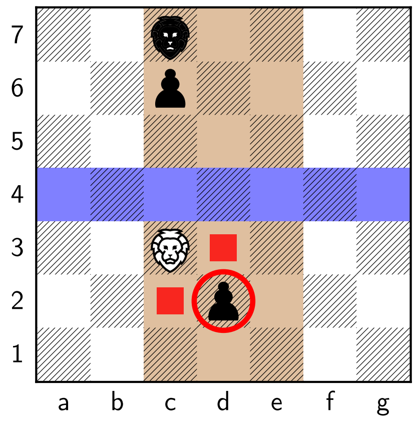
```
int N;
cin >> N;
cin.ignore(); //NB!
for (int i = 0; i < N; ++i) {
    string fen;
    getline(cin, fen);
}

```

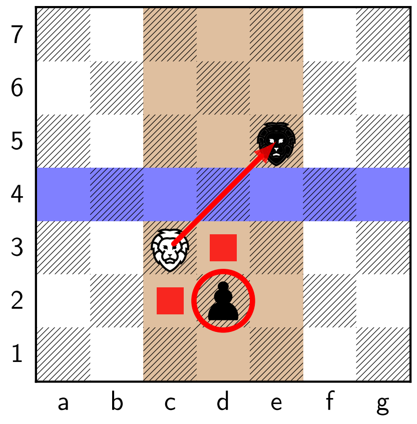
Submission 1: Generate Lion Moves

Write a C++ program that accepts a FEN string, stores the piece location information in appropriate data structures, and then outputs the valid lion moves that can be made by whichever player it is to move. As a reminder, the lion moves as follows:

Lion: The lion moves and captures one square in any direction (including diagonally), but it may not leave its 3×3 castle (highlighted in brown in the diagrams). However, there is one exception to this rule: a lion is allowed to move in a straight or diagonal line across the river if it is able to immediately **capture the opposing lion**. If a player's lion is captured, that player immediately loses the game. The diagram below shows the moves the white lion is able to make.



(a) In the current position, White's lion on c3 can move to the squares marked by a red square or can capture Black's pawn on d2 (red circle).



(b) Compared to the position on the left, the White lion on c3 can now also move across the river to capture the Black lion on e5 and win the game (red arrow)! The White lion could also potentially capture Black's lion anywhere along the c-file.

Input

The first line of input is N, the number of FEN strings that must be read in as input. N lines follow, with each line consisting of a single FEN string. You may assume that each FEN string is a valid position, and so will contain exactly one white lion and one black lion.

Output

For each FEN string i, output the valid lion moves. The moves should be printed in alphabetical order, and should be separated by a single space. Each move should be printed using the encoding described in Section 2. If there are no valid moves, nothing should be printed.

Example Input-Output

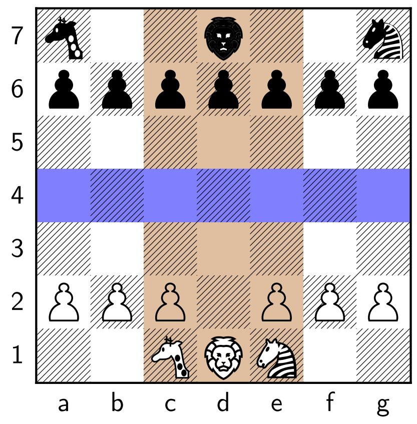
Sample Input

```
2
g2l2z/ppppppp/7/7/7/PPP1PPP/2GLZ2 w 4
1g1Gl2/P1P2P1/1P5/7/1Z3P1/1P5/4L2 b 79
```

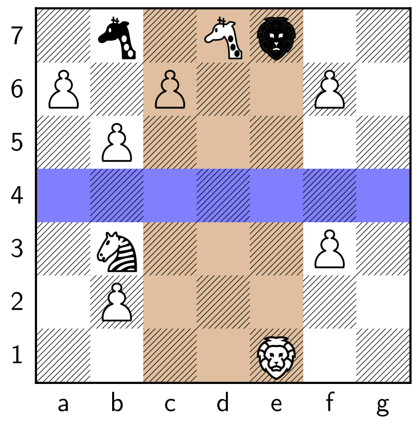
Sample Output

```
d1d2
e7d6 e7d7 e7e1 e7e6
```

Visualisation of Above Test Cases



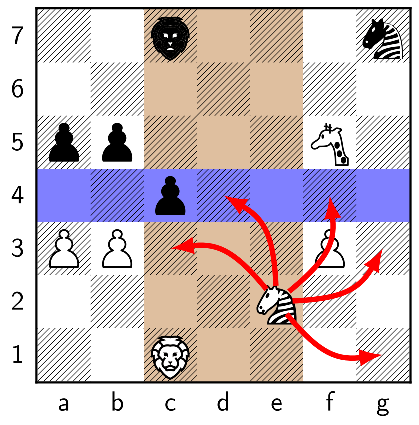
(a) The White lion can only move to d2, since it is blocked by its other pieces.



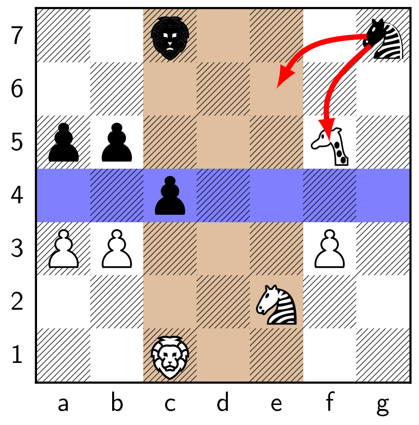
(b) The Black lion can move to the empty squares on d6 and e6 and can capture on d7. It cannot move to f7 or f6, since that would be outside of its 3×3 castle. It can also fly across the river to capture the white lion on e1.

Submission 2: Generate Zebra Moves

Write a C++ program that accepts a FEN string, stores the piece location information in appropriate data structures, and then outputs the valid zebra moves that can be made by whichever player it is to move. As a reminder, the zebra moves two squares vertically and one square horizontally, or two squares horizontally and one square vertically (with both forming the shape of an L). The zebra can jump over pieces to reach its destination.



(a) The White zebra on e2 can move to any square indicated by the arrows. It cannot move to c1, since that square is occupied by another White piece. Note that the pawn on f3 does not prevent the zebra from jumping to f4 or g3.



(b) The Black zebra can move to any square indicated by the arrows and can thus choose to capture the White piece on f5.

Input

The first line of input is N, the number of FEN strings that must be read in as input. N lines follow, with each line consisting of a single FEN string. The FEN string may contain 0 or 1 Black and White zebras.

Output

For each FEN string i, output the valid zebra moves. The moves should be printed in alphabetical order, and should be separated by a single space. Each move should be printed using the encoding described in Section 2. If there are no valid moves, or no zebras for the side to move exist, nothing should be printed.

Example Input-Output

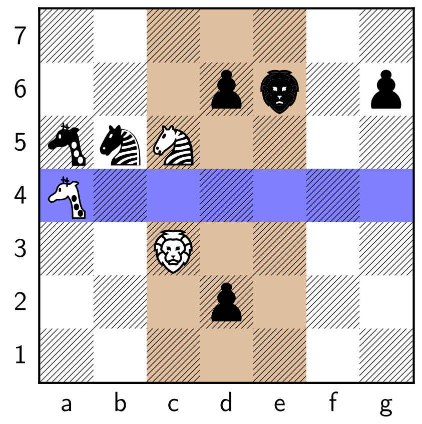
Sample Input

```
2
7/3pl1p/gzZ4/G6/2L4/3p3/7 w 45
7/7/2lP1p1/7/2ppgG1/1z4P/4L2 b 32
```

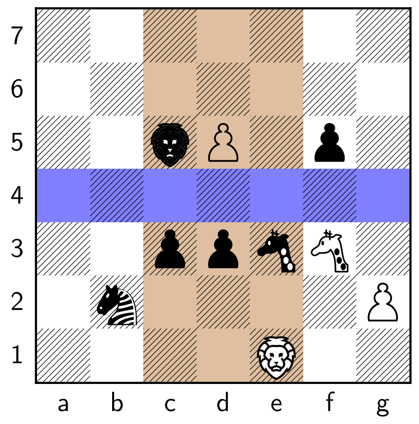
Sample Output

```
c5a6 c5b3 c5b7 c5d3 c5d7 c5e4 c5e6
b2a4 b2c4 b2d1
```

Visualisation of Above Test Cases



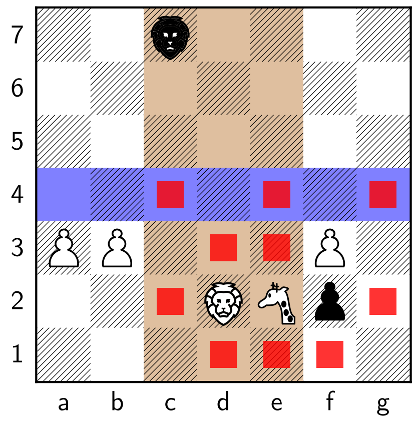
(a) The White zebra can move to empty squares on a6,b3,b7,d3,d7 and e4. It can also capture the Black lion on e6. It cannot move to a4, since it is occupied by a white piece.



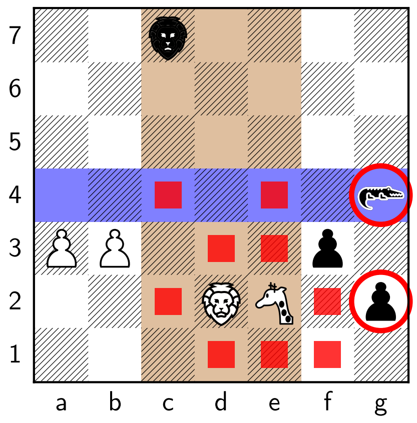
(b) The Black zebra can move to empty squares on a4, c4 and d1. It cannot move to d3, since it is occupied by a black piece.

Submission 3: Generate Giraffe Moves

Write a C++ program that accepts a FEN string, stores the piece location information in appropriate data structures, and then outputs the valid giraffe moves that can be made by whichever player it is to move. As a reminder, the giraffe can move one or two steps in any direction (orthogonally or diagonally), but **can only make captures two squares away**. If the giraffe moves two squares, the intermediate square is jumped.



(a) The White giraffe on e2 can move to any of the squares marked by a red square. Note that it can move to c2, g2 or g4 by jumping over the intermediate pieces. However, it cannot move to d2 (which is occupied by a White piece) and cannot capture the Black piece on f2, since it is only one square away.



(b) The White giraffe on e2 can move to any of the squares marked by a red square. It cannot move to or capture f3 since it is only one square away, but it can jump two squares to capture the Black piece on g4 or g2 (red circles).

Input

The first line of input is N, the number of FEN strings that must be read in as input. N lines follow, with each line consisting of a single FEN string. The FEN string may contain 0 or 1 Black and White giraffes.

Output

For each FEN string i, output the valid giraffe moves. The moves should be printed in alphabetical order, and should be separated by a single space. Each move should be printed using the encoding described in Section 2. If there are no valid moves, or no giraffes for the side to move exist, nothing should be printed.

Example Input-Output

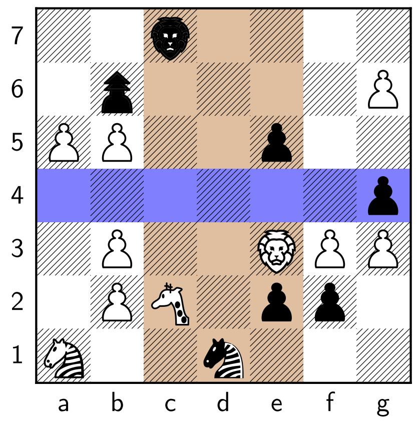
Sample Input

```
2
2l4/1s4P/PP2p2/6p/1P2LPP/1PG1pp1/Z2z3 w 23
5E1/Z5S/P3lP1/1p2P2/2g1pP1/P2P3/z1L4 b 35
```

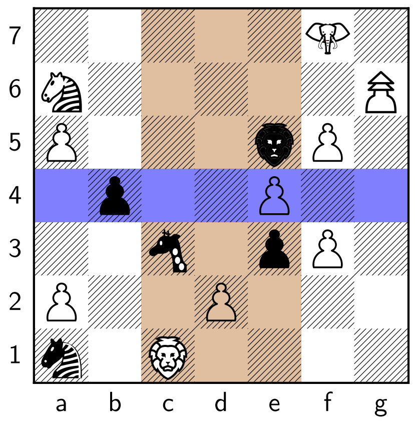
Sample Output

```
c2a2 c2a4 c2b1 c2c1 c2c3 c2c4 c2d2 c2d3 c2e2 c2e4
c3a3 c3a5 c3b2 c3b3 c3c1 c3c2 c3c4 c3c5 c3d3 c3d4 c3e1
```

Visualisation of Above Test Cases



(a) The White giraffe can move two squares away orthogonally or diagonally, unless there is a White piece occupying the target square. It can also move one square the same directions, but the square it is moving to must be empty (hence it cannot move to d1).

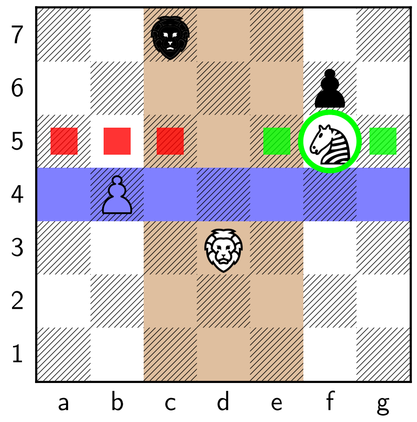


(b) The Black giraffe can move two squares away orthogonally or diagonally, unless there is a Black piece occupying the target square. It can also move one square the same directions, but the square it is moving to must be empty (hence it cannot move to d2).

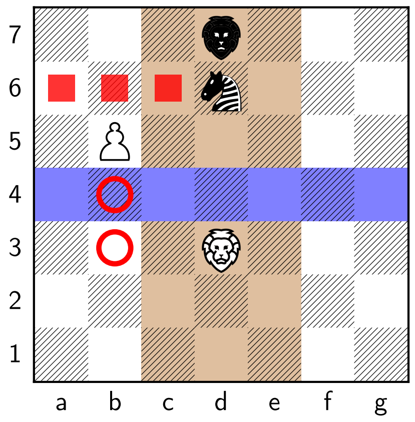
Submission 4: Generate Pawn Moves

Note: this section deals with regular pawns only. In the next section, we will consider superpawns.

Write a C++ program that accepts a FEN string, stores the piece location information in appropriate data structures, and then outputs the valid pawn moves that can be made by whichever player it is to move. As a reminder, a pawn can move and capture one step straight or diagonally forward. (Black pawns move forward towards rank 1, while White pawns move forward towards the rank 7.) When past the river, it can move (**but not capture**) one or two steps straight backwards. However, this is not a jumping move, and so the pawn may not move backwards if there is a piece in its path. When a pawn reaches the opposite end of the board, it is promoted (converted) to a superpawn.



(a) In this position, neither pawn has managed to cross the river completely yet. The White pawn on b4 can therefore only advance onto any square indicated by a red square, while the Black pawn on f6 can only advance to any square indicated by a green square or capture the White piece on f5 (green circle).



(b) The White pawn on b5 has crossed the river and so now can retreat one or two squares straight backwards (indicated by red circles) in addition to advancing forward (red squares).

Input

The first line of input is N, the number of FEN strings that must be read in as input. N lines follow, with each line consisting of a single FEN string.

Output

For each FEN string i, output the valid pawn moves. The moves should be printed in alphabetical order, and should be separated by a single space. Each move should be printed using the encoding described in Section 2. If there are no valid moves, or no pawns for the side to move exist, nothing should be printed.

Example Input-Output

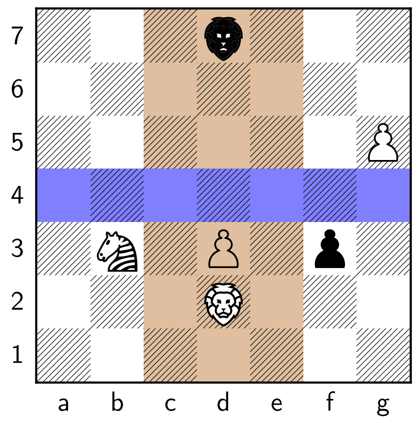
Sample Input

```
2
3l3/7/6P/7/1Z1P1p1/3L3/7 w 23
3l3/p6/7/7/1Z1P1p1/1p1L3/7 b 42
```

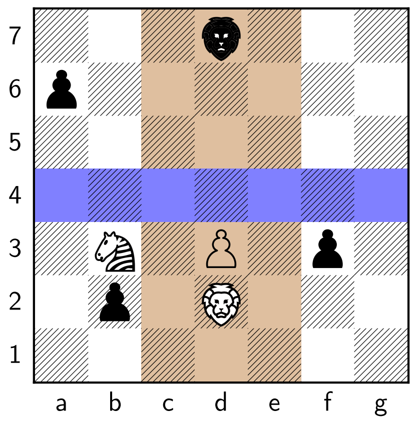
Sample Output

```
d3c4 d3d4 d3e4 g5f6 g5g3 g5g4 g5g6
a6a5 a6b5 b2a1 b2b1 b2c1 f3e2 f3f2 f3f4 f3f5 f3g2
```

Visualisation of Above Test Cases



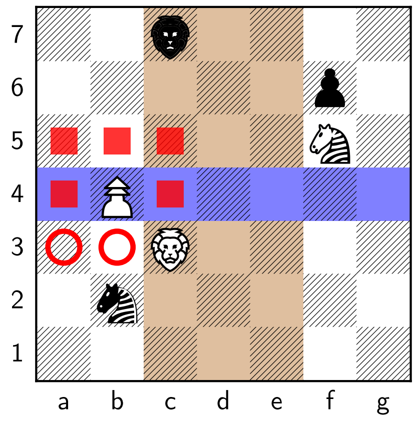
(a) The White pawn on d3 can move to the three squares in front of it. The pawn on g5 can move to the two square in front, and can also retreat to either g4 or g3, since it is beyond the river.



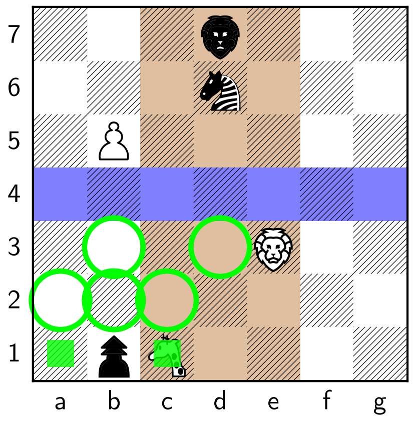
(b) The Black pawn on a6 can move forward to a5 or b5. The pawn on b1 can move to a1, b2 or c1 and promote, but it is blocked from retreated to b3 by the White zebra. The pawn on f3 can move forward to e2, f2 or g2 and can also retreat to either f4 or f5.

Submission 5: Generate Superpawn Moves

Write a C++ program that accepts a FEN string, stores the piece location information in appropriate data structures, and then outputs the valid superpawn moves that can be made by whichever player it is to move. As a reminder, a superpawn is identical to a pawn, except that it can move or capture one step sideways and when retreating, it can move one or two steps straight backwards or diagonally backwards. It may not capture when retreating and a retreat is not a jumping move, and so the superpawn may not move backwards if there is a piece in its path. Finally, its abilities do not depend on its position on the board (e.g. retreating can be done anywhere)



(a) In this position, the White superpawn on b4 can therefore only move to any square indicated by a red square and can retreat to squares marked by red circle. Note that it cannot retreat to c3 or d2, since it is blocked by the White piece. It also cannot retreat to b2 since it is occupied by a Black piece, and the superpawn may not capture when retreating.



(b) The Black superpawn on b1 can move to a1 or capture the White piece on c1. It can also retreat one or two squares backward or diagonally backward, as indicated by the green circles.

Input

The first line of input is N, the number of FEN strings that must be read in as input. N lines follow, with each line consisting of a single FEN string.

Output

For each FEN string i, output the valid superpawn moves. The moves should be printed in alphabetical order, and should be separated by a single space. Each move should be printed using the encoding described in Section 2. If there are no valid moves, or no superpawns for the side to move exist, nothing should be printed.

Example Input-Output

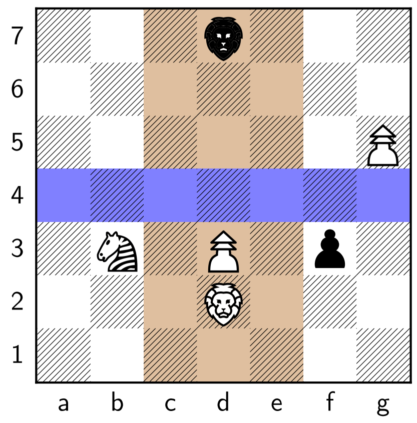
Sample Input

```
2
3l3/7/6S/7/1Z1S1p1/3L3/7 w 23
s2l3/7/7/7/1Z1P3/Ps1L3/7 b 42
```

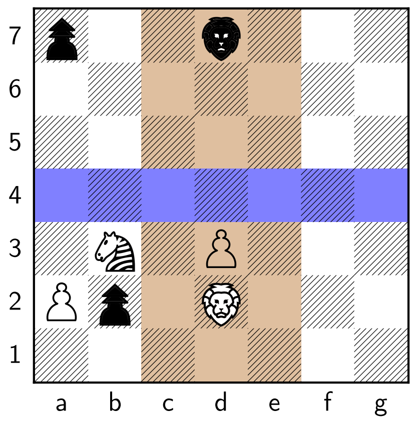
Sample Output

```
d3b1 d3c2 d3c3 d3c4 d3d4 d3e2 d3e3 d3e4 d3f1 g5e3 g5f4 g5f5 g5f6 g5g3 g5g4 g5g6
a7a6 a7b6 a7b7 b2a1 b2a2 b2a3 b2b1 b2c1 b2c2 b2c3 b2d4
```

Visualisation of Above Test Cases



(a) The White superpawn on d3 can move to all adjacent cells (except d2) and can also retreat to b1 and f1. The superpawn on g5 can move to the two square in front or sideways to f5. It can also retreat to e3, f4, g3 or g4.



(b) The Black superpawn on a7 can move forward to a6 or b6 or sideways to b7. The superpawn on b2 can move to a1, b1, c1 or c2, it can capture the White piece on a2 or it can retreat to a3, c3 or d4. It is blocked from retreated to b3 by the White zebra.