Assignment # 2

Kristopher Beauchemin - 40109581

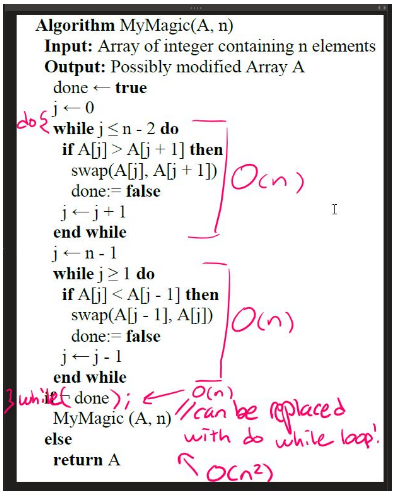
Concordia University

COMP 352 - Data Structures and Algorithms

October 20th, 2020

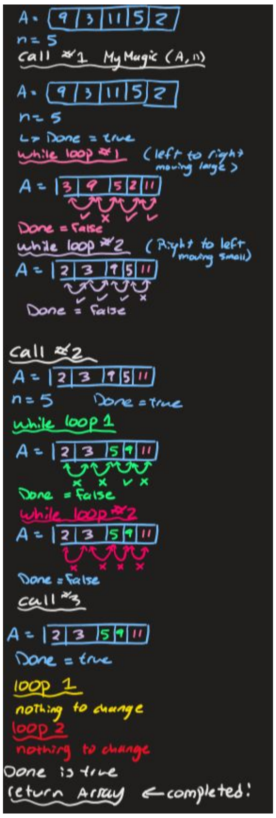
Question #1:

a) Time complexity follows O(n^2) and Ω(n). The O(n^2) comes from the tail recursion nesting the 2 while loop that goes through all n elements. This means that the time complexity would be O(n^2). The best time Ω(n), would be if the array is already sorted, in this case done would never be set to false and we would skip the recursion and only return the array. This would give us Ω(n).



```
Algorithm MyMagic(A, n)
  Input: Array of integer containing n elements
  Output: Possibly modified Array A
   done ← true
   j ← 0
do{ while j ≤ n - 2 do
      if A[j] > A[j + 1] then
        swap(A[j], A[j + 1])            O(n)
        done:= false
      j ← j + 1
   end while
   j ← n - 1
   while j ≥ 1 do
      if A[j] < A[j - 1] then
        swap(A[j - 1], A[j])            O(n)
        done:= false
      j ← j - 1
   end while
}while(done); ← O(n)
   MyMagic (A, n)  //can be replaced
  else             with do while loop!
   return A        ↖ O(n^2)
```

Figure 1: Steps for Question 1a



b) The resulting array will be (2, 3, 5, 9, 11).

c) The resulting array will be sorted from least to greatest

d) The runtime on this particular algorithm cannot be improved easily. It would require writing a completely new algorithm. Small changes can still be made. Like for example: Checking to see if done is true from the last while loop before entering the second while loop.

e) MyMagic does have tail recursion which should be converted to a do while loop to save on space.

Question #2:

i) Ω - f grows at least as fast as g (n*log n + n^3 > log(n))
ii) O - f grows no faster than g (log n^2 < log n)
iii) Ω - f grows at least as fast as g (n^3 > log n)
iv) Ω - f grows at least as fast as g (n^3/2 > log n^2)
v) Ω - f grows at least as fast as g (10^n > n^2)
vi) O - f grows no faster than g (n! < n^n)
vii) Ω - f grows at least as fast as g (log^2n > log n)
viii) Ω - f grows at least as fast as g (n > log n)
xi) Ω - f grows at least as fast as g (n^1/2 > log n)
x) O - f grows no faster than g (2^n < 3^n)
xi) O - f grows no faster than g (2^n < n^n)

Question #3:

MagicBoard Version 1 (Recursive):

Main Class in Pseudocode:

```
Main

   void main(args)
       Board board ← new Board()
       gameLoop(board)

   void gameLoop(board)
       board.displayBoard()
       board.getInput()
       if !board.checkWin() then
           gameLoop(board)
       else
           Print("YOU HAVE WON!!!!!")
```

Board Class in Pseudocode: (Game Implementation)

```
Board

   Board()
       d ← RandomNumber(0 to 25)
       size ← d * d
       checkerboard ← ArrayList<>(d * d)
       fillBoard(0)
       location ← 0
       moves ← 0
       Node val ← SolveNew(Node(null, 0, checkerboard.get(location)),
null, ArrayList<>())
       Print(val == null ? "No path" : "Has path")
       displayBoard()

   displayBoard(i, loc)
       if i == loc then
           Print("\t" + ">" + checkerboard.get(i ← i + 1))
        else
           Print("\t" + checkerboard.get(i ← i + 1))
       if i mod d == 0 then
           Print("\n")
       if i < size then
           displayBoard(i, loc)
```

```
getInput()
    Print("Which Direction would you like to go in?")
    input ← GetUserInput()
    switch input
        case "North" -> move(Direction.North)
        case "South" -> move(Direction.South)
        case "East" -> move(Direction.East)
        case "West" -> move(Direction.West)

fillBoard(i)
    checkerboard.add(1 + RandomNumber(d - 1))
    i ← i + 1
    if i < size then
        fillBoard(i)
    else
        targetLocation ← 1 + RandomNumber(size - 1)
        checkerboard.set(targetLocation, 0)
        return

checkWin()
    return checkerboard.get(location) == 0

validateMove(dir, oldLocation, newLocation)
    if newLocation < 0 OR newLocation > size - 1 then
        return false
    else if dir == Direction.East OR dir == Direction.West then
        one = (oldLocation + 1) / d
        two = (newLocation) / d
        if one != two then
            return false

    return true

move(dir)
    steps ← checkerboard.get(location)
    switch (dir)
        case North:
            steps ← steps * -d
            break
        case South:
            steps ← steps * d
            break
        case East:
            break
        case West:
            steps ← steps * -1
            break

    newLocation ← location + steps
    if !validateMove(dir, location, newLocation) then
        return
```

```
    location ← newLocation
    moves ← moves + 1
    checkWin()
```

Programming Part for Solution 1 in Pseudocode:

```
displayPath(n)
    if n == null if return
    displayBoard(0, n.Position)
    displayPath(n.ParentNode)

displayBoard()
    printBorder()
    Print("Current Moves: " + moves)
    displayBoard(0, location)
    printBorder()

SolveNew(n, lastMove, posList)
    value ← n.Value
    if value == 0 then
    solvable ← true
    return n

    posList.add(n.Position)

    canGoWest ← lastMove != Direction.East
 AND fakeMove(Direction.West, n.Position) > -1
 AND !posList.contains(fakeMove(Direction.West, n.Position))

    canGoEast ← lastMove != Direction.West
 AND fakeMove(Direction.East, n.Position) > -1
 AND !posList.contains(fakeMove(Direction.East, n.Position))

    canGoNorth ← lastMove != Direction.South
 AND fakeMove(Direction.North, n.Position) > -1
 AND !posList.contains(fakeMove(Direction.North, n.Position))

    canGoSouth = lastMove != Direction.North
 AND fakeMove(Direction.South, n.Position) > -1
 AND !posList.contains(fakeMove(Direction.South, n.Position))

    if canGoWest then
        westPos ← fakeMove(Direction.West, n.Position)
        westVal ← checkerboard.get(westPos)
        n.WestChild ← Node(n, westPos, westVal)
        node ← SolveNew(n.WestChild, Direction.West, ArrayList<>(posList))
        if node != null then return node
        else
```

```
            canGoWest = false

    if canGoEast then
        eastPos ⟵ fakeMove(Direction.East, n.Position)
        eastVal ⟵ checkerboard.get(eastPos)
        n.EastChild ⟵ new Node(n, eastPos, eastVal)
        node = SolveNew(n.EastChild, Direction.East, ArrayList<>(posList))
        if node != null then return node
        else
            canGoEast = false

    if canGoNorth then
        northPos ⟵ fakeMove(Direction.North, n.Position)
        northVal ⟵ checkerboard.get(northPos)
        n.NorthChild ⟵ Node(n, northPos, northVal)
        node ⟵ SolveNew(n.NorthChild, Direction.North, ArrayList<>(posList))
        if (node != null then return node
        else
            canGoNorth = false

    if canGoSouth then
        southPos ⟵ fakeMove(Direction.South, n.Position)
        southVal ⟵ checkerboard.get(southPos)
        n.SouthChild ⟵ Node(n, southPos, southVal)
        node ⟵ SolveNew(n.SouthChild, Direction.South, ArrayList<>(posList))
        if node != null then return node
        else
            canGoSouth ⟵ false
    if !canGoWest && !canGoEast && !canGoNorth && !canGoSouth then
        return null
    return null

fakeMove(dir, _location)
    steps ⟵ checkerboard.get(_location)
    switch (dir)
        case North:
            steps ⟵ steps * -d
            break
        case South:
            steps ⟵ steps * d
            break
        case East:
            break
        case West:
            steps ⟵ steps * -1
            break

    newLocation ⟵_location + steps
    if !validateMove(dir, _location, newLocation) then
        return -1
    checkWin()
```

```
    return newLocation
```

Node Class for Solution 1 in Pseudocode:

```
Node

   Node ParentNode ← null

   Node WestChild ← null
   Node EastChild ← null
   Node NorthChild ← null
   Node SouthChild ← null

   Value
   Position

   Node(parentNode, position, value)
       ParentNode ← parentNode
       Position ← position
       Value ← value
```

MagicBoard Version 2 (Iterative):

Solution Method:

```
solve()

   tree.push(startNode)

   value ← startNode.Value

   if value == 0 then
       solvable ← true
       return


   while(!solvable) then
       if tree.isEmpty() then
           return

       n = tree.peek()
       fakeMoveIndex ← fakeMove(Direction.South, n.Position)

       if  fakeMoveIndex > -1 && (n.lastDir == null
    || n.lastDir.getVal() < Direction.South.getVal())
```

```
    && ! n.Path.contains(fakeMoveIndex)) then
            southPos ← fakeMoveIndex
            southVal ← checkerboard.get(southPos)
            n.SouthChild ← Node(n, southPos, southVal,
 new ArrayList<>(n.Path))
            n.lastDir ← Direction.South
            tree.push(n.SouthChild)
            if southVal == 0 then
                solvable ← true
                break

            continue


        fakeMoveIndex ← fakeMove(Direction.West, n.Position)

        if  fakeMoveIndex > -1 && !n.Path.contains(fakeMoveIndex)
    && (n.lastDir == null || n.lastDir.getVal() < Direction.West.getVal()) then
            westPos ← fakeMoveIndex
            westVal ← checkerboard.get(westPos)
            n.WestChild ← Node(n, westPos, westVal, ArrayList<>(n.Path))
            n.lastDir ← Direction.West
            tree.push(n.WestChild)
            if westVal == 0 then
                solvable = true
                break

            continue


        fakeMoveIndex = fakeMove(Direction.North, n.Position);

        if (fakeMoveIndex > -1 && !n.Path.contains(fakeMoveIndex)
    && (n.lastDir == null || n.lastDir.getVal() < Direction.North.getVal()))
then
            northPos ← fakeMoveIndex
            northVal ← checkerboard.get(northPos)
            n.NorthChild = Node(n, northPos, northVal, ArrayList<>(n.Path))
            n.lastDir = Direction.North;
            tree.push(n.NorthChild);
            if northVal == 0 then
                solvable ← true
                break

            continue


        fakeMoveIndex ← fakeMove(Direction.East, n.Position);

        if fakeMoveIndex > -1 && !n.Path.contains(fakeMoveIndex) && (n.lastDir ==
null || n.lastDir.getVal() < Direction.East.getVal()) then
```

```
        eastPos ← fakeMoveIndex
        eastVal ← checkerboard.get(eastPos)
        n.EastChild = Node(n, eastPos, eastVal, ArrayList<>(n.Path))
        n.lastDir = Direction.East
        tree.push(n.EastChild)
        if eastVal == 0 then
            solvable ← true
            break

        continue

    if !tree.isEmpty() then
        tree.pop()
```

Node Class

```
Node

   Node ParentNode ← null

   Node WestChild ← null
   Node EastChild ← null
   Node NorthChild ← null
   Node SouthChild ← null

   ArrayList<Integer> Path;

   Value ← -1
   Position ← -1
   Board.Direction lastDir = null

   Node(parentNode, position, value, pathList)
       ParentNode = parentNode
       Position = position
       Value = value
       Path = pathList
       Path.add(Position)
```

a)

Time Complexity of Version 1: $O(n^2)$
Space Complexity of Version 1: $O(n)$ - from height of tree

Time Complexity of Version 2: $O(n^2)$
Space Complexity of Version 2: $O(n)$ - from height of tree

b) Tree recursion was the type of recursion that was used for question 1. Tree recursion, tree recursion was used to traverse the tree. Therefore the worst case would be $O(n)$ for this tree traversal. Tail-recursion for version 1 would not be possible because the tree is of degree 4 which means there must be at least 4 calls per call which would not be tail-recursion.

c) Stacks can be used to create a tree because recursion uses an internal stack to recurse over the tree. Stacks are also much easier to enter and remove from the top since they are $O(1)$ insertion and deletion from top. While queues could have been used with amortization and may achieve $O(1)$ for the most part, in our algorithm we delete from the front many times therefore using a queue would cost us $O(n)$. While a stack would achieve the same thing but in $O(1)$.

d) Please check “Q3A-20-tests.txt” and “Q3B-20-tests.txt” included in submission

e) I used an arraylist to store the last positions and checked to see if that position was already visited. If it was visited already then it would skip it. This got rid of loops in the maze that would result in a longer execution or even an infinite loop.