```
!pip install python-chess
```

```
Collecting python-chess
  Downloading python_chess-1.999-py3-none-any.whl.metadata (776 bytes)
Collecting chess<2,>=1 (from python-chess)
  Downloading chess-1.11.1.tar.gz (156 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 156.5/156.5 kB 12.1 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Downloading python_chess-1.999-py3-none-any.whl (1.4 kB)
Building wheels for collected packages: chess
  Building wheel for chess (setup.py) ... done
  Created wheel for chess: filename=chess-1.11.1-py3-none-any.whl size=148497 sha256=f6497053b45ccd7c0c8c0b7df5801515c0fd2be
  Stored in directory: /root/.cache/pip/wheels/2e/2d/23/1bfc95db984ed3ecbf6764167dc7526d0ab521cf9a9852544e
Successfully built chess
Installing collected packages: chess, python-chess
Successfully installed chess-1.11.1 python-chess-1.999
```

```
import chess
import chess.pgn
from chess import Board, Move
from enum import Enum
from google.colab import drive
import io
import itertools
import os
import pandas as pd
from pandas import DataFrame
import numpy as np
import sys
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils.rnn import pad_sequence
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from typing import Iterator, List
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
token = "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"
```

```
rm -rf /content/chess-ml
```

```
!git clone https://{token}@github.com/johnsonlarryl/chess-ml.git
```

```
Cloning into 'chess-ml'...
remote: Enumerating objects: 158, done.
remote: Counting objects: 100% (158/158), done.
remote: Compressing objects: 100% (104/104), done.
remote: Total 158 (delta 62), reused 134 (delta 44), pack-reused 0 (from 0)
Receiving objects: 100% (158/158), 101.61 KiB | 14.52 MiB/s, done.
Resolving deltas: 100% (62/62), done.
```

```
os.environ["POSTGRES_HOSTNAME"]=''
os.environ["POSTGRES_DATABASE"]=''
os.environ["POSTGRES_USERNAME"]=''
os.environ["POSTGRES_PASSWORD"]=''
os.environ["POSTGRES_TOTAL_GAMES"]=''
```

```
sys.path.append("/content/chess-ml/chess-ml-dao")
```

```
from chess_ml_dao.dao.postgres import PGNDAO
from chess_ml_dao.algo import ChessMoveModel
from chess_ml_dao.model.transformation import PlayerTurn
from chess_ml_dao.util.transformation import get_board_array, \
                                             get_move_mask, \
                                             get_players_turn_array, \
                                             generate_all_possible_white_promotion_moves, \
                                             generate_all_possible_black_promotion_moves, \
                                             generate_standard_possible_moves, \
```

```
                                      generate_standard_possible_moves, \
                                      generate_black_castling, \
                                      generate_white_castling, \
                                      POSSIBLE_MOVES


class ChessMoveModel(nn.Module):
    def __init__(self, number_of_possible_moves):
        super(ChessMoveModel, self).__init__()

        # Convolutional Layers
        self.conv1 = nn.Conv2d(in_channels=14, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=2, padding=1)  # Stride reduces spatial si
        self.bn2 = nn.BatchNorm2d(64)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        self.conv4 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=2, padding=1)  # Stride reduces spatial
        self.bn4 = nn.BatchNorm2d(256)

        self.conv5 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.bn5 = nn.BatchNorm2d(512)

        self.conv6 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.bn6 = nn.BatchNorm2d(512)

        # Global Average Pooling
        self.global_pool = nn.AdaptiveAvgPool2d((1, 1))  # Global pooling
        self.fc1 = nn.Linear(512, 512)
        self.bn_fc = nn.BatchNorm1d(512)
        self.fc2 = nn.Linear(512, number_of_possible_moves)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))  # Stride reduces spatial dimensions

        x = F.relu(self.bn3(self.conv3(x)))
        x = F.relu(self.bn4(self.conv4(x)))  # Stride reduces spatial dimensions

        x = F.relu(self.bn5(self.conv5(x)))
        x = F.relu(self.bn6(self.conv6(x)))

        # Global Average Pooling
        x = self.global_pool(x)  # Shape: (batch_size, 512, 1, 1)
        x = x.view(x.size(0), -1)  # Flatten to (batch_size, 512)

        # Fully Connected Layers
        x = F.relu(self.bn_fc(self.fc1(x)))
        x = self.fc2(x)

        return x



# Load your trained model
number_of_possible_moves=len(POSSIBLE_MOVES)
model = ChessMoveModel(number_of_possible_moves=number_of_possible_moves)
model.load_state_dict(torch.load('/content/drive/MyDrive/UNT/CSCE 5218/Semester Project/models/cnn.12.14.2024.11.46.pytorch'))
model.eval()  # Set the model to evaluation mode
```

```
<ipython-input-11-02908f1794d8>:4: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default
  model.load_state_dict(torch.load('/content/drive/MyDrive/UNT/CSCE 5218/Semester Project/models/cnn.12.14.2024.11.46.pytorc
ChessMoveModel(
  (conv1): Conv2d(14, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bn4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv6): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn6): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (global_pool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc1): Linear(in_features=512, out_features=512, bias=True)
```

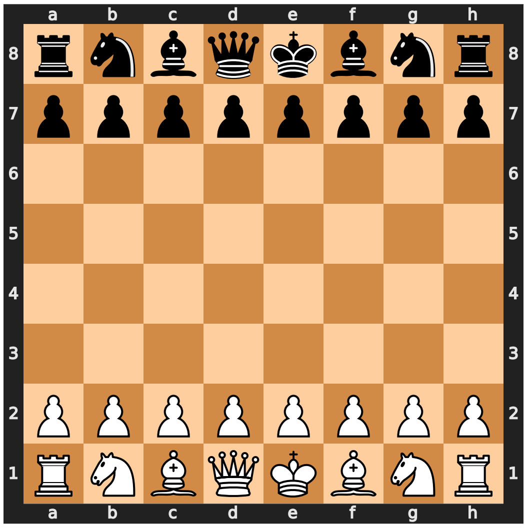
```
    (bn_fc): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (fc2): Linear(in_features=512, out_features=4548, bias=True)
  )


def prepare_input_tensor(board: chess.Board) -> torch.Tensor:
    board_array = get_board_array(board)

    players_turn = np.full((8, 8), 1 if board.turn == chess.WHITE else -1, dtype=np.float32)

    if board.turn:
      current_player = chess.WHITE
    else:
      current_player = chess.BLACK

    players_turn = get_players_turn_array(board, current_player)
    pieces = PGNDAO.get_piece_channels(board)

    board_features = np.vstack([
        np.expand_dims(board_array, axis=0),
        np.expand_dims(players_turn, axis=0),
        pieces
    ])

    input_tensor = torch.tensor(board_features, dtype=torch.float32).unsqueeze(0)  # Shape: (1, 14, 8, 8)

    return input_tensor



def make_move(board: chess.Board) -> str:
    """
    Generate a move for the computer using legal moves.
    """
    legal_moves = list(board.legal_moves)  # Get all legal moves
    if not legal_moves:
        raise ValueError("No legal moves available.")

    # Pick a random legal move (or implement your logic here)
    chosen_move = legal_moves[0]  # Example: pick the first legal move
    return chosen_move.uci()  # Return the move in UCI format



def filter_model_outputs(outputs: torch.Tensor, possible_moves: list, original_moves: list) -> torch.Tensor:
    # Map reduced moves to their indices in the original move set
    indices = [original_moves.index(move) for move in possible_moves]

    # Filter the logits to include only reduced moves
    return outputs[:, indices]



def generate_original_moves() -> list:
    # Standard moves
    standard_moves = generate_standard_possible_moves()

    # Promotions
    white_promotions = generate_all_possible_white_promotion_moves()
    black_promotions = generate_all_possible_black_promotion_moves()

    # Castling
    castling_moves = generate_white_castling() + generate_black_castling()

    # Combine all moves
    all_moves = standard_moves + white_promotions + black_promotions + castling_moves

    # Ensure the total matches 4612
    while len(all_moves) < len(POSSIBLE_MOVES):
        all_moves.append(f"placeholder_move_{len(all_moves)}")

    return all_moves

# Define ORIGINAL_MOVES
ORIGINAL_MOVES = generate_original_moves()
print(f"Number of ORIGINAL_MOVES: {len(ORIGINAL_MOVES)}")  # Should print 4612
```

```
Number of ORIGINAL_MOVES: 4548
```

```
def make_move(board: chess.Board) -> str:
    input_tensor = prepare_input_tensor(board)

    # Get model predictions
    with torch.no_grad():
        outputs = model(input_tensor)  # Shape: (1, num_original_moves)

    # Filter outputs to match the reduced move set
    outputs = filter_model_outputs(outputs, POSSIBLE_MOVES, ORIGINAL_MOVES)

    # Generate the move mask
    move_mask = get_move_mask(board)
    move_mask_tensor = torch.tensor(move_mask, dtype=torch.float32)

    # Apply the mask to outputs
    masked_outputs = outputs * move_mask_tensor

    # Handle edge case: no valid moves
    if move_mask_tensor.sum() == 0:
        if board.is_checkmate():
            raise ValueError("Checkmate: No valid moves available.")
        elif board.is_stalemate():
            raise ValueError("Stalemate: No valid moves available.")
        else:
            raise ValueError("No valid moves available; the game is over or an error occurred.")

    # Get the best move index
    _, best_move_index = masked_outputs.max(dim=1)

    # Map the index to a UCI move
    predicted_move = POSSIBLE_MOVES[best_move_index.item()]

    # Validate the predicted move
    legal_moves = {move.uci() for move in board.legal_moves}
    if predicted_move not in legal_moves:
        print(f"Invalid move generated: {predicted_move}. Falling back to best legal move.")
        # Find the best legal move
        legal_moves_indices = [POSSIBLE_MOVES.index(move) for move in legal_moves]
        legal_logits = outputs[0, legal_moves_indices]  # Logits for legal moves only
        best_legal_index = legal_moves_indices[legal_logits.argmax().item()]
        predicted_move = POSSIBLE_MOVES[best_legal_index]

    return predicted_move



def human_move(board: Board, move: str) -> Board:
  move = chess.Move.from_uci(move)
  board.push(move)

  return board


def computer_move(board: chess.Board) -> chess.Board:
    move = make_move(board)
    print(f"Computer: {move}")

    try:
        board.push(chess.Move.from_uci(move))
    except chess.InvalidMoveError:
        print(f"Invalid move generated: {move}")

    return board



def get_move_mask(board: chess.Board) -> np.ndarray:
    # Initialize the move mask
    move_mask = np.zeros(len(POSSIBLE_MOVES), dtype=np.float32)

    # Get all legal moves in UCI format (filtered by python-chess to handle checks)
    legal_moves = {move.uci() for move in board.legal_moves}

    # Iterate over all possible moves and mark legal ones
    for i, move in enumerate(POSSIBLE_MOVES):
        if move in legal_moves:
            move_mask[i] = 1
```

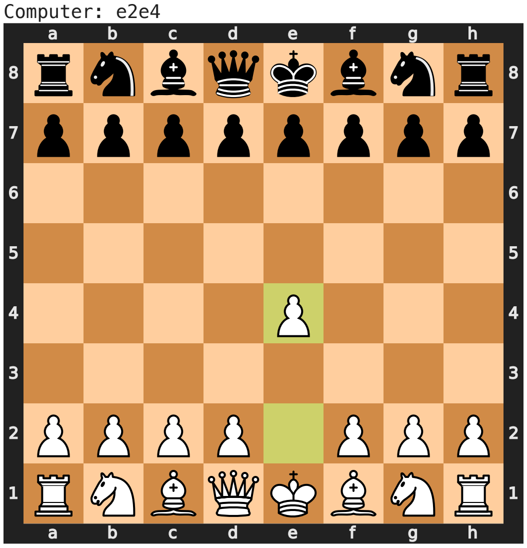
```
    return move_mask
```

```
# Create a new chess board (starting position)
board = chess.Board()
board
```

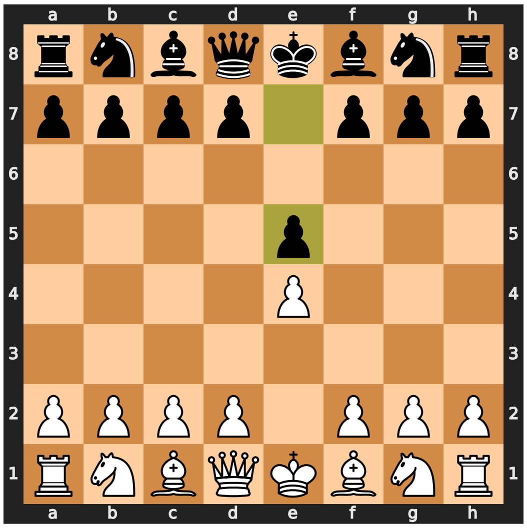


```
computer_move(board)
```

```
Computer: e2e4
```

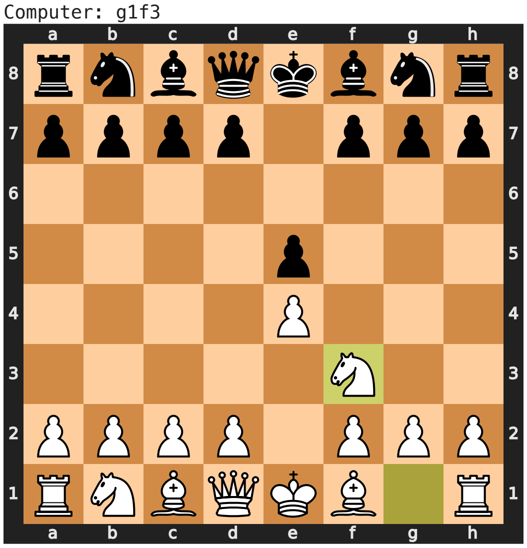


```
human_move(board, "e7e5")
```

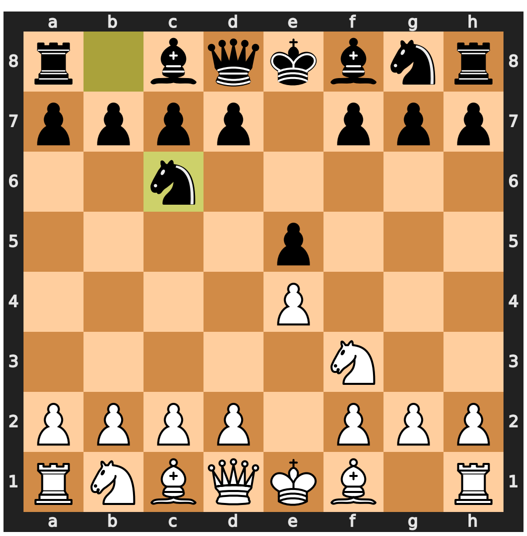


```
computer_move(board)
```

```
Computer: g1f3
```

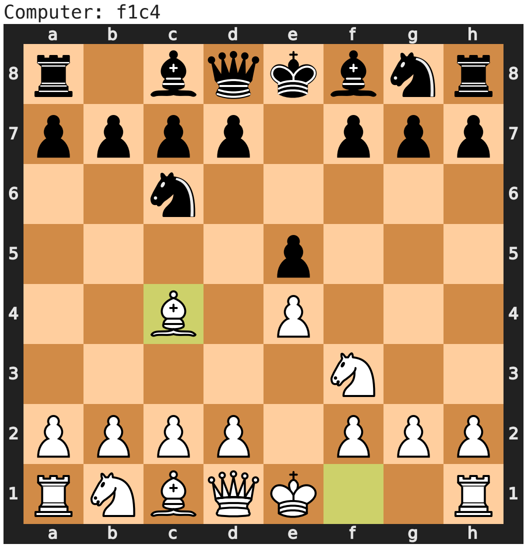


```
human_move(board, "b8c6")
```

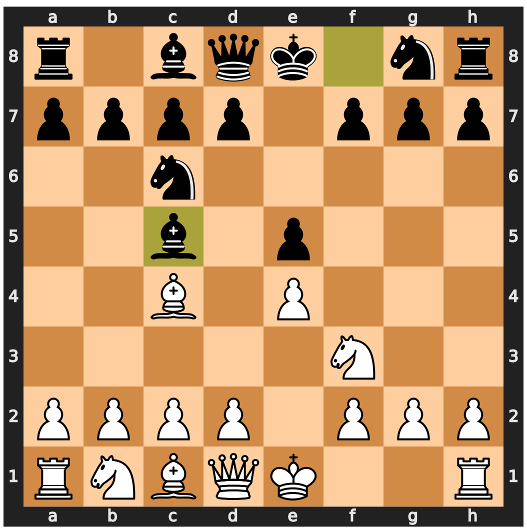


```
computer_move(board)
```

```
Computer: f1c4
```

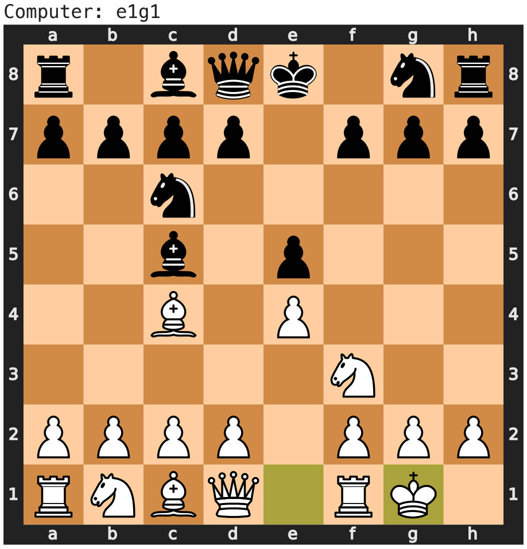


```
human_move(board, "f8c5")
```

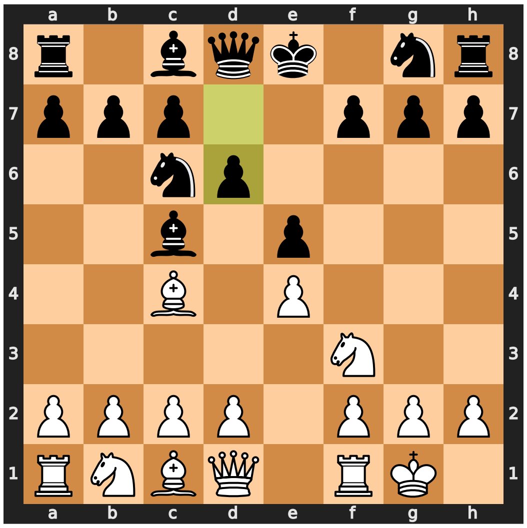


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: e1g1
```

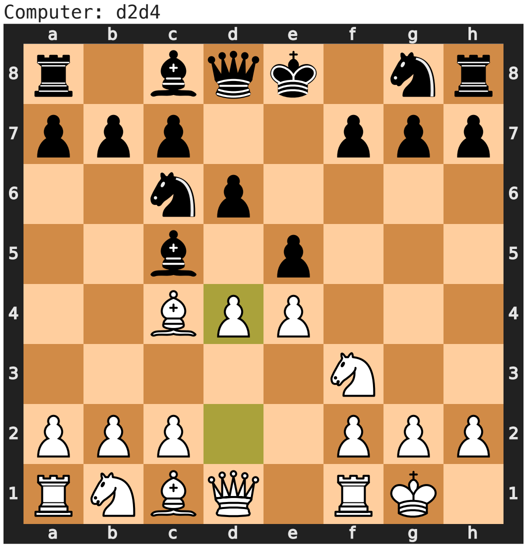


```
human_move(board, "d7d6")
```

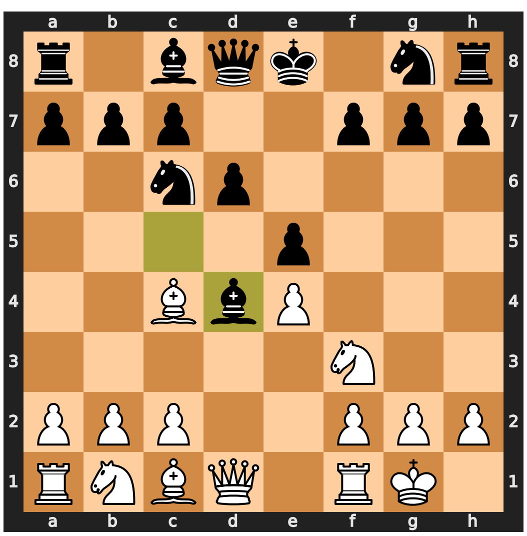


```
computer_move(board)
```

```
Computer: d2d4
```

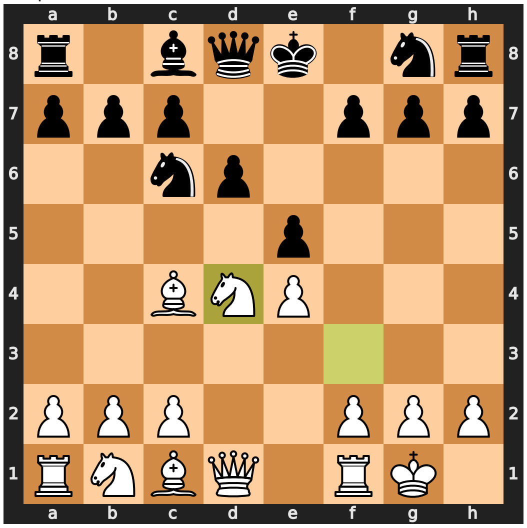


```
human_move(board, "c5d4")
```

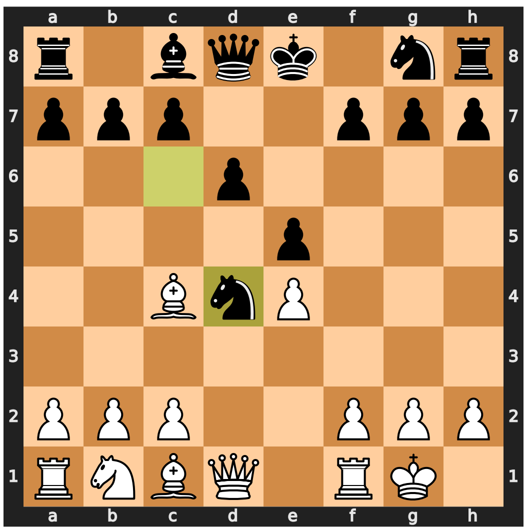


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: f3d4
```

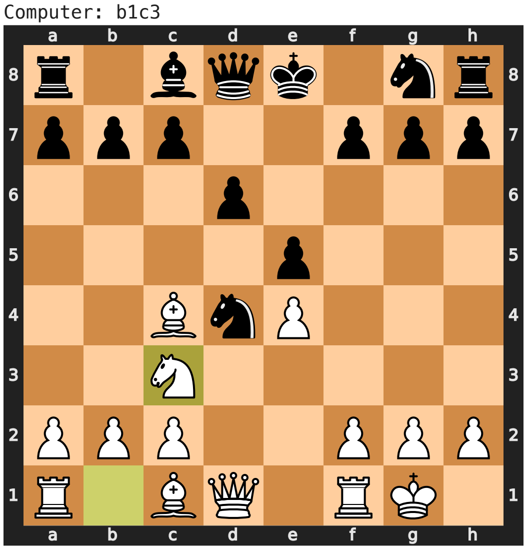


```
human_move(board, "c6d4")
```

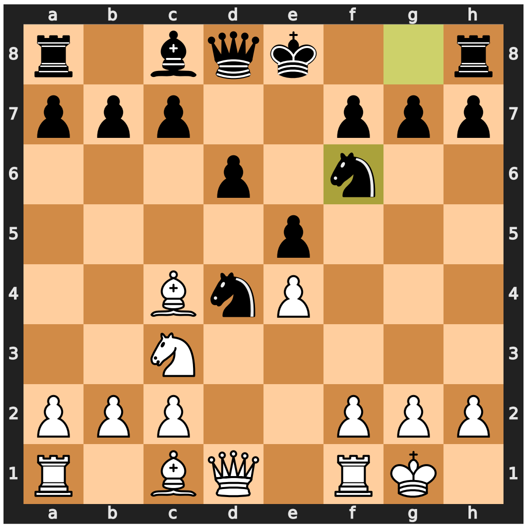


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: b1c3
```

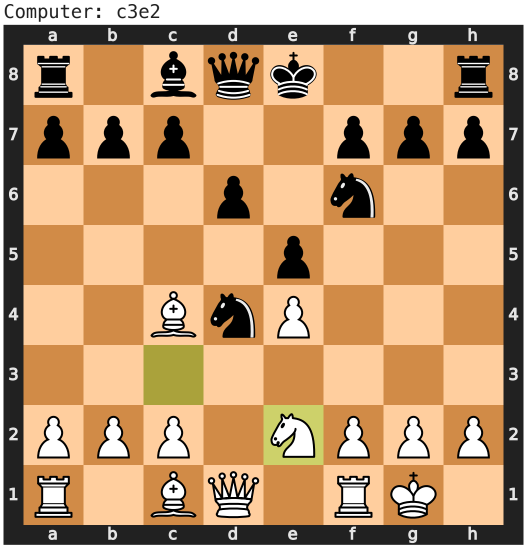


```
human_move(board, "g8f6")
```

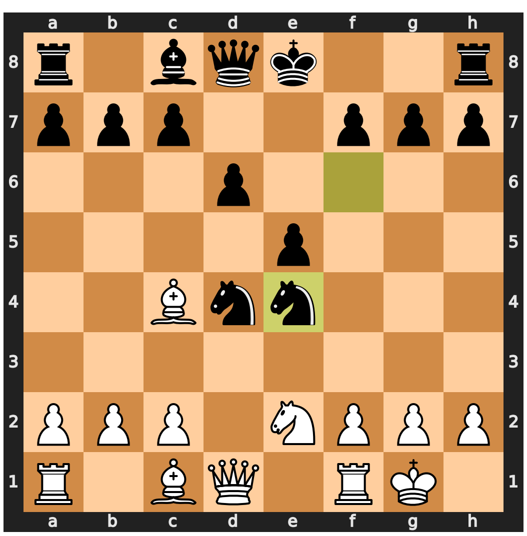


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: c3e2
```

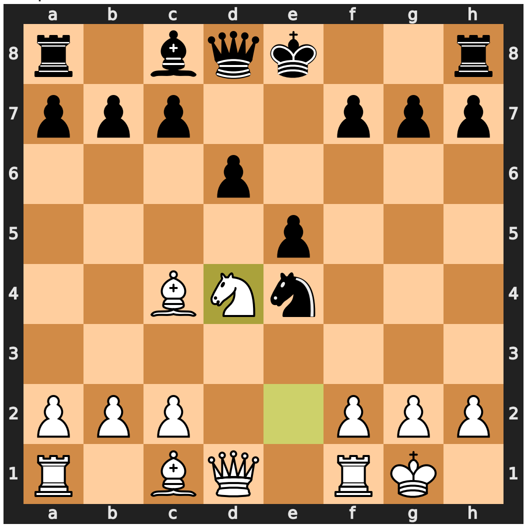


```
human_move(board, "f6e4")
```

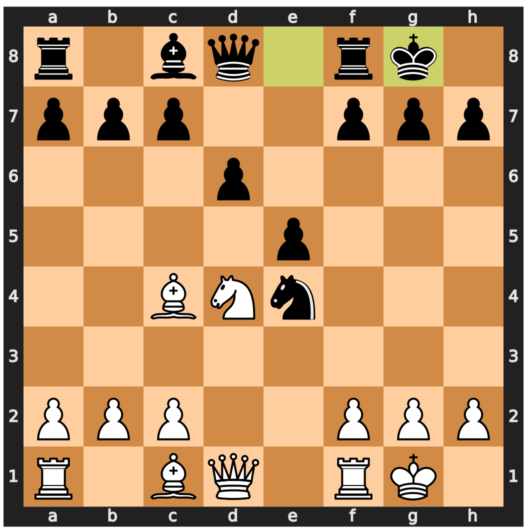


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: e2d4
```

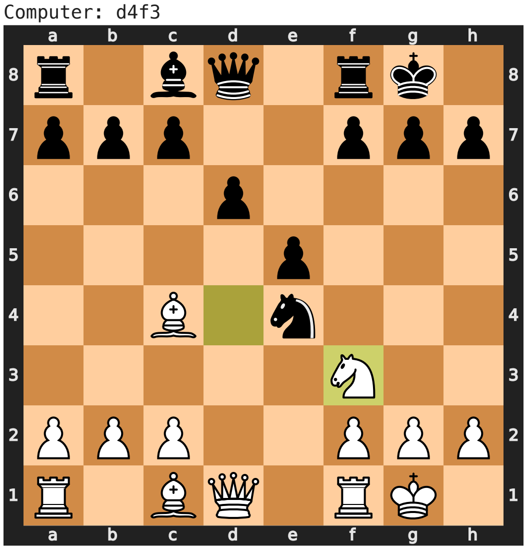


```
human_move(board, "e8g8")
```

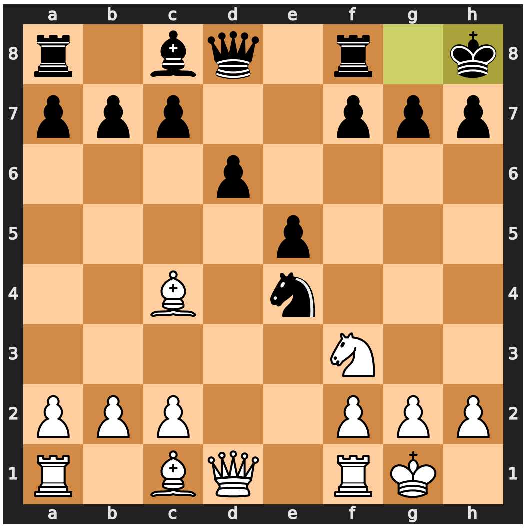


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: d4f3
```

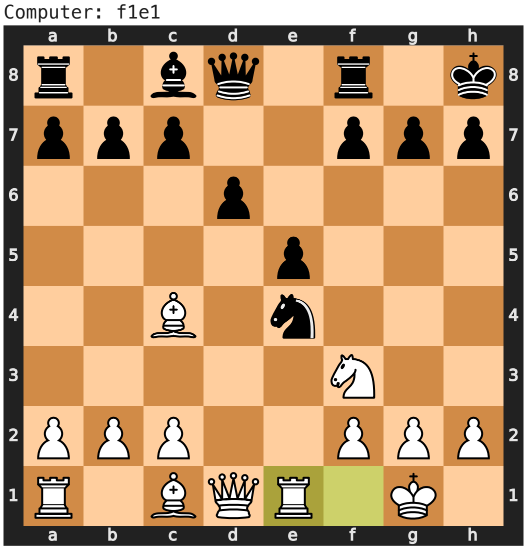


```
human_move(board, "g8h8")
```

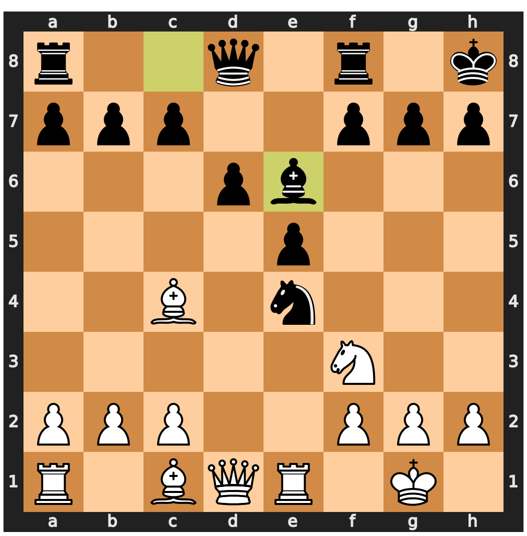


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: f1e1
```

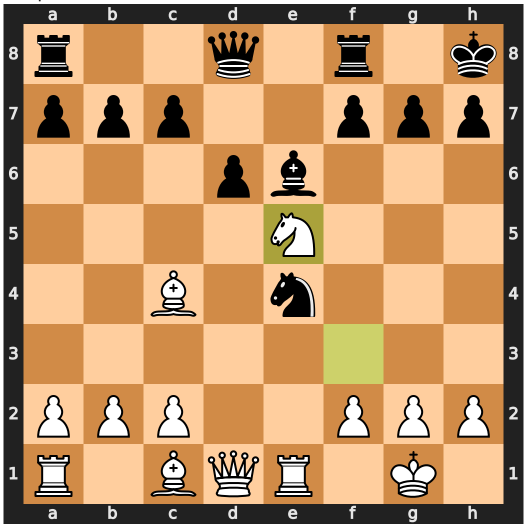


```
human_move(board, "c8e6")
```

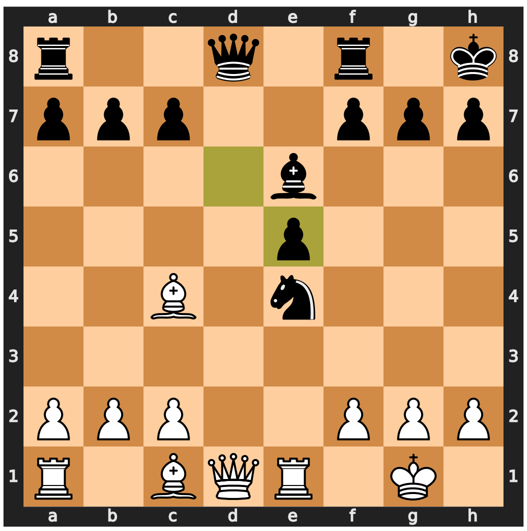


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: f3e5
```

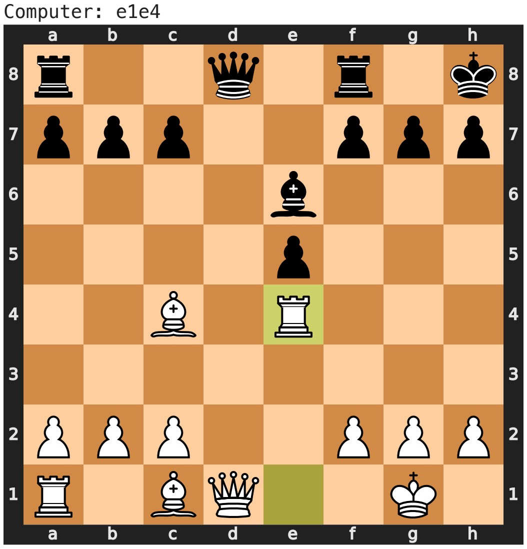


```
human_move(board, "d6e5")
```

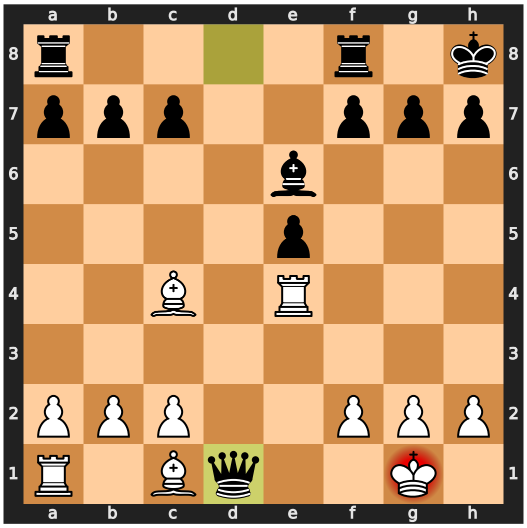


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: e1e4
```

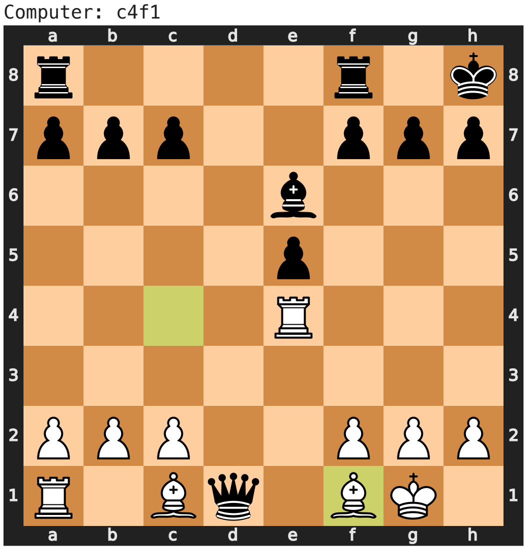


```
human_move(board, "d8d1")
```

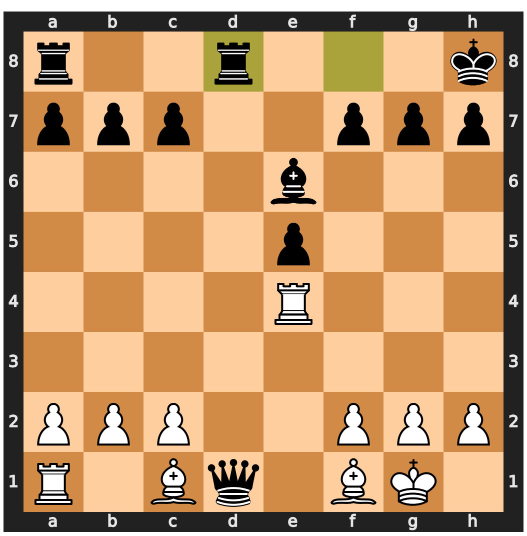


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: c4f1
```

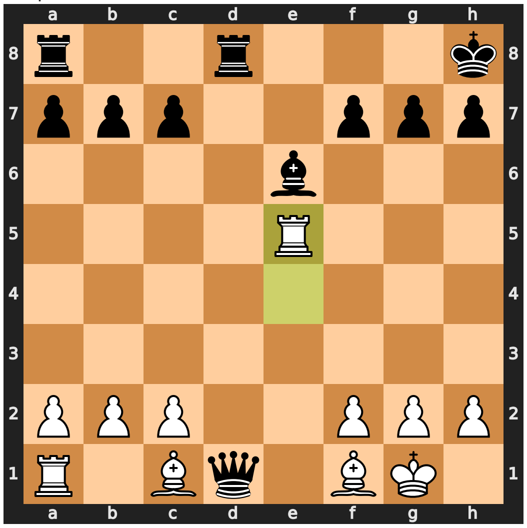


```
human_move(board, "f8d8")
```

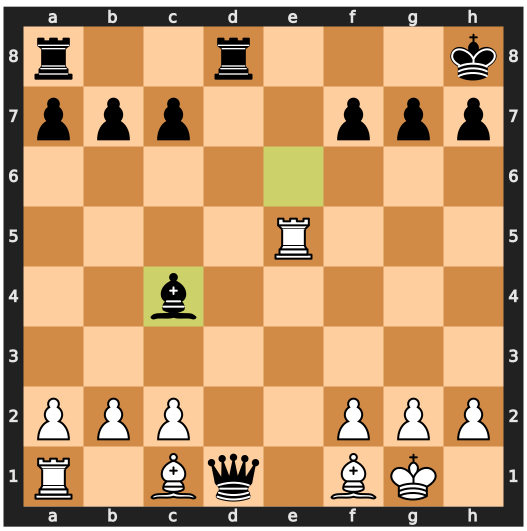


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: e4e5
```

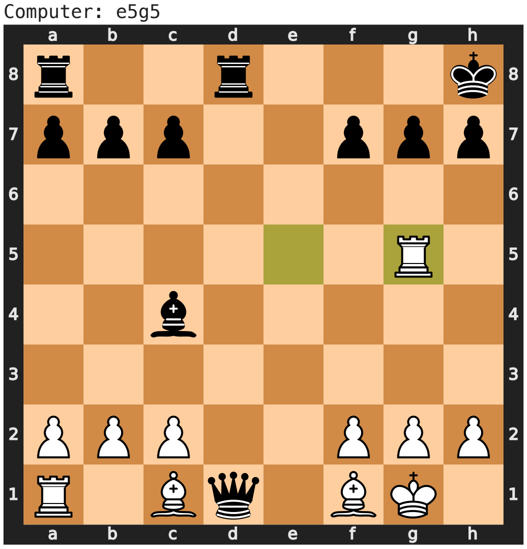


```
human_move(board, "e6c4")
```

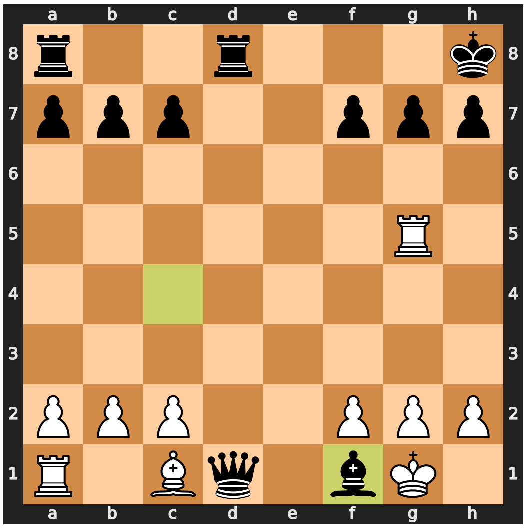


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: e5g5
```

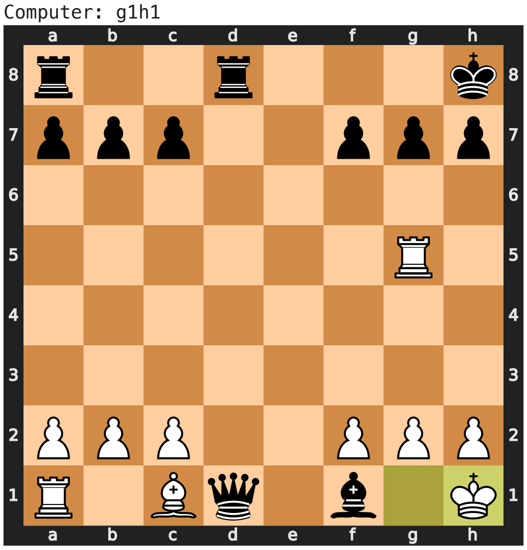


```
human_move(board, "c4f1")
```

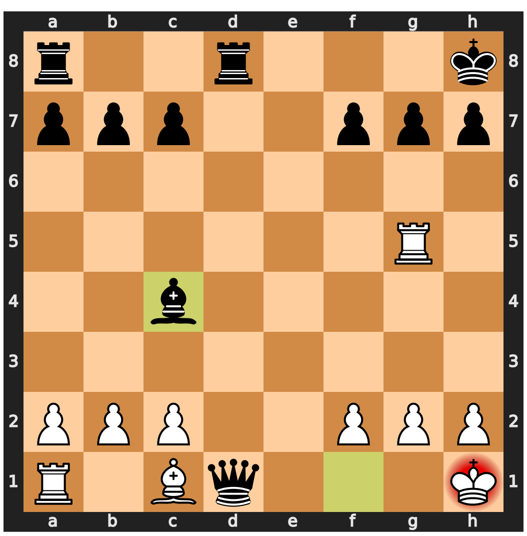


```
computer_move(board)
```

```
Invalid move generated: a1a2. Falling back to best legal move.
Computer: g1h1
```



```
human_move(board, "f1c4")
```



```
computer_move(board)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```