1 Chess

1.a Initialization

Initialization can be written as a simple assignment.

```
function board = initialize()
% Returns an initialized chessboard

board = [ 8  9 10 11 12 10  9  8;
          7  7  7  7  7  7  7  7;
          0  0  0  0  0  0  0  0;
          0  0  0  0  0  0  0  0;
          0  0  0  0  0  0  0  0;
          0  0  0  0  0  0  0  0;
          1  1  1  1  1  1  1  1;
          2  3  4  5  6  4  3  2 ];
```

1.b Board Validation

A simple validation can be done by counting the number of pieces and figuring out if there is an impossible scenario is in place. One can think of many complicated situations to create conditional statements and then combine them with logical operators for the validation. But since we will return only a true/false value, at any point when we detect an invalid situation, we can set the output to false and `return`.
Let's write a couple conditions. The first one would be to check whether each player has less than 9 pawns. Since pawns can be converted to other pieces when they advance to the other edge of the board, for each extra piece of other pieces, the number of lost pawns has to compensate for the difference. When we count the number of excess pieces the bishop is tricky! Remember that our validation function is incomplete.

```
function v = isvalid(board)
% Checks if the matrix provided as a chessboard is valid
% Consider counting pieces, board size etc. You don't need
% to consider pawns reaching the lowest or highest ranks.
% (but in the solutions we did.)

% Check board size
if ~isequal(size(board),[8 8]), v = 0; return;

% Check number of pawns
nwp = sum(sum(board == 1)); % or use nnz function!
nbp = sum(sum(board == 7));
num_pawns = nwp < 9 && nbp < 9;
if ~num_pawns, v = 0; return;

% Number of kings
nwk = sum(sum(board == 6));
```

```
nbk = sum(sum(board ==12));
% We can only have one king for each player
if nwk ~= 1, v = 0; return;
if nbk ~= 1, v = 0; return;

% Number of bishops
nwb = sum(sum(board == 4));
nbb = sum(sum(board ==10));

% Number of knights
nwn = sum(sum(board == 3));
nbn = sum(sum(board == 9));

% Number of rooks
nwr = sum(sum(board == 2));
nbr = sum(sum(board == 8));

% Number of queens
nwq = sum(sum(board == 5));
nbq = sum(sum(board ==11));

% For each player initially we have two bishops sitting
% on black or white squares. For a player, we can ignore
% at most one bishop sitting on a white/black squares.
% The r+c value is even for white squares, and odd for
% black squares!

[rwb, cwb] = find(board == 4);
wb = rwb+cwb; % r+c values for White's bishops
wb_on_white = sum(rem(wb,2) == 0);
wb_on_black = sum(rem(wb,2) == 1);
% Minimum excess bishops for White
nweb = max(wb_on_white - 1, 0) + max(wb_on_black - 1, 0);

[rbb, cbb] = find(board == 4);
bb = rbb+cbb; % r+c values for White's bishops
bb_on_white = sum(rem(bb,2) == 0);
bb_on_black = sum(rem(bb,2) == 1);
% Minimum excess bishops for Black
nbeb = max(wb_on_white - 1, 0) + max(wb_on_black - 1, 0);

% Excess pieces for White
nwe = nweb + max(nwn-2,0) + max(nwr-2,0) + max(nwq-2,0);
if (nwp-8) < nwe, v = 0; return;
% Excess pieces for Black
nbe = nbeb + max(nbn-2,0) + max(nbr-2,0) + max(nbq-2,0);
if (nbp-8) < nbe, v = 0; return;

% Two kings can't be neighbors!
[rwk, cwk] = find(board == 6);
[rbk, cbk] = find(board ==12);
if abs(rwk-rbk) == 1 || abs(cwk-cbk) == 1, v = 0; return;

% The pawns of a given player can't be at their home rank!
wp = sum(sum(board(1,:) == 7));
bp = sum(sum(board(8,:) == 1));
if wp > 0 || bp > 0, v = 0; return;
```

1.c Moving Pieces

First, we need to convert the string into row and column indices for the board. In order to move a piece from a source location to a destination, we read the value and copy it to the target. The source location becomes empty after the move.

```
function board = move_piece(board, move)
% Update the board, using the string move, which encodes a
% chess move in coordinate notation.

move = upper(move);

initial_file = 1 +  move(1) - 'A';
initial_rank = 9 - (move(2) - '0');

final_file   = 1 +  move(4) - 'A';
final_rank   = 9 - (move(5) - '0');

% Assuming move is valid!

board(final_rank,final_file) = board(initial_rank,initial_file);
board(initial_rank,initial_file) = 0;
```

1.d Performing Multiple Moves

We read a game recorded in coordinate notation in a text file. Each line contains a single move. We count until the `move_number` is reached and update the board after each move.

```
function board = read_moves(filename, move_number)
% Reads the list of moves from a text file, and returns
% the board state after a specified move (line) number.

f = fopen(filename,'r');

board = initialize();
move_id = 0;

while ~feof(f) && move_id < move_number
   move  = fscanf(f,'%c',5);
   temp  = fscanf(f,'%c',1); % 2 for Windows
   board = move_piece(board, move);
   move_id = move_id + 1;
end

fclose(f);
```

1.e Possible Moves

Assuming we have functions for each piece which return its possible moves, we can scan the board piece by piece to generate their possible moves and combine them together. We ignore castling, en passant etc. We also assume that we are not in check position, so we don't need to move our King or need to protect it right now.

```
function moves = possible_moves(board, bw, filename)
% Returns the possible moves on the current board.
% If bw is 0, moves correspond to White, if bw is 1
% moves correspond to Black. The moves are also written
% to a text file specified by its filename.
% The storage in the file should be in coordinate notation.
% Each move should occupy a single line.

t = [];
for r = 1:8
   for c = 1:8
      m = [];
      if bw == 0,
         switch board(r,c)
           case 1, m = pawn_moves(r,c,board,bw);
           case 2, m = rook_moves(r,c,board,bw);
           case 3, m = knight_moves(r,c,board,bw);
           case 4, m = bishop_moves(r,c,board,bw);
           case 5, m = queen_moves(r,c,board,bw);
           case 6, m = king_moves(r,c,board,bw);
         end
      else
         switch board(r,c)
           case 7, m = pawn_moves(r,c,board,bw);
           case 8, m = rook_moves(r,c,board,bw);
           case 9, m = knight_moves(r,c,board,bw);
           case 10,m = bishop_moves(r,c,board,bw);
           case 11,m = queen_moves(r,c,board,bw);
           case 12,m = king_moves(r,c,board,bw);
         end
      end

      t = append_moves(r,c,t,m);
   end
end

moves = write_moves_to_file(t,filename);
```

The target locations return by piece functions have to be appended with their source location. We can employ array expansion using comma and semicolons. The size of array `t` is (4×numberOfDiscoveredMovesSoFar), whereas the size of `m` is (2×numberOfMovesOfTheCurrentPiece). We can append two rows on top of `m`, and paste the result on the side of `t`.

```
function t = append_moves(r,c,t,m)
% Assuming m stores the target positions in two rows
l = size(m,2);
if l > 0,
t = [t, [[r*ones(1,l); c*ones(1,l)]; m]];
end
```

The subfunction which writes the moves to a text file is as listed as follows:

```
function moves = write_moves_to_file(t,fname)
moves = [ char(t(2,:)-1+'A');
          char(9-t(1,:)+'0');
          char(t(4,:)-1+'A');
          char(9-t(3,:)+'0')];
f = fopen(fname,'w');
for ii = 1:size(moves,2)
fprintf(f,'%c%c-%c%c\n',moves(1,ii),moves(2,ii),moves(3,ii),moves(4,ii));
end
fclose(f);
```

Helper Functions

In the code listings for piece move functions, we made use of the following helper functions.

```
function ob = onboard(r,c)
ob = r < 9 && r > 0 && c < 9 && c > 0;
```

```
function io = isopponent(r,c,board,bw)
io = any(board(r,c) == ((1:6)+(1-bw)*6));
```

```
function e = isitempty(r,c,board)
e = onboard(r,c) && (board(r,c) == 0);
```

```
function yn = cantake(r,c,board,bw)
yn = onboard(r,c) && isopponent(r,c,board,bw);
```

```
function yn = canmove(r,c,board,bw)
yn = onboard(r,c) && (cantake(r,c,board,bw) || isitempty(r,c,board));
```

Pawn

```
function t = pawn_moves(r,c,board,bw)

t = [];

d = -1+2*bw; % direction based on player's color
if onboard(r+d,c),
  if isitempty(r+d,c,board), t = [t,[r+d;c]]; end
end

for j = c-1:2:c+1 % check diagonals for opponent
   if onboard(r+d,j),
      if isopponent(r+d,j,board,bw)
          t = [t,[r+d;j]];
      end
   end
end

if bw == 1 && r == 2, % starting position?
   if isitempty(r+1,c,board) && isitempty(r+2,c,board),
      t = [t, [r+2;c]];
   end
end

if bw == 0 && r == 7, % starting position?
   if isitempty(r-1,c,board) && isitempty(r-2,c,board),
      t = [t, [r-2;c]];
   end
end
```

Bishop

We use an indicator whether we are *blocked* or not in some direction. By setting that condition to *true*, we can *break* out of the loop. Similar detection for blocking can be used for the rook as well.

```
function t = bishop_moves(r,c,board,bw)
t = [];
j = c;
blocked = 0;
for i = (r+1):8      % South-West direction
   j = j - 1;
   if j > 0 && possible(i,j), t = [t, [i;j]]; end
   if blocked, break; end
end

k = c;
blocked = 0;
for i = (r+1):8      % South-East direction
   k = k + 1;
   if k < 9 && possible(i,k), t = [t, [i;k]]; end
   if blocked, break; end
```

```
end

j = c;
for i = (r-1):-1:1  % North-West direction
   j = j - 1;
   if j > 0 && possible(i,j), t = [t, [i;j]]; end
   if blocked, break; end
end

k = c;
blocked = 0;
for i = (r-1):-1:1  % North-East direction
   k = k + 1;
   if k < 9 && possible(i,k), t = [t, [i;k]]; end
   if blocked, break; end
end
```

The subfunction `possible` is given as follows:

```
function p = possible(a,b)
   p = ~blocked && canmove(a,b,board,bw);
   if ~isitempty(a,b,board),
     blocked = 1;
   end
end % possible

end % bishop_moves
```

Knight

Knight makes L-shaped moves and it can jump over other pieces.

```
function t = knight_moves(r,c,board,bw)
t = [];

if r > 2
     if c > 1 && possible(r-2,c-1), t  = [t, [r-2; c-1]]; end
     if c < 8 && possible(r-2,c+1), t  = [t, [r-2; c+1]]; end
end

if c > 2
     if r > 1 && possible(r-1,c-2), t  = [t, [r-1; c-2]]; end
     if r < 8 && possible(r+1,c-2), t  = [t, [r+1; c-2]]; end
end

if r < 7
     if c > 1 && possible(r+2,c-1), t  = [t, [r+2; c-1]]; end
     if c < 8 && possible(r+2,c+1), t  = [t, [r+2; c+1]]; end
end

if c < 7
     if r > 1 && possible(r-1,c+2), t  = [t, [r-1; c+2]]; end
     if r < 8 && possible(r+1,c+2), t  = [t, [r+1; c+2]]; end
end
```

```
function p = possible(a,b)
p = canmove(a,b,board,bw);
end

end
```

Rook

```
function t = rook_moves(r,c,board,bw)
t = [];

blocked = 0;
for i = r-1:-1:1
   if possible(i,c), t = [t,[i;c]]; end
   if blocked, break; end
end

blocked = 0;
for i = r+1:8
   if possible(i,c), t = [t,[i;c]]; end
   if blocked, break; end
end

blocked = 0;
for i = c-1:-1:1
   if possible(r,i), t = [t,[r;i]]; end
   if blocked, break; end
end

blocked = 0;
for i = c+1:8
   if possible(r,i), t = [t,[r;i]]; end
   if blocked, break; end
end

function p = possible(a,b)
   p = ~blocked && canmove(a,b,board,bw);
   if ~isitempty(a,b,board),
     blocked = 1;
   end
end % possible

end % rook_moves
```

Queen

Queen is no different than a superposition of a bishop and a rook.

```
function t = queen_moves(r,c,board,bw)
t1 = bishop_moves(r,c,board,bw);
t2 = rook_moves(r,c,board,bw);
t  = [t1, t2];
```

King

King is slightly difficult since we need to take into account whether a destination square is targetted by oppenent's pieces. You have to consider the possible moves of your opponent in the next turn to figure out where you can't move to. You need to create a temporarily updated board and call a modified version of the `possible_moves` function. The complication comes from two kings being close to each other. Do you see why? In order to reduce the complexity, let's simply return a list of possible squares around it.

```
function t =  king_moves(r,c,board,bw)
t = [];

blocked = 0;

for i = r-1:r+1
  for j = c-1:c+1
     if possible(i,j), t = [t,[i;j]]; end
  end
end

function p = possible(a,b)
p = canmove(a,b,board,bw);
end % possible

end % king_moves
```