Big-O Analysis

Grant Gapinski & Bailey Middendorf

makeMove(...)

```
/**
 * Makes a move for the current player
 * @param x - The x coordinate to move to
 * @param y - The y coordinate to move to
 * @throws Exception -
 *  If the game is over
 *  If the space is invalid
 */
public void makeMove(int x, int y) throws Exception {
  if(this.gameOver) {
    throw new Exception("Game is Over");
  }
  Player pointer = null;
  if(this.turn == 1) {
    pointer = playerOne;
  } else if(this.turn == 2) {
    pointer = playerTwo;
  }
  if(validPlayerMove(pointer, x, y)) {
    // get the current space location of the player
    Space old =
this.board.get(pointer.getPosition()[0]).get(pointer.getPosition()[1]);
    // Set the intended space as the space holding the player
    this.board.get(x).get(y).setPlayerSpace(pointer);
    pointer.setPosition(new int[] {x,y});
    // Set the old playerSpace to holding null
    old.setPlayerSpace(null);
  } else {
    throw new Exception("That is not a valid space to move to");
  }
  // Player has now moved, it is the other players turn
  if(!isGameOver()) {
    changeTurn();
  }
  this.setChanged();
  this.notifyObservers("updateBoard");
}

/**
 * If the move is valid for the current player
 * @param currentPlayer - The player that is currently making a turn
 * @param x - The x move in question
 * @param y - The y move in question
 * @return - a boolean stating if the move is valid or not.
 */
private boolean validPlayerMove(Player currentPlayer, int x, int y) {
  boolean valid = false;
  // Get location of the currentPlayer
  int playerX = currentPlayer.getPosition()[0];
```

```
    int playerY = currentPlayer.getPosition()[1];

    // Get location of otherPlayer
    int otherPlayerX;
    int otherPlayerY;
    if(currentPlayer.equals(playerOne)) {
      otherPlayerX = playerTwo.getPosition()[0];
      otherPlayerY = playerTwo.getPosition()[1];
    } else {
      otherPlayerX = playerOne.getPosition()[0];
      otherPlayerY = playerOne.getPosition()[1];
    }
    // Create an arrayList of spaces which are valid for current player to move to.
    ArrayList<Space> validSpaces = new ArrayList<Space>();
    if(!this.board.get(playerX).get(playerY).getTop().getPlaced())
      validSpaces.add(this.board.get(playerX - 1).get(playerY));
    if(!this.board.get(playerX).get(playerY).getBottom().getPlaced())
      validSpaces.add(this.board.get(playerX + 1).get(playerY));
    if(!this.board.get(playerX).get(playerY).getLeft().getPlaced())
      validSpaces.add(this.board.get(playerX).get(playerY - 1));
    if(!this.board.get(playerX).get(playerY).getRight().getPlaced())
      validSpaces.add(this.board.get(playerX).get(playerY + 1));

    if(validSpaces.contains(this.board.get(otherPlayerX).get(otherPlayerY))) {
      // Cannot move to the other player, as the other player lives there
      // Remove that space from valid Spaces.
      validSpaces.remove(this.board.get(otherPlayerX).get(otherPlayerY));
      // The Spaces that may be valid
      if(!this.board.get(otherPlayerX).get(otherPlayerY).getTop().getPlaced())
        validSpaces.add(this.board.get(otherPlayerX - 1).get(otherPlayerY));
      if(!this.board.get(otherPlayerX).get(otherPlayerY).getBottom().getPlaced())
        validSpaces.add(this.board.get(otherPlayerX + 1).get(otherPlayerY));
      if(!this.board.get(otherPlayerX).get(otherPlayerY).getLeft().getPlaced())
        validSpaces.add(this.board.get(otherPlayerX).get(otherPlayerY - 1));
      if(!this.board.get(otherPlayerX).get(otherPlayerY).getRight().getPlaced())
        validSpaces.add(this.board.get(otherPlayerX).get(otherPlayerY + 1));

      // Remove the space the player is coming from
      validSpaces.remove(this.board.get(playerX).get(playerY));
    }
    // If the validSpaces algorithm finds the space player wants to go to
    // Then it is true
    if(validSpaces.contains(this.board.get(x).get(y))) {
      valid = true;
    }
    return valid;
  }

  /**
   * Tests if the game is over
   * @return - if the game is over boolean
   */
  private boolean isGameOver() {
    Set<Space> playerOneGoal = new HashSet<Space>();
    Set<Space> playerTwoGoal = new HashSet<Space>();
    for(int i = 0; i < this.boardSize; i++) {
      playerOneGoal.add(this.board.get(this.boardSize - 1).get(i));
      playerTwoGoal.add(this.board.get(0).get(i));
    }


  if(playerOneGoal.contains(this.board.get(this.getPlayerOne().getPosition()[0]).get(
  this.getPlayerOne().getPosition()[1])) ||

  playerTwoGoal.contains(this.board.get(this.getPlayerTwo().getPosition()[0]).get(thi
  s.getPlayerTwo().getPosition()[1]))) {
      this.gameOver = true;
      return true;
    }
    return false;
  }
```

In Depth Analysis:

Assuming N = boardSize

```
if(this.gameOver) {
  throw new Exception("Game is Over");
}
```

Calling this.gameOver() brings us to another method.

```
private boolean isGameOver() {
  Set<Space> playerOneGoal = new HashSet<Space>();
  Set<Space> playerTwoGoal = new HashSet<Space>();
  for(int i = 0; i < this.boardSize; i++) {
    playerOneGoal.add(this.board.get(this.boardSize - 1).get(i));
    playerTwoGoal.add(this.board.get(0).get(i));
  }


if(playerOneGoal.contains(this.board.get(this.getPlayerOne().getPosition()[0]).get(
this.getPlayerOne().getPosition()[1])) ||

playerTwoGoal.contains(this.board.get(this.getPlayerTwo().getPosition()[0]).get(thi
s.getPlayerTwo().getPosition()[1]))) {
    this.gameOver = true;
    return true;
  }
  return false;
}
```

In gameOver() we create two HashSets, (constant time), O(1) then we iterate through a for loop from 0 to boardSize thus this is linear O(N) and in the for loop we pull data points from a 2D ArrayList which is constant access O(1). Calling .contains on a set as in the if statement is amortized O(1) and calling .get is always constant on an ArrayList so the whole if statment is constant. **Thus gameOver() is overall O(N) since having a constant + a constant + a linear time complexity is**

linear. Assuming gameOver() returns false we continue...

```
Player pointer = null;
if(this.turn == 1) {
  pointer = playerOne;
} else if(this.turn == 2) {
  pointer = playerTwo;
}
```

This chunk of code above is all constant time so it is O(1).

```
if(validPlayerMove(pointer, x, y)) {
  // get the current space location of the player
  Space old =
this.board.get(pointer.getPosition()[0]).get(pointer.getPosition()[1]);
  // Set the intended space as the space holding the player
  this.board.get(x).get(y).setPlayerSpace(pointer);
  pointer.setPosition(new int[] {x,y});
  // Set the old playerSpace to holding null
  old.setPlayerSpace(null);
} else {
  throw new Exception("That is not a valid space to move to");
}
// Player has now moved, it is the other players turn
if(!isGameOver()) {
  changeTurn();
}
this.setChanged();
this.notifyObservers("updateBoard");
}
```

Besides the method call which starts the chunk of code above (& assuming the move is a valid one), we get data points from a 2D ArrayList which is constant, then we set a playerSpace which is constant then we set a position which is constant, and then set an old playerSpace to null which is constant. We call gameOver() again in an if statement which is O(N) again and then we setchanged and notifyObservers which is constant. **So this chunk of code's worst case is O(N)**

Now to analyze validPlayerMove(...)

```
private boolean validPlayerMove(Player currentPlayer, int x, int y) {
  boolean valid = false;
  // Get location of the currentPlayer
  int playerX = currentPlayer.getPosition()[0];
  int playerY = currentPlayer.getPosition()[1];

  // Get location of otherPlayer
  int otherPlayerX;
  int otherPlayerY;
  if(currentPlayer.equals(playerOne)) {
    otherPlayerX = playerTwo.getPosition()[0];
    otherPlayerY = playerTwo.getPosition()[1];
  } else {
    otherPlayerX = playerOne.getPosition()[0];
    otherPlayerY = playerOne.getPosition()[1];
  }
```

In the chunk of code above we set things equal to things and check if statments which amounts to constant time so O(1).

```
// Create an arrayList of spaces which are valid for current player to move to.
ArrayList<Space> validSpaces = new ArrayList<Space>();
if(!this.board.get(playerX).get(playerY).getTop().getPlaced())
  validSpaces.add(this.board.get(playerX - 1).get(playerY));
if(!this.board.get(playerX).get(playerY).getBottom().getPlaced())
  validSpaces.add(this.board.get(playerX + 1).get(playerY));
if(!this.board.get(playerX).get(playerY).getLeft().getPlaced())
  validSpaces.add(this.board.get(playerX).get(playerY - 1));
if(!this.board.get(playerX).get(playerY).getRight().getPlaced())
  validSpaces.add(this.board.get(playerX).get(playerY + 1));

if(validSpaces.contains(this.board.get(otherPlayerX).get(otherPlayerY))) {
  // Cannot move to the other player, as the other player lives there
  // Remove that space from valid Spaces.
  validSpaces.remove(this.board.get(otherPlayerX).get(otherPlayerY));
  // The Spaces that may be valid
  if(!this.board.get(otherPlayerX).get(otherPlayerY).getTop().getPlaced())
    validSpaces.add(this.board.get(otherPlayerX - 1).get(otherPlayerY));
  if(!this.board.get(otherPlayerX).get(otherPlayerY).getBottom().getPlaced())
    validSpaces.add(this.board.get(otherPlayerX + 1).get(otherPlayerY));
  if(!this.board.get(otherPlayerX).get(otherPlayerY).getLeft().getPlaced())
    validSpaces.add(this.board.get(otherPlayerX).get(otherPlayerY - 1));
  if(!this.board.get(otherPlayerX).get(otherPlayerY).getRight().getPlaced())
    validSpaces.add(this.board.get(otherPlayerX).get(otherPlayerY + 1));

  // Remove the space the player is coming from
  validSpaces.remove(this.board.get(playerX).get(playerY));
}
```

In the chunk of code above, we create an ArrayList (constant) that holds valid spaces surrounding the player that the player can move to (max of 4 spaces if no fences are placed/other player is not occupying one of those spaces) so this is all done in constant time since it is accessing an ArrayList and calling constant time methods. In the last outside if statment (if validSpaces.contains...) worst case is O(P) where P is the number of elements in the ArrayList at that time (and the max P can be is 4), and inside that if statment worst case of remove would be O(P) (but not nested) since contains would finish evaluating before remove would run. After .remove() we check whether the spaces are still valid, if so we add those spaces to the ArrayList (constant) and then we remove the space where the player is coming from since it isn't really a valid space which worst case is O(M) where M is the number of elements in the ArrayList (max of 7 no matter the size). **So we get O(P) where P is the number of elements in the ArrayList at the time (max of 4), and O(M) where M is the number of elements in the ArrayList after we add more to it (max of 7) so the time complexity is O(P) + O(M) which = O(P + M).**

Conclusion

So adding all the sections together we get O(N) + O(N) + O(P + M) + a ton of O(1) so we get O(2N + P + M) which has the potential to be quadratic if P + M add up to N which then we would have O(2N^2) or just O(N^2)

placeFence(...)

```
/**
 * Places a fence move
 * @param x - The x coord for the space that holds the fence
 * @param y - The y coord for the space that holds the fence
 * @param key - A string that says what fence one is trying to place
 * ^ key can be == "top" || "bottom" || "right" || "left"
 * @throws Exception - If the fence key is given and invalid
 */
public void placeFence(int x, int y, String key) throws Exception {
  if(this.gameOver) {
    throw new Exception("Game is Over");
  }
  // Temporary fence;
  Fence temp = null;
  if(key.equals("top")) {
    temp = this.board.get(x).get(y).getTop();
  } else if(key.equals("bottom")) {
    temp = this.board.get(x).get(y).getBottom();
  } else if(key.equals("left")) {
    temp = this.board.get(x).get(y).getLeft();
  } else if(key.equals("right")) {
    temp = this.board.get(x).get(y).getRight();
  } else {
    throw new Exception("There are no fences called: " + key);
  }

  if(temp.getPlaced()) {
    throw new Exception("There is already a fence there!");
  }

  // Place the fence so we can make sure its valid for both players
  temp.placeFence();
  // Test validity for both players
  // If either makes it an invalid move, then the fence is removed, and exception
thrown
  if(!(validFencePlacement(playerOne) && validFencePlacement(playerTwo))) {
    temp.removeFence();
    throw new Exception("You are not allowed to block a path to a player goal!");
  }
  // If the player sets a valid fence, then change the turn
  changeTurn();
  this.setChanged();
  this.notifyObservers("updateBoard");
}

/**
 * If there is a path from player to playerGoal
```

```java
 * @param player - The player in question
 * @return - boolean that says if the fence placement is valid
 */
private boolean validFencePlacement(Player player) {
  // By default the fence placement is not valid
  boolean valid = false;
  int x = player.getPosition()[0];
  int y = player.getPosition()[1];
  // Queue of searching
  Queue<Space> q = new LinkedList<Space>();
  // A set of all seen Spaces
  Set<Space> seen = new HashSet<Space>();
  // An ArrayList of spaces that we want to reach
  ArrayList<Space> goalSpaces = new ArrayList<Space>();

  // Add the player location space to the queue
  q.add(this.board.get(x).get(y));

  // Creates the seen spaces depending on the player
  // For player two its goal is to get to row 0
  int row = 0;
  // For player one its goal is to get to row 1
  if(player.equals(playerOne)) {
    row = this.boardSize - 1;
  }
  for(int i = 0; i < this.boardSize; i++) {
    goalSpaces.add(this.board.get(row).get(i));
  }

  while(!q.isEmpty()) {
    Space temp = q.remove();
    int tempX = temp.getPosition()[0];
    int tempY = temp.getPosition()[1];
    // If temp has been seen already, continue
    if(seen.contains(temp)) {
      continue;
    }
    // If goal spaces contains temp, then player can make it to its positions.
    if(goalSpaces.contains(temp)) {
      valid = true;
      break;
    }

    // If there is no top fence placed
    if(!temp.getTop().getPlaced()) {
      q.add(this.board.get(tempX - 1).get(tempY));
      //seen.add(this.board.get(tempX - 1).get(tempY));
    }
    // If there is no bottom fence placed
    if(!temp.getBottom().getPlaced()) {
      q.add(this.board.get(tempX + 1).get(tempY));
      //seen.add(this.board.get(tempX + 1).get(tempY));
    }
    // If there is no left fence placed
    if(!temp.getLeft().getPlaced()) {
      q.add(this.board.get(tempX).get(tempY - 1));
      //seen.add(this.board.get(tempX).get(tempY - 1));
    }
    // If there is no right fence placed
    if(!temp.getRight().getPlaced()) {
      q.add(this.board.get(tempX).get(tempY + 1));
      //seen.add(this.board.get(tempX).get(tempY + 1));
    }
    // Adds the space to the seen set
    seen.add(temp);
  }
  return valid;
}

/**
 * Tests if the game is over
 * @return - if the game is over boolean
 */
private boolean isGameOver() {
  Set<Space> playerOneGoal = new HashSet<Space>();
  Set<Space> playerTwoGoal = new HashSet<Space>();
  for(int i = 0; i < this.boardSize; i++) {
    playerOneGoal.add(this.board.get(this.boardSize - 1).get(i));
    playerTwoGoal.add(this.board.get(0).get(i));
  }


if(playerOneGoal.contains(this.board.get(this.getPlayerOne().getPosition()[0]).get(
this.getPlayerOne().getPosition()[1])) ||

playerTwoGoal.contains(this.board.get(this.getPlayerTwo().getPosition()[0]).get(thi
s.getPlayerTwo().getPosition()[1]))) {
    this.gameOver = true;
    return true;
  }
  return false;
}
```

In Depth Analysis:

Assuming N = boardSize

```
public void placeFence(int x, int y, String key) throws Exception {
  if(this.gameOver) {
    throw new Exception("Game is Over");
  }
  // Temporary fence;
  Fence temp = null;
  if(key.equals("top")) {
    temp = this.board.get(x).get(y).getTop();
  } else if(key.equals("bottom")) {
    temp = this.board.get(x).get(y).getBottom();
  } else if(key.equals("left")) {
    temp = this.board.get(x).get(y).getLeft();
  } else if(key.equals("right")) {
    temp = this.board.get(x).get(y).getRight();
  } else {
    throw new Exception("There are no fences called: " + key);
  }

  if(temp.getPlaced()) {
    throw new Exception("There is already a fence there!");
  }

  // Place the fence so we can make sure its valid for both players
  temp.placeFence();
  // Test validity for both players
  // If either makes it an invalid move, then the fence is removed, and exception
thrown
```

We have all constant time operations since we are just checking if variables are equal to other variables and if so access the 2D ArrayList. **So this all is O(1).**

```
if(!(validFencePlacement(playerOne) && validFencePlacement(playerTwo))) {
  temp.removeFence();
  throw new Exception("You are not allowed to block a path to a player goal!");
}
```

This next method we will analyze after this, but if we were to enter the if statment body which for worst case we don't, it would be constant anyways.

```
private boolean validFencePlacement(Player player) {
  // By default the fence placement is not valid
  boolean valid = false;
  int x = player.getPosition()[0];
  int y = player.getPosition()[1];
  // Queue of searching
  Queue<Space> q = new LinkedList<Space>();
  // A set of all seen Spaces
  Set<Space> seen = new HashSet<Space>();
  // An ArrayList of spaces that we want to reach
  ArrayList<Space> goalSpaces = new ArrayList<Space>();

  // Add the player location space to the queue
  q.add(this.board.get(x).get(y));

  // Creates the seen spaces depending on the player
  // For player two its goal is to get to row 0
  int row = 0;
  // For player one its goal is to get to row 1
  if(player.equals(playerOne)) {
    row = this.boardSize - 1;
  }
```

We create and assign variables in constant time and then we add an item to the Queue q which is worst case constant. We then check if player equals playerOne which is constant **so the above section overall is O(1).**

```
for(int i = 0; i < this.boardSize; i++) {
            goalSpaces.add(this.board.get(row).get(i));
        }
```

Here we go from 0 to boardSize so this is O(N).

```
while(!q.isEmpty()) {
  Space temp = q.remove();
  int tempX = temp.getPosition()[0];
  int tempY = temp.getPosition()[1];
  // If temp has been seen already, continue
  if(seen.contains(temp)) {
    continue;
  }
  // If goal spaces contains temp, then player can make it to its positions.
  if(goalSpaces.contains(temp)) {
    valid = true;
    break;
  }

  // If there is no top fence placed
  if(!temp.getTop().getPlaced()) {
    q.add(this.board.get(tempX - 1).get(tempY));
    //seen.add(this.board.get(tempX - 1).get(tempY));
  }
  // If there is no bottom fence placed
  if(!temp.getBottom().getPlaced()) {
    q.add(this.board.get(tempX + 1).get(tempY));
    //seen.add(this.board.get(tempX + 1).get(tempY));
  }
  // If there is no left fence placed
  if(!temp.getLeft().getPlaced()) {
    q.add(this.board.get(tempX).get(tempY - 1));
    //seen.add(this.board.get(tempX).get(tempY - 1));
  }
  // If there is no right fence placed
  if(!temp.getRight().getPlaced()) {
    q.add(this.board.get(tempX).get(tempY + 1));
    //seen.add(this.board.get(tempX).get(tempY + 1));
  }
  // Adds the space to the seen set
  seen.add(temp);
}
return valid;
}
```

q.isEmpty() is constant so we create and assign variables in constant time, then we call .contains on a set which is amoritzed constant. We then call .contains on an ArrayList which worst case is O(Q) where Q is the size of the ArrayList. Then if the rest of the if statements are hit we are just doing things in constant time since adding an item to a Queue is constant, and adding something to a set is constant. **So for this section worst case is a bunch of constants + O(Q) where Q is the size of goalSpaces which is max of 5.**

```
// If the player sets a valid fence, then change the turn
changeTurn();
this.setChanged();
this.notifyObservers("updateBoard");
```

Then the rest of the method is just constant, O(1).

Conclusion

So overall we will have O(1) + O(N) + O(Q) + O(1) + a bunch of useless constants (since it will never be constant worst case we can ignore them.) (where Q is max of 5). **So worst case could be O(N^2) when boardSize/Q both equal 5, but if the size of the board is bigger than the worst case is O(N + Q).**