MyBot (class)

Type	Definition	Notes
Move	Think(Board board, Timer timer)	This is the function you need to implement for this challenge.

Board (class)

Type	Definition	Notes
void	MakeMove(Move move)	Updates the board state with the given move. The move is assumed to be legal, and may result in errors if it is not. Note that this doesn't make the move in the actual game, it just allows you to look ahead at future possibilities. So, making a move will toggle the IsWhiteToMove property, and calling GetLegalMoves() will now return the other player's moves. The move can be undone with the UndoMove() method.
void	UndoMove(Move move)	Undo a move that was made with the MakeMove method. Moves must be undone in reverse order, so for example, if moves A, B, C are made, they must be undone in the order C, B, A
bool	TrySkipTurn()	Try skip the current turn. This will fail and return false if the player is currently in check. Note that skipping a turn is not allowed during an actual game, but can be used as a search technique. Skipped turns can be undone with UndoSkipTurn()
void	ForceSkipTurn()	Forcibly skips the current turn. Unlike TrySkipTurn(), this will work even when in check, which has some dangerous side-effects if done: 1) Generating 'legal' moves will now include the illegal capture of the king. 2) If the skipped turn is undone, the board will now incorrectly report that the position is not check. Note: skipping a turn is not allowed in the game, but it can be used as a search technique. Skipped turns can be undone with UndoSkipTurn()
void	UndoSkipTurn()	Undo a turn that was succesfully skipped with TrySkipTurn() or ForceSkipTurn()
Move[]	GetLegalMoves(bool capturesOnly)	Gets an array of the legal moves in the current position. Can choose to get only capture moves with the optional 'capturesOnly' parameter.

void	GetLegalMovesNonAlloc(ref Span<Move> moves, bool capturesOnly)	(New in V1.13). Fills the given move span with legal moves, and slices it to the correct length. This gives the same result as the GetLegalMoves function, but allows you to be more efficient with memory by allocating moves on the stack rather than the heap. Can choose to get only capture moves with the optional 'capturesOnly' parameter.
		Example usage:
		System.Span<Move> moves = stackalloc Move[256];
		board.GetLegalMovesNonAlloc(ref moves);

bool	IsInCheck()	
bool	IsInCheckmate()	
bool	IsDraw()	Test if the current position is a draw due stalemate, repetition, insufficient material, or 50-move rule. Note: this function will return true if the same position has occurred twice on the board (rather than 3 times, which is when the game is actually drawn). This quirk is to help bots avoid repeating positions unnecessarily.
bool	IsInStalemate()	Test if the current position is a draw by stalemate (no legal moves, but not in check)
bool	IsRepeatedPosition()	Test if the current position has occurred at least once before on the board. This includes both positions in the actual game, and positions reached by making moves while the bot is thinking.
bool	IsInsufficientMaterial()	Test if there are sufficient pieces remaining on the board to potentially deliver checkmate. If not, the game is automatically a draw.
bool	IsFiftyMoveDraw()	Test if the current position is a draw by the 50-move rule (no pawn moves or captures in the last 50 moves)
bool	HasKingsideCastleRight(bool white)	Does the given player still have the right to castle kingside? Note that having the right to castle doesn't necessarily mean castling is legal right now (for example, a piece might be in the way, or player might be in check, etc).
bool	HasQueensideCastleRight(bool white)	Does the given player still have the right to castle queenside? Note that having the right to castle doesn't necessarily mean castling is legal right now (for example, a piece might be in the way, or player might be in check, etc).
bool	SquareIsAttackedByOpponent(Square square)	Is the given square attacked by the opponent? (opponent being whichever player doesn't currently have the right to move).
Square	GetKingSquare(bool white)	Gets the square that the king (of the given colour) is currently on.
Piece	GetPiece(Square square)	Gets the piece on the given square. If the square is empty, the piece will have a PieceType of None.
PieceList	GetPieceList(PieceType type, bool white)	Gets a list of pieces of the given type and colour
PieceList[]	GetAllPieceLists()	Gets an array of all the piece lists. In order these are: Pawns(white), Knights (white), Bishops (white), Rooks (white), Queens (white), King (white), Pawns (black), Knights (black), Bishops (black), Rooks (black), Queens (black), King (black).
string	GetFenString()	FEN representation of the current position.
ulong	GetPieceBitboard(PieceType type, bool white)	64-bit number where each bit set to 1 represents a square that contains a piece of the given type/colour.
ulong	WhitePiecesBitboard	64-bit number where each bit set to 1 represents a square that contains a white piece.
ulong	BlackPiecesBitboard	64-bit number where each bit set to 1 represents a square that contains a black piece.
ulong	AllPiecesBitboard	64-bit number where each bit set to 1 represents a square that contains a piece.
bool	IsWhiteToMove	Is it white's turn to move in the current position? Note that called MakeMove() and UndoMove() will toggle this value.
int	PlyCount	Number of ply (a single move by either white or black) played so far.
int	FiftyMoveCounter	Number of ply (a single move by either white or black) since the last pawn move or capture. If this value reaches a hundred (meaning 50 full moves without a pawn move or capture), the game is drawn.
ulong	ZobristKey	64-bit hash of the current position.
ulong[]	GameRepetitionHistory	Zobrist keys for all the positions played in the game so far. This is reset whenever a pawn move or capture is made, as previous positions are now impossible to reach again. Note that this is not updated when your bot makes moves on the board while thinking, but rather only when moves are actually played in the game.
string	GameStartFenString	FEN representation of the game's starting position.
Move[]	GameMoveHistory	All the moves played in the game so far. This only includes moves played in the actual game, not moves made on the board while the bot is thinking.
string	CreateDiagram(bool blackAtTop, bool includeFen, bool includeZobristKey)	Creates an ASCII-diagram of the current position. This can be printed to the console to help with debugging.
Board	Board.CreateBoardFromFEN(string fen)	Creates a board from the given fen string. Please note that this is quite slow, and so it is advised to use the board given in the Think function, and update it using MakeMove and UndoMove instead.

Move (struct)

Type	Definition	Notes
Move	new Move(string name, Board board)	Constructor for creating a move from its name (in UCI format). For example, to move from the square b1 to c3, the move string would be "b1c3". If the move is a pawn promotion, the promotion type must be added to the end: 'q' = queen, 'r' = rook, 'n' = knight, 'b' = bishop. So an example move would be "e7e8q". You'll typically want to get legal moves from the board, rather than creating them yourself.
Square	StartSquare	The square that this move is moving the piece from.
Square	TargetSquare	The square that this move is moving the piece to.
PieceType	MovePieceType	The type of piece that is being moved.
PieceType	CapturePieceType	If this is a capture move, the type of piece that is being captured.
PieceType	PromotionPieceType	If this is a pawn promotion, the type of piece that the pawn is being promoted to.
bool	IsCapture	
bool	IsEnPassant	
bool	IsPromotion	
bool	IsCastles	
bool	IsNull	
bool	Equals(Move otherMove)	Tests if two moves are the same. This is true if they move to/from the same square, and move/capture/promote the same piece type.
Move	Move.NullMove	Represents a null/invalid move, which can be used as a placeholder until a valid move has been found.

Square (struct)

Type	Definition	Notes
Square	new Square(string name)	Constructor for creating a square from its algebraic name (e.g. "e4")
Square	new Square(int fileIndex, int rankIndex)	Constructor for creating a square from a file and rank index [0, 7]
Square	new Square(int squareIndex)	Constructor for creating a square from a square index [0, 63]
int	File	Value from 0 to 7 representing files 'a' to 'h'
int	Rank	Value from 0 to 7 representing ranks '1' to '8'
int	Index	Value from 0 to 63. The values map to the board like so: 0 – 7 : a1 – h1 8 – 15 : a2 – h2 ... 56 – 63 : a8 – h8
string	Name	The algebraic name of the square, e.g. "e4"

Piece (struct)

Type	Definition	Notes
Piece	new Piece(PieceType type, bool isWhite, Square square)	Constructor for creating a new piece. You'll typically want to get pieces from the board, rather than constructing them yourself.
bool	IsWhite	
PieceType	PieceType	
Square	Square	The square that the piece is on. Note: this value will not be updated if the piece is moved (it is a snapshot of the state of the piece when it was looked up).
bool	IsPawn	
bool	IsKnight	
bool	IsBishop	
bool	IsRook	
bool	IsQueen	
bool	IsKing	
bool	IsNull	This will be true if the piece was retrieved from an empty square on the board

PieceType (enum)

		None = 0, Pawn = 1, Knight = 2, Bishop = 3, Rook = 4, Queen = 5, King = 6

PieceList (class)

Type	Definition	Notes
int	Count	The number of pieces in the list
bool	IsWhitePieceList	True if the pieces in this list are white, false if they are black
PieceType	TypeOfPieceInList	The type of piece stored in this list (a PieceList always contains only one type and colour of piece)
Piece	GetPiece(int index)	Get the i-th piece in the list.

Timer (class)

Type	Definition	Notes
int	GameStartTimeMilliseconds	The amount of time (in milliseconds) that each player started the game with
int	IncrementMilliseconds	The amount of time (in milliseconds) that will be added to the remaining time after a move is made
int	MillisecondsElapsedThisTurn	The amount of time elapsed since the current player started thinking (in milliseconds)
int	MillisecondsRemaining	The amount of time left on the clock for the current player (in milliseconds)
int	OpponentMillisecondsRemaining	The amount of time left on the clock for the other player (in milliseconds)

BitboardHelper (static class)

Type	Definition	Notes
		A bitboard is a 64 bit integer (ulong), in which each bit that is set to 1 represents something about the state of the corresponding square on the board (such as whether a particular type of piece is there, etc.) The 64 bits map to the squares like so: Bit 0 (LSB) to bit 7 : 'a1' to 'h1'. Bit 8 to bit 15 : 'a2' to 'h2' ... Bit 56 to bit 63 : 'a8' to 'h8'.
void	SetSquare(ref ulong bitboard, Square square)	Set the given square on the bitboard to 1.
void	ClearSquare(ref ulong bitboard, Square square)	Clear the given square on the bitboard to 0.
void	ToggleSquare(ref ulong bitboard, Square square)	Toggle the given square on the bitboard between 0 and 1.
bool	SquareIsSet(ulong bitboard, Square square)	Returns true if the given square is set to 1 on the bitboard, otherwise false.
int	ClearAndGetIndexOfLSB(ref ulong bitboard)	Returns index of the first bit that is set to 1. The bit will also be cleared to zero. This can be useful for iterating over all the set squares in a bitboard
int	GetNumberOfSetBits(ulong bitboard)	Returns the number of bits that set to 1 in the given bitboard.
ulong	GetSliderAttacks(PieceType type, Square square, Board board)	Gets a bitboard where each bit that is set to 1 represents a square that the given sliding piece type is able to attack. These attacks are calculated from the given square, and take the given board state into account (so attacks will be blocked by pieces that are in the way). Valid only for sliding piece types (queen, rook, and bishop).
ulong	GetSliderAttacks(PieceType type, Square square, ulong blockers)	Gets a bitboard where each bit that is set to 1 represents a square that the given sliding piece type is able to attack. These attacks are calculated from the given square, and take the given blocker bitboard into account (so attacks will be blocked by pieces that are in the way). Valid only for sliding piece types (queen, rook, and bishop).
ulong	GetKnightAttacks(Square square)	Gets a bitboard of squares that a knight can attack from the given square.
ulong	GetKingAttacks(Square square)	Gets a bitboard of squares that a king can attack from the given square.
ulong	GetPawnAttacks(Square square, bool isWhite)	Gets a bitboard of squares that a pawn (of the given colour) can attack from the given square.
ulong	GetPieceAttacks(PieceType type, Square square, Board board, bool isWhite)	Returns a bitboard where each bit that is set to 1 represents a square that the given piece type is able to attack. These attacks are calculated from the given square, and take the given board state into account (so queen, rook, and bishop attacks will be blocked by pieces that are in the way). The isWhite parameter determines the direction of pawn captures.
ulong	GetPieceAttacks(PieceType type, Square square, ulong blockers, bool isWhite)	Returns a bitboard where each bit that is set to 1 represents a square that the given piece type is able to attack. These attacks are calculated from the given square, and take the given blockers bitboard into account (so queen, rook, and bishop attacks will be blocked by pieces that are in the way). The isWhite parameter determines the direction of pawn captures.
void	VisualizeBitboard(ulong bitboard)	A debug function for visualizing bitboards. Highlights the squares that are set to 1 in the given bitboard with a red colour, and the squares that are set to 0 with a blue colour.