BitBoard Report

Design and Implementation of the Utility Class

The utility class serves as the primary class for dealing with the necessary bitwise operations and arithmetic computations.

- Bit Manipulation Methods
    - setBit(long num, long bitPosition)
        - This method will set the bit in the specified position in the long.
    - clearBit(long num, long bitPosition)
        - This method will clear the bit in the specified position in the long.
    - toggleBit(long num, long bitPosition)
        - This method will toggle the bit in the specified position in the long.
    - getBit(long num, long bitPosition)
        - This method will return the value of the specified bit in the long.
- Arithmetic Methods
    - add(long a, long b)
        - This method will add two longs.
    - subtract(long a, long b)
        - This method will subtract two longs.
    - multiply(long a, long b)
        - This method will multiply two longs.
    - divide(long a, long b)
        - This method will divide two longs.

- Conversion Methods
    - decimalToBinary(long decimal)
        - This method will convert a decimal number into a binary number in String form.
    - binaryToDecimal(String binary)
        - This method will convert a binary number in String form to a decimal number.
    - decimalToHex(long decimal)
        - This method will convert a decimal number to a hexadecimal number in Strong form.
    - hexToDecimal(String hex)
        - This method will convert a hexadecimal number in String form to a decimal number.

How Bitwise Operations and Binary Arithmetic Were Used in the Utility Class and Bitboard

Since the entire game is represented as a bitboard, I was able to make quite a lot of use of the bitwise operations and binary arithmetic.

- Determining if a square is occupied.
    - Through use of the getBit() method in the Utility class, I could effectively determine whether or not the selected square (or bit) was occupied.
- Making a move

        - When a player would make a move, they would first have their old position cleared to represent it now being empty, and then their new position set to represent that the position is now taken. I utilized the setBit() and clearBit() methods from the Utility class.
- Displaying the current board state
    - In order for the player to actually understand and see the current state of the game, I utilized the displayBoard() function in the Board class, which itself used the getBit() function from the Utility class to determine whether or not a square was occupied by a certain player, and whether or not that square was occupied by a king or normal piece.
- Arithmetic
    - Arithmetic operations were also performed throughout the entire Board class, using the Utility functions add(), subtract(), multiply(), and divide().
- Conversions for Binary and Hexadecimal
    - After every move, the user-friendly board state would be visualized, and the board state would also be printed in binary and hexadecimal. This would make use of the Utility functions decimalToBinary() and decimalToHex()

Significance of Bitwise Operations in Low-Level Computing and Game Development

Bitwise operations are incredibly important in the realm of low-level computing and game development. Since data is stored in the form of bits, bitwise operations enable faster and more memory-efficient methods.

- Bitwise operations can be far faster compared to traditional arithmetic operations, since the CPU of a computer works directly with these kinds of operations and parameters (bits). These operations will work directly on bits, and as a result of this and the aforementioned stuff, will significantly speed up the processes related to whatever is going on, such as when you might be performing a similar operation in multiple different points. For game development specifically, we are storing our entire game through a 64-bit long integer, which means that we can use these fast operations on our game and speed things up.
- Another really useful thing is related to how we are storing our game itself. As mentioned before, the entire board and the pieces are stored in 64-bit long integers, which is incredibly memory-efficient, especially compared to using something such as an array. In game development, you can store various aspects of your game in long integers which will for one be more memory efficient, and two enable you to use bitwise operations. For Chess for example, you could have a long to represent the board, each players pawns, each players knights, etc.. and that would be really efficient.
- The key idea here is efficiency in terms of memory and speed. Bitwise operations are just incredibly fast compared to other tools, and if we are storing our data using something memory-efficient such as a long or short, then we are freeing up memory that could be used somewhere else. Although it can make things appear more complex and rudimentary, if speed and memory are a primary concern, then this is absolutely something to think about.

The Process of Converting Data Between Different Formats

To properly and effectively display the current state of the board/game over and over again, it was important to implement proper data conversions between different formats.

- Although I didn't really need or make much use of data conversions, I made sure to always display the state of the board in both binary and hexadecimal.
- Through the use of my decimalToBinary() Utility function, I could print out the state of the board in binary, which would allow the user to see which "squares" or positions are occupied. The decimalToBinary() function would take a long integer and then convert it into a binary string.
- In a similar fashion, I utilized my decimalToHex() Utility function to also represent the board in a different format, this time through hexadecimal.

Challenges Faced and How They Were Overcome

- The earliest and most frustrating challenge that I faced right at the beginning was figuring out how to represent the Checkers Board using a long. What was difficult was me not fully understand how to properly structure the long, and then me also needing to figure out how to represent all the relevant information using longs. I had to create a long for the white pieces, black pieces, and kings. I was able to overcome this challenge mainly through just a ton of trial and error and googling and stack overflow.
- Another challenge that I faced was dealing with validating moves. Determining whether or not a move was legal or illegal turned out to be quite frustrating. Checking that the destination square, or the square that the player was trying to move to, was empty honestly wasn't too difficult. My difficulty lied within trying to determine whether or not

the move was a valid diagonal move if the piece was not a king, and if it were a king to allow for forward and backward diagonal moves. Overcoming this part was arguably more tedious than the first challenge, but I was thankfully able to figure it out. I found a video on someone implementing a Bitboard for Chess, so I just watched their video and saw how they dealt with a similar situation for their bishops, since they can only move diagonally, and from there I had a pretty good idea of what I needed to do.

- The final difficulty that I will touch on had to do with actually displaying the board. Far too much time was spent on displaying the board correctly and making it look somewhat decent. Just figuring out how to go about looping through the rows and columns and then what to do for each square proved to be a challenge. Similar to what I did with the move validation aspect, I watched the video for the Chess Bitboard and was able to get an idea of how to properly display the board, even though there were some differences that I had to fill in.