ENCARNACION, Stephen Mary S.

CS 21 22.1 – Lab 1

CS 21 Machine Problem 1: Peg Solitaire – Documentation

cs21project1B.asm – How it Works:

We first initialize the input board array. Since we are working with 7 rows and 7 columns, we can think of our array as a 2d array.

We then ask the user for input, 7 times. For each input, we have a string of 7 characters that we will iterate through, storing each character to our array, until it is filled. We also keep track of the 'E' or 'O' characters in our input, since this is where the final peg should be. We store the address of this character, and replace it with a "." if the character is an "E," or an "o" if the character is an "O."

Next up, we define our solver function. We have two base cases. That is when there is only 1 peg in the board, and it is located at the address of the E/O character. If this is the scenario, the function returns True. However, if we have only 1 peg in the board, and it is **NOT** located at the address of the E/O character, then the function returns False. If neither of the base cases were reached, then it is time to iterate.

We iterate through every element in the array. If the element is a peg, or the character "o," then we check if it can move up. If so then we can modify our board. If not then we check for another movement. If we moved up, we then try to call the solver function again, with our newly modified board. If it returns true, we return true, meaning that we have successfully solved the board. However, if it returns false, then we have not solved the board. We undo our upwards movement, then we try another movement.

We then check if we can move down. If so, then we modify the board then call the solver function. If not, then check for another possible movement. If the solver function returns true, then we return true. Else, we undo our downwards movement from the board, then check for another possible movement.

Once we have checked for every possible movement, and none of them were successful in solving the board, then it is time to increment the index. Now, if we have traversed through every element in the array, but have not solved the board, then we return false.

Python Equivalent:

Building the 2d array from user input.

```python
grid = []
moves = []
post = []

#build the grid
for i in range(7):
    inp = input()
    row = []
    for char in inp:
        row.append(char)

    grid.append(row)
```

Locating the E/O characters, then replacing them, and storing their index in a variable.

```python
#replace E/O
for i in range(7):
    for j in range(7):
        if grid[i][j] == "E":
            grid[i][j] = "."
            loc = i,j

        if grid[i][j] == "O":
            grid[i][j] = "o"
            loc = i,j

endrow, endcol = loc
```

Movement functions. If movement is possible, then modify the grid, and return true, else, return false.

```python
def moveUp(row, col):
    global grid
    if row >= 2:
        if grid[row][col] == 'o' and grid[row-1][col] == 'o' and grid[row-2][col] == '.':
            grid[row][col] = '.'
            grid[row-1][col] = '.'
            grid[row-2][col] = 'o'
            return True
        else:
            return False
    return False

def moveDown(row, col):
    global grid
    if row <= 4:
        if grid[row][col] == 'o' and grid[row+1][col] == 'o' and grid[row+2][col] == '.':
            grid[row][col] = '.'
            grid[row+1][col] = '.'
            grid[row+2][col] = 'o'
            return True
        else:
            return False
    return False
```

```
def moveLeft(row, col):
  global grid
  if col >= 2:
    if grid[row][col] == 'o' and grid[row][col-1] == 'o' and grid[row][col-2] == '.':
      grid[row][col] = '.'
      grid[row][col-1] = '.'
      grid[row][col-2] = 'o'
      return True
    else:
      return False
  return False

def moveRight(row, col):
  global grid
  if col <=4 :
    if grid[row][col] == 'o' and grid[row][col+1] == 'o' and grid[row][col+2] == '.':
      grid[row][col] = '.'
      grid[row][col+1] = '.'
      grid[row][col+2] = 'o'
      return True
    else:
      return False
  return False
```

The solver function. Included in the image are its base cases. We count for the number of pegs, and check if it is already at the target location.

```
def solver():
  global grid
  global endrow, endcol
  global moves
  global post
  count = 0


  for i in range(7):
    for j in range(7):
      if grid[i][j] == 'o':
        count += 1


#base case, only 1 peg left, and already at target destination
  if count == 1 and grid[endrow][endcol] == 'o':
    return True

#improperly placed
  if count == 1 and grid[endrow][endcol] != 'o':
    return False
```

Check for every movement of every peg. We call the movement functions. If it is true, then that means we have modified our board. We call the solver function again. If it returns True, then we return True. Else, we restore the board. Once we have iterated through every peg, without reaching a solution, we return false.

```
#check every movement of every peg
  for i in range(7):
    for j in range(7):
      if grid[i][j] == 'o':

        moved = moveUp(i,j)
        if moved == True:

          if solver():
            moves.append("up")
            post.append([i,j])
            return True
          else: #restoration
            grid[i][j] = 'o'
            grid[i-1][j] = 'o'
            grid[i-2][j] = '.'
```

```python
        moved = moveDown(i,j)
        if moved == True:

          if solver():
            moves.append("down")
            post.append([i,j])
            return True
          else: #restoration
            grid[i][j] = 'o'
            grid[i+1][j] = 'o'
            grid[i+2][j] = '.'

        moved = moveLeft(i,j)
        if moved == True:

          if solver():
            moves.append("left")
            post.append([i,j])
            return True
          else: #restoration
            grid[i][j] = 'o'
            grid[i][j-1] = 'o'
            grid[i][j-2] = '.'

        moved = moveRight(i,j)
        if moved == True:

          if solver():
            moves.append("right")
            post.append([i,j])
            return True
          else: #restoration
            grid[i][j] = 'o'
            grid[i][j+1] = 'o'
            grid[i][j+2] = '.'

  return False
```

The final part of the program. We check if there is a solution. The moves and post (position) arrays are used to ensure that our program works as intended. The printer function prints the final state of our board

```python
if solver():
  print("YES")
else:
  print("NO")


printer()
print(moves)
print(post)
```

```python
#grid printer
def printer():
  global grid
  for i in range(7):
    for j in range(7):
      print (grid[i][j], end="")
    print()
  print()
```

Sample Input/Output

```
xx...xx
xxo..xx
..o....
..oO...
.......
xx...xx
xx...xx

YES
xx...xx
xx...xx
.......
...o...
.......
xx...xx
xx...xx

['left', 'down', 'right']
[[3, 3], [1, 2], [3, 1]]
```

cs21project1b.asm – In-depth Explanation:

The main function is as follows. We get 7 lines of input, then jump to the solver function. If $v1 is equal to 1, then that means we have solved the board. If not, then we have not solved the board.

```
.text
main:
	la board, game_board #set the address of our array
	move $t0, board

	#7 lines of input, build the board
	jal input
	jal input
	jal input
	jal input
	jal input
	jal input
	jal input

	jal solver


	beq $v1, 1, yes
	printNO
	exit

	yes:
	printYES
	exit
```

Getting the Input (1/8)

In the data segment, we set line: .space 9. For each line of input later, we are expecting 8 characters, the 7 characters and a newline. Along with that, we initialize our array to contain 49 words, since we will be working with 7 rows and 7 columns.

```
.data
	line: .space 9
	game_board: .word 0:49 #initialize the game board, 7 rows with 7 columns
```

We also make use of the read_str macro, storing our input to the line.

```
.macro read_str # reads input
	li $v0, 8
	la $a0, line
	li $a1, 9
	syscall
.end_macro
```

```
input:
	addi $sp, $sp, -8
	sw $s0, 0($sp)
	sw $s1, 4($sp)

	read_str
	loop_store:
	beq $s0, 7, end_input
	#$s1 contains a character
	lb $s1, line($s0)

	beq $s1, 69, found_E
	beq $s1, 79, found_O

	return:
	sw $s1, ($t0)
	addi $t0, $t0, 4
	addi $s0, $s0, 1
	j loop_store

	end_input:
	lw $s1, 4($sp)
	lw $s0, 0($sp)
	addi $sp, $sp, 8
	jr $ra

	found_E: #we store the address of E, and change 'E' to '.'
	la end_index, ($t0)
	li $s1, 46 #ASCII of "."
	j return

	found_O: #we store the address of O, and change 'O' to 'o'
	la end_index, ($t0)
	li $s1, 111 #ASCII of "o"
	j return
```

Now when the function input is called, we first store the necessary registers. $s0 and $s1 are both equal to 0 at the point this function is called. We call the read_str macro. $t0 contains the address of our array.

What happens is that we iterate through our string stored in line, and store each character to our array. Each time the input function is called, we store 7 elements to our array. After 7 iterations, the function is finished. We restore the necessary registers.

When our character is equal to “E” or “O,” we keep track of the address of this character in the array, then change that character to a “.” or “o,” respectively.

Remember from the main function that we have called this function 7 times. After that, we have successfully built our board. We also have the address/index of the E/O character, and that is where the final peg should end up.

Solver Function, Counter Function (2/8)

```
solver:
	addi $sp, $sp, -20
	sw $a1, 0($sp) #number of pegs
	sw $s0, 4($sp) #i
	sw $s1, 8($sp) #j
	sw $s2, 12($sp) #board[i][j]
	sw $ra, 16($sp)

	jal counter #we have the number of pegs now, stored in $a1
	beq $a1, 1, base_case #if we have only 1 peg, check if it's in the end_index
```

The solver function begins with saving the necessary registers, which is followed by the calling of the counter function.

```
#count number of pegs
counter:
	addi $sp, $sp, -20
	sw $s0, 0($sp) #i
	sw $s1, 4($sp) #j
	sw $s2, 8($sp) #board[i][j]
	sw $s3, 12($sp) #number of pegs
	sw $ra, 16($sp)

	li $s3, 0 #count of pegs
	li $s0, 0 #i

	for1:
	li $s1, 0 #j

	for2:
	beq $s1, 7, check_outer

	#check board[i][j]
	mul $s2, $s0, 7
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add $s2, $s2, board
	lw $s2, ($s2)

	beq $s2, 111, increment_count
	j increment_index

	increment_count:
	addi $s3, $s3, 1

	increment_index:
	addi $s1, $s1, 1
	j for2

	check_outer:
	addi $s0, $s0, 1
	beq $s0, 7, counting_end
	j for1

	counting_end:
	move $a1, $s3 #keep track of count of pegs
	lw $s0, 0($sp)
	lw $s1, 4($sp)
	lw $s2, 8($sp)
	lw $s3, 12($sp)
	lw $ra, 16($sp)
	addi $sp, $sp, 20
	jr $ra
```

The counter function is like a nested for loop.

We try to count the total number of pegs in our board. To access the board[i][j] in terms of memory, we apply the formula base address of board + (i * 7 + j) * 4.

Once we have accessed a certain character, we try to check if it is a peg, or the character “o”. If so, then we increment $s3 by 1, then increment the index. If not, then we simply increment the index.

At the end of this function, once all elements in the array have been checked, we move to $a1 the contents of $s3. $a1 now contains the total number of pegs in the board. We then restore the necessary registers.

Solver Function, Base Cases (3/8)

```
solver:
	addi $sp, $sp, -20
	sw $a1, 0($sp) #number of pegs
	sw $s0, 4($sp) #i
	sw $s1, 8($sp) #j
	sw $s2, 12($sp) #board[i][j]
	sw $ra, 16($sp)

	jal counter #we have the number of pegs now, stored in $a1
	beq $a1, 1, base_case #if we have only 1 peg, check if it's in the end_index
```

After the counter function has been called, $a1 now contains the total number of pegs in the board. If it is equal to 1, then we proceed to the base_case label.

```
base_case: #only 1 peg left
	lw $t0, 0(end_index)
	beq $t0, 111, base_case_met
	j solver_end #not met

base_case_met:
	li $v1, 1
	j solver_end

solver_end:
	lw $a1, 0($sp) #number of pegs
	lw $s0, 4($sp) #i
	lw $s1, 8($sp) #j
	lw $s2, 12($sp) #board[i][j]
	lw $ra, 16($sp)
	addi $sp, $sp, 20
	jr $ra
```

We then access the element in the array where the character E/O was supposed to be. If the element we accessed is equal to 111, or the character 'o' which is a peg, then we are successful. That means we only have 1 peg in the board, and that peg is located at the target destination. We then branch out to the base_case_met label, setting $v1 to 1, and proceed to the end of the function.

However, if the element we accessed is **NOT** equal to 111, then we are unsuccessful. That means we only have 1 peg in the board, and it is not located at the target destination. We proceed to the end of the function in this case. $v0 remains 0.

Solver Function, Check Movements (4/8)

```
check_movements:
        li $s0, 0 #i
        forr1:
        li $s1, 0 #j
        forr2:
        beq $s1, 7, check_outerr

        #check board[i][j]
        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4

        add $s2, $s2, board
        lw $s2, ($s2)

        beq $s2, 111, peg_detected #the character 'o' is found at board[i][j]
        j increment_indexx #if current character is not a peg, increment index

        peg_detected:
```

```

        increment_indexx:
        addi $s1, $s1, 1
        j forr2

        check_outerr:
        addi $s0, $s0, 1
        beq $s0, 7, solver_end
        j forr1
```

Now, when $a1, the total number of pegs in the board is not equal to 1, we then proceed to the check_movements label. Again this is a nested for loop, where we try to visit each element in the board array. Now, if board[i][j] is equal to 111, or the character “o” which is a peg, we branch out to the peg_detected label. Else, we simply increment the index. Once we have reached the end of the array without a successful movement, we go to the end of the function.

Solver Function, Check Movements, Peg Detected (5/8)

```
        peg_detected:
        check_up:
        jal moveUp
        beq $a2, 1, up #check if we can move up
        j check_down #if we cant go up, check if we can go down

        up:
        li $a2, 0
        jal tryMoveUp
        beq $v1, 1, solver_end


        check_down:
        jal moveDown
        beq $a2, 1, down #check if we can move up
        j check_left #if we cant go down, check left

        down:
        li $a2, 0
        jal tryMoveDown
        beq $v1, 1, solver_end

```

```
        check_left:
        jal moveLeft
        beq $a2, 1, left #check if we can move left
        j check_right #if we cant go left, check_right

        left:
        li $a2, 0
        jal tryMoveLeft
        beq $v1, 1, solver_end

        check_right:
        jal moveRight
        beq $a2, 1, right #check if we can move right
        j increment_indexx #no more possible movements, increment index

        right:
        li $a2, 0
        jal tryMoveRight
        beq $v1, 1, solver_end

        increment_indexx:
        addi $s1, $s1, 1
        j forr2

```

Once a peg is detected, we try to move our peg. Firstly, we try to check if we can move up by calling the moveUp function. If $a2 becomes equal to 1, then that means we can indeed move up, so we branch to the up label. If not, then that means we cannot move up. We jump to the check_down label.

In the up label, we reset $a2 to 0, then call the tryMoveUp function. Now, if $v1 becomes 1, then we proceed to the end of the solver function, if not, then we go to the check_down label.

In the check_down label, we try to check if we can move down by calling the moveDown function. If $a2 becomes equal to 1, then that means we can indeed move down, so we branch to the down label. If not, then that means we cannot go down. We jump to the check_left label.

In the down label, we reset $a2 to 0, then call the tryMoveDown function. Now, if $v1 becomes 1, then we proceed to the end of the solver function, if not, then we go to the check_left label.

In the check_left label, we try to check if we can move left by calling the moveLeft function. If $a2 becomes equal to 1, then that means we can indeed move left, so we branch to the left label. If not, then that means we cannot go left. We jump to the check_right label.

In the left label, we reset $a2 to 0, then call the tryMoveLeft function. Now, if $v1 becomes 1, then we proceed to the end of the solver function, if not, then we go to the check_right label.

In the check_right label, we try to check if we can move right by calling the moveRight function. If $a2 becomes equal to 1, then that means we can indeed move right, so we branch to the right label. If not, then that means we cannot go right. Since we have tried for the 4 possible movements, and none of them were successful, we jump to the increment_indexx label.

In the right label, we reset $a2 to 0, then call the tryMoveRight function. Now, if $v1 becomes 1, then we proceed to the end of the solver function, if not, then we go to the increment_indexx label.

Check if Movement is Possible – moveDirection Functions (6/8)

```
moveUp:
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        blt $s0, 2, moveUp_end

        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4

        add $s2, $s2, board
        addi $s2, $s2, -28
        move $t1, $s2
        lw $s2, ($s2) #board[i-1][j]

        beq $s2, 111, upCondition1 #if #board[i-1][j] = 'o'

        moveUp_end:
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra

        upCondition1:
        addi $t0, $0, 1
        addi $t1, $t1, -28
        lw $t1, ($t1) #board[i-2][j]
        beq $t1, 46, upCondition2 #if #board[i-2][j] = '.'
        j moveUp_end

        upCondition2:
        li $a2, 1
        j moveUp_end
```

moveUp

Recall that $s0 and $s1 contains the i and j, which we can use to access certain elements of the board array. For the moveUp function, we first check if i is less than 2. If so, then we proceed to the end of the moveUp function. Else, we proceed. Since board[i][j] is a peg, we check if board[i-1][j] is also a peg. If it is not a peg, then we proceed to the end of the function. If it is indeed a peg, we then branch to upCondition1 where we check if board[i-2][j] is the character ".". If not, then we proceed to the end of the function. If so, then we go to the upCondition2 label, where we set $a2 to 1, then proceed to the end of the function. $a2 having a value of 1 means that movement is possible.

The following python code can be seen as something analogous

```python
if row >= 2:
  if grid[row][col] == 'o' and grid[row-1][col] == 'o' and grid[row-2][col] == '.':
    grid[row][col] = '.'
    grid[row-1][col] = '.'
    grid[row-2][col] = 'o'
    return True
```

moveDown

```
moveDown:
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        bgt $s0, 4, moveDown_end

        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4

        add $s2, $s2, board
        addi $s2, $s2, 28
        move $t1, $s2
        lw $s2, ($s2) #board[i+1][j]

        beq $s2, 111, downCondition1 #if #board[i+1][j] = 'o'

        moveDown_end:
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra

        downCondition1:
        addi $t0, $0, 1
        addi $t1, $t1, 28
        lw $t1, ($t1) #board[i+2][j]
        beq $t1, 46, downCondition2 #if #board[i+2][j] = '.'
        j moveDown_end

        downCondition2:
        li $a2, 1
        j moveDown_end
```

Recall that $s0 and $s1 contains the i and j, which we can use to access certain elements of the board array. For the moveDown function, we first check if i is greater than 4. If so, then we proceed to the end of the moveDown function. Else, we proceed. Since board[i][j] is a peg, we check if board[i+1][j] is also a peg. If it is not a peg, then we proceed to the end of the function. If it is indeed a peg, we then branch to downCondition1, where we check if board[i+2][j] is the character ".". If not, then we proceed to the end of the function. If so, then we go to the leftCondition2 label, where we set $a2 to 1, then proceed to the end of the function. $a2 having a value of 1 means that movement is possible.

The following python code can be seen as something analogous

```python
def moveDown(row, col):
  global grid
  if row <= 4:
    if grid[row][col] == 'o' and grid[row+1][col] == 'o' and grid[row+2][col] == '.':
      grid[row][col] = '.'
      grid[row+1][col] = '.'
      grid[row+2][col] = 'o'
      return True
```

moveLeft

```
moveLeft:
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        blt $s1, 2, moveLeft_end

        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4

        add $s2, $s2, board
        addi $s2, $s2, -4
        move $t1, $s2
        lw $s2, ($s2) #board[i][j-1]

        beq $s2, 111, leftCondition1 #if #board[i][j-1] = 'o'

        moveLeft_end:
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra

        leftCondition1:
        addi $t0, $0, 1
        addi $t1, $t1, -4
        lw $t1, ($t1) #board[i][j-2]
        beq $t1, 46, leftCondition2 #if #board[i][j-2] = '.'
        j moveLeft_end

        leftCondition2:
        li $a2, 1
        j moveLeft_end
```

Recall that $s0 and $s1 contains the i and j, which we can use to access certain elements of the board array. For the moveLeft function, we first check if j is less than 2. If so, then we proceed to the end of the moveLeft function. Else, we proceed. Since board[i][j] is a peg, we check if board[i][j-1] is also a peg. If it is not a peg, then we proceed to the end of the function. If it is indeed a peg, we then branch to leftCondition1 where we check if board[i][j-2] is the character ".". If not, then we proceed to the end of the function. If so, then we go to the leftCondition2 label, where we set $a2 to 1, then proceed to the end of the function. $a2 having a value of 1 means that movement is possible.

The following python code can be seen as something analogous

```python
def moveLeft(row, col):
  global grid
  if col >= 2:
    if grid[row][col] == 'o' and grid[row][col-1] == 'o' and grid[row][col-2] == '.':
      grid[row][col] = '.'
      grid[row][col-1] = '.'
      grid[row][col-2] = 'o'
      return True
```

moveRight

```
moveRight:
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        bgt $s1, 4, moveRight_end

        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4

        add $s2, $s2, board
        addi $s2, $s2, 4
        move $t1, $s2
        lw $s2, ($s2) #board[i][j+1]

        beq $s2, 111, rightCondition1 #if #board[i][j+1] = 'o'

        moveRight_end:
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra

        rightCondition1:
        addi $t0, $0, 1
        addi $t1, $t1, 4
        lw $t1, ($t1) #board[i][j+2]
        beq $t1, 46, rightCondition2 #if #board[i][j+2] = '.'
        j moveRight_end

        rightCondition2:
        li $a2, 1
        j moveRight_end
```

Recall that $s0 and $s1 contains the i and j, which we can use to access certain elements of the board array. For the moveRight function, we first check if j is greater than 4. If so, then we proceed to the end of the moveRight function. Else, we proceed. Since board[i][j] is a peg, we check if board[i][j+1] is also a peg. If it is not a peg, then we proceed to the end of the function. If it is indeed a peg, we then branch to rightCondition1 where we check if board[i][j+2] is the character ".". If not, then we proceed to the end of the function. If so, then we go to the rightCondition2 label, where we set $a2 to 1, then proceed to the end of the function. $a2 having a value of 1 means that movement is possible.

The following python code can be seen as something analogous

```python
def moveRight(row, col):
  global grid
  if col <=4 :
    if grid[row][col] == 'o' and grid[row][col+1] == 'o' and grid[row][col+2] == '.':
      grid[row][col] = '.'
      grid[row][col+1] = '.'
      grid[row][col+2] = 'o'
      return True
```

Try Moving a Peg + Recursion (7/8)

Now that we have called moveUp / moveDown / moveLeft / moveRight, we can now know if movement is indeed is possible. If $a2 is equal to 1, then we try moving the the peg in board[i][j] in a certain direction. Let’s now call tryMoveDirection.

```
peg_detected:
check_up:
jal moveUp
beq $a2, 1, up #check if we can move up
j check_down #if we cant go up, check if we can go down

up:
li $a2, 0
jal tryMoveUp
beq $v1, 1, solver_end


check_down:
jal moveDown
beq $a2, 1, down #check if we can move up
j check_left #if we cant go down, check left

down:
li $a2, 0
jal tryMoveDown
beq $v1, 1, solver_end

check_left:
jal moveLeft
beq $a2, 1, left #check if we can move left
j check_right #if we cant go left, check_right

left:
li $a2, 0
jal tryMoveLeft
beq $v1, 1, solver_end

check_right:
jal moveRight
beq $a2, 1, right #check if we can move right
j increment_indexx #no more possible movements, increment index

right:
li $a2, 0
jal tryMoveRight
beq $v1, 1, solver_end

increment_indexx:
addi $s1, $s1, 1
j forr2
```

tryMoveUp:

```
tryMoveUp:
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4
        add $s2, $s2, board

        li $t1, 46
        sw $t1, ($s2) #grid[i][j] = '.'

        addi $s2, $s2, -28
        sw $t1, ($s2) #grid[i-1][j] = '.'

        addi $s2, $s2, -28
        li $t1, 111
        sw $t1, ($s2) #grid[i-2][j] = 'o'

        jal solver #if solver(), return True
        beq $v1, 1, movedUp

        #else we undo the movement
        li $t1, 46
        sw $t1, ($s2) #grid[i-2][j] = '.'

        li $t1, 111
        addi $s2, $s2, 28
        sw $t1, ($s2) #grid[i-1][j] = 'o'

        addi $s2, $s2, 28
        sw $t1, ($s2) #grid[i][j] = 'o'

        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra

        movedUp:
        #printmsg1
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

For this function to be called, movement upwards must first be possible. Now, since we can indeed move upwards, we modify our board. We first store the necessary registers.

We set grid[i][j] = '.', grid[i-1][j] = '.', and grid[i-2][j] = 'o'.

With our modified board, we call the solver function again. This is where the recursion occurs. Now after that call finishes executing, we then check for the value of $v1. If it is is equal to 1, then we simply go to the label movedUp, where we restore the necessary registers and go back to $ra. If not, then we undo our modifications to the board, then restore the necessary registers and return to $ra.

There is a comment #printmsg1 in the movedUp label, which allowed me to trace the movement of the pegs and verify that my program works as intended.

tryMoveDown

```
tryMoveDown:
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4
        add $s2, $s2, board

        li $t1, 46
        sw $t1, ($s2) #grid[i][j] = '.'

        addi $s2, $s2, 28
        sw $t1, ($s2) #grid[i+1][j] = '.'

        addi $s2, $s2, 28
        li $t1, 111
        sw $t1, ($s2) #grid[i+2][j] = 'o'

        jal solver #if solver(), return True
        beq $v1, 1, movedDown

        #else we undo the movement
        li $t1, 46
        sw $t1, ($s2) #grid[i+2][j] = '.'

        li $t1, 111
        addi $s2, $s2, -28
        sw $t1, ($s2) #grid[i+1][j] = 'o'

        addi $s2, $s2, -28
        sw $t1, ($s2) #grid[i][j] = 'o'

        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra

        movedDown:
        #printmsg2
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

For this function to be called, movement downwards must first be possible. Now, since we can indeed move downwards, we modify our board. We first store the necessary registers.

We set grid[i][j] = '.', grid[i+1][j] = '.', and grid[i+2][j] = 'o'.

With our modified board, we call the solver function again. This is where the recursion occurs. Now after that call finishes executing, we then check for the value of $v1. If it is is equal to 1, then we simply go to the label movedDown, where we restore the necessary registers and go back to $ra. If not, then we undo our modifications to the board, then restore the necessary registers and return to $ra.

tryMoveLeft

```
tryMoveLeft:
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4
        add $s2, $s2, board

        li $t1, 46
        sw $t1, ($s2) #grid[i][j] = '.'

        addi $s2, $s2, -4
        sw $t1, ($s2) #grid[i][j-1] = '.'

        addi $s2, $s2, -4
        li $t1, 111
        sw $t1, ($s2) #grid[i][j-2] = 'o'

        jal solver #if solver(), return True
        beq $v1, 1, movedLeft

        #else we undo the movement
        li $t1, 46
        sw $t1, ($s2) #grid[i][j-2] = '.'

        li $t1, 111
        addi $s2, $s2, 4
        sw $t1, ($s2) #grid[i][j-1] = 'o'

        addi $s2, $s2, 4
        sw $t1, ($s2) #grid[i][j] = 'o'

        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra

        movedLeft:
        #printmsg3
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

For this function to be called, movement leftwards must first be possible. Now, since we can indeed move leftwards, we modify our board. We first store the necessary registers.

We set grid[i][j] = '.', grid[i1][j-1] = '.', and grid[i][j-2] = 'o'.

With our modified board, we call the solver function again. This is where the recursion occurs. Now after that call finishes executing, we then check for the value of $v1. If it is is equal to 1, then we simply go to the label movedLeft, where we restore the necessary registers and go back to $ra. If not, then we undo our modifications to the board, then restore the necessary registers and return to $ra.

tryMoveRight

```
tryMoveRight:
	addi $sp, $sp, -4
	sw $ra, 0($sp)

	mul $s2, $s0, 7
	add $s2, $s2, $s1
	mul $s2, $s2, 4
	add $s2, $s2, board

	li $t1, 46
	sw $t1, ($s2) #grid[i][j] = '.'

	addi $s2, $s2, 4
	sw $t1, ($s2) #grid[i][j+1] = '.'

	addi $s2, $s2, 4
	li $t1, 111
	sw $t1, ($s2) #grid[i][j+2] = 'o'

	jal solver #if solver(), return True
	beq $v1, 1, movedRight

	#else we undo the movement
	li $t1, 46
	sw $t1, ($s2) #grid[i][j+2] = '.'

	li $t1, 111
	addi $s2, $s2, -4
	sw $t1, ($s2) #grid[i][j+1] = 'o'

	addi $s2, $s2, -4
	sw $t1, ($s2) #grid[i][j] = 'o'

	lw $ra, 0($sp)
	addi $sp, $sp, 4
	jr $ra

	movedRight:
	#printmsg4
	lw $ra, 0($sp)
	addi $sp, $sp, 4
	jr $ra
```

For this function to be called, movement rightwards must first be possible. Now, since we can indeed move rightwards, we modify our board. We first store the necessary registers.

We set grid[i][j] = '.', grid[i1][j+1] = '.', and grid[i][j+2] = 'o'.

With our modified board, we call the solver function again. This is where the recursion occurs. Now after that call finishes executing, we then check for the value of $v1. If it is is equal to 1, then we simply go to the label movedRight, where we restore the necessary registers and go back to $ra. If not, then we undo our modifications to the board, then restore the necessary registers and return to $ra.

Conclusion (8/8)

To reiterate, we iterate through element in the array. We check if that element is a peg, and if not, we increment our index. If it is indeed a peg, we go the the peg_detected label.

In the peg_detected label, we check if a movement is possible, then we try modifying our board, then calling the solver function again through the tryMoveDirection function. After the tryMoveDirection is called, we try to check if $v1 equals 1. If so, then we have solved the board, and we immediately go to the end of the solver function. If not, then we go check for the next possible movement. If all possible movements were checked, and none of them were able to solve the board, then we increment the index.

```
check_movements:
        li $s0, 0 #i
        forr1:
        li $s1, 0 #j
        forr2:
        beq $s1, 7, check_outerr

        #check board[i][j]
        mul $s2, $s0, 7
        add $s2, $s2, $s1
        mul $s2, $s2, 4

        add $s2, $s2, board
        lw $s2, ($s2)

        beq $s2, 111, peg_detected #the character 'o' is found at board[i][j]
        j increment_indexx #if current character is not a peg, increment index

        peg_detected:
```

```

        increment_indexx:
        addi $s1, $s1, 1
        j forr2

        check_outerr:
        addi $s0, $s0, 1
        beq $s0, 7, solver_end
        j forr1
```

```
        peg_detected:
        check_up:
        jal moveUp
        beq $a2, 1, up #check if we can move up
        j check_down #if we cant go up, check if we can go down

        up:
        li $a2, 0
        jal tryMoveUp
        beq $v1, 1, solver_end


        check_down:
        jal moveDown
        beq $a2, 1, down #check if we can move up
        j check_left #if we cant go down, check left

        down:
        li $a2, 0
        jal tryMoveDown
        beq $v1, 1, solver_end

```

```
        check_left:
        jal moveLeft
        beq $a2, 1, left #check if we can move left
        j check_right #if we cant go left, check_right

        left:
        li $a2, 0
        jal tryMoveLeft
        beq $v1, 1, solver_end

        check_right:
        jal moveRight
        beq $a2, 1, right #check if we can move right
        j increment_indexx #no more possible movements, increment index

        right:
        li $a2, 0
        jal tryMoveRight
        beq $v1, 1, solver_end

        increment_indexx:
        addi $s1, $s1, 1
        j forr2

```

When $s0 reaches 7, then we branch to the end of the solver function.

```
solver_end:
        lw $a1, 0($sp) #number of pegs
        lw $s0, 4($sp) #i
        lw $s1, 8($sp) #j
        lw $s2, 12($sp) #board[i][j]
        lw $ra, 16($sp)
        addi $sp, $sp, 20
        jr $ra
```

The following python code is analogous to the process I have described.

```python
#check every movement of every peg
    for i in range(7):
        for j in range(7):
            if grid[i][j] == 'o':

                moved = moveUp(i,j)
                if moved == True:

                    if solver():
                        moves.append("up")
                        post.append([i,j])
                        return True
                    else: #restoration
                        grid[i][j] = 'o'
                        grid[i-1][j] = 'o'
                        grid[i-2][j] = '.'

                moved = moveDown(i,j)
                if moved == True:

                    if solver():
                        moves.append("down")
                        post.append([i,j])
                        return True
                    else: #restoration
                        grid[i][j] = 'o'
                        grid[i+1][j] = 'o'
                        grid[i+2][j] = '.'

                moved = moveLeft(i,j)
                if moved == True:

                    if solver():
                        moves.append("left")
                        post.append([i,j])
                        return True
                    else: #restoration
                        grid[i][j] = 'o'
                        grid[i][j-1] = 'o'
                        grid[i][j-2] = '.'

                moved = moveRight(i,j)
                if moved == True:

                    if solver():
                        moves.append("right")
                        post.append([i,j])
                        return True
                    else: #restoration
                        grid[i][j] = 'o'
                        grid[i][j+1] = 'o'
                        grid[i][j+2] = '.'

    return False
```

Upon calling the solver function, $v1 will either be 0 or 1. It will be 0 if the board is unsolvable, or it will be 1 if the board has been solved. If $v1 is equal to 0, then we print the string “NO”, else, we print the string “YES.”

```
main:
        la board, game_board #set the address of our array
        move $t0, board

        #7 lines of input, build the board
        jal input
        jal input
        jal input
        jal input
        jal input
        jal input
        jal input

        jal solver


        beq $v1, 1, yes
        printNO
        exit

        yes:
        printYES
        exit
```

```
.macro printYES
        li $v0, 4
        la $a0, msg5
        syscall
.end_macro

.macro printNO
        li $v0, 4
        la $a0, msg6
        syscall
.end_macro
```

```
.data
        line: .space 9
        game_board: .word 0:49 #initialize the game board, 7 rows with 7 columns
        msg1: .asciiz "up "
        msg2: .asciiz "down "
        msg3: .asciiz "left "
        msg4: .asciiz "right "
        msg5: .asciiz "YES"
        msg6: .asciiz "NO"
```

Test Cases

Input:

xx...xx

xxo..xx

..o....

..oO...

....

xx...xx

xx...xx

Output:

YES

Input:

xx...xx

xxo..xx

..o....

E.oo...

....

xx...xx

xx...xx

Output:

YES

Input:

xx...xx

xxo..xx

..O....

..oo...

....

xx...xx

xx...xx

Output:

NO

Input:

xx...xx

xxo..xx

..o....

..oo...

....

xx.E.xx

xx...xx

Output:

NO

Input:

xx.o.xx

xxooOxx

..oo...

...oo..

....

xx...xx

xx...xx

Output:

YES

Input:

...xE.o

...oo..

...oo..

....

....

....

....

Output:

YES

Input:

....

....

....

...E...

....

....

....

Output:

NO

Input:

....

....

....

....

....

....

.....O.

Output:

YES

Input:

...xx.x

...xE.o

...oo..

x..oo..

..xxx..

...x...

o..x...

Output:

NO

Input:

xx.o.xx

xxo.oxx

..ooE..

..o.o..

...o...

xx...xx

xx...xx

Output:

YES