

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.

```
1  grid = []
2  moves = []
3  post = []
4
5  #build the grid
6  for i in range(7):
7      inp = input()
8      row = []
9      for char in inp:
10         row.append(char)
11
12     grid.append(row)
```

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

```
16  #replace E/O
17  for i in range(7):
18      for j in range(7):
19          if grid[i][j] == "E":
20              grid[i][j] = "."
21              loc = i,j
22
23          if grid[i][j] == "O":
24              grid[i][j] = "o"
25              loc = i,j
26
27  endrow, endcol = loc
```

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

```
39  def moveUp(row, col):
40      global grid
41      if row >= 2:
42          if grid[row][col] == 'o' and grid[row-1][col] == 'o' and grid[row-2][col] == '.':
43              grid[row][col] = '.'
44              grid[row-1][col] = '.'
45              grid[row-2][col] = 'o'
46              return True
47          else:
48              return False
49      return False
50
51  def moveDown(row, col):
52      global grid
53      if row <= 4:
54          if grid[row][col] == 'o' and grid[row+1][col] == 'o' and grid[row+2][col] == '.':
55              grid[row][col] = '.'
56              grid[row+1][col] = '.'
57              grid[row+2][col] = 'o'
58              return True
59          else:
60              return False
61      return False
```

```

63 def moveLeft(row, col):
64     global grid
65     if col >= 2:
66         if grid[row][col] == 'o' and grid[row][col-1] == 'o' and grid[row][col-2] == '.':
67             grid[row][col] = '.'
68             grid[row][col-1] = '.'
69             grid[row][col-2] = 'o'
70             return True
71     else:
72         return False
73     return False
74
75 def moveRight(row, col):
76     global grid
77     if col <= 4:
78         if grid[row][col] == 'o' and grid[row][col+1] == 'o' and grid[row][col+2] == '.':
79             grid[row][col] = '.'
80             grid[row][col+1] = '.'
81             grid[row][col+2] = 'o'
82             return True
83     else:
84         return False
85     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.

```

88 def solver():
89     global grid
90     global endrow, endcol
91     global moves
92     global post
93     count = 0
94
95
96     for i in range(7):
97         for j in range(7):
98             if grid[i][j] == 'o':
99                 count += 1
100
101
102     #base case, only 1 peg left, and already at target destination
103     if count == 1 and grid[endrow][endcol] == 'o':
104         return True
105
106     #improperly placed
107     if count == 1 and grid[endrow][endcol] != 'o':
108         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.

```

111 #check every movement of every peg
112 for i in range(7):
113     for j in range(7):
114         if grid[i][j] == 'o':
115
116             moved = moveUp(i,j)
117             if moved == True:
118
119                 if solver():
120                     moves.append("up")
121                     post.append([i,j])
122                     return True
123             else: #restoration
124                 grid[i][j] = 'o'
125                 grid[i-1][j] = 'o'
126                 grid[i-2][j] = '.'

```

```

128     moved = moveDown(i,j)
129     if moved == True:
130
131         if solver():
132             moves.append("down")
133             post.append([i,j])
134             return True
135         else: #restoration
136             grid[i][j] = 'o'
137             grid[i+1][j] = 'o'
138             grid[i+2][j] = '.'
139
140     moved = moveLeft(i,j)
141     if moved == True:
142
143         if solver():
144             moves.append("left")
145             post.append([i,j])
146             return True
147         else: #restoration
148             grid[i][j] = 'o'
149             grid[i][j-1] = 'o'
150             grid[i][j-2] = '.'
151
152     moved = moveRight(i,j)
153     if moved == True:
154
155         if solver():
156             moves.append("right")
157             post.append([i,j])
158             return True
159         else: #restoration
160             grid[i][j] = 'o'
161             grid[i][j+1] = 'o'
162             grid[i][j+2] = '.'
163
164     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

```

170 if solver():
171     print("YES")
172 else:
173     print("NO")
174
175
176 printer()
177 print(moves)
178 print(post)

```

---

```

29 #grid printer
30 def printer():
31     global grid
32     for i in range(7):
33         for j in range(7):
34             print (grid[i][j], end=" ")
35         print()
36     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 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.

```
503 #count number of pegs
504 counter:
505     addi $sp, $sp, -20
506     sw $s0, 0($sp) #i
507     sw $s1, 4($sp) #j
508     sw $s2, 8($sp) #board[i][j]
509     sw $s3, 12($sp) #number of pegs
510     sw $ra, 16($sp)
511
512     li $s3, 0 #count of pegs
513     li $s0, 0 #i
514
515     for1:
516     li $s1, 0 #j
517
518     for2:
519     beq $s1, 7, check_outer
520
521     #check board[i][j]
522     mul $s2, $s0, 7
523     add $s2, $s2, $s1
524     mul $s2, $s2, 4
525
526     add $s2, $s2, board
527     lw $s2, ($s2)
528
529     beq $s2, 111, increment_count
530     j increment_index
531
532     increment_count:
533     addi $s3, $s3, 1
534
535     increment_index:
536     addi $s1, $s1, 1
537     j for2
538
539     check_outer:
540     addi $s0, $s0, 1
541     beq $s0, 7, counting_end
542     j for1
543
544     counting_end:
545     move $a1, $s3 #keep track of count of pegs
546     lw $s0, 0($sp)
547     lw $s1, 4($sp)
548     lw $s2, 8($sp)
549     lw $s3, 12($sp)
550     lw $ra, 16($sp)
551     addi $sp, $sp, 20
552     jr $ra
553
```

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  $\text{base address of board} + (i * 7 + j) * 4$ .

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)

```
112 check_movements:
113     li $s0, 0 #i
114     forr1:
115     li $s1, 0 #j
116     forr2:
117     beq $s1, 7, check_outerr
118
119     #check board[i][j]
120     mul $s2, $s0, 7
121     add $s2, $s2, $s1
122     mul $s2, $s2, 4
123
124     add $s2, $s2, board
125     lw $s2, ($s2)
126
127     beq $s2, 111, peg_detected #the character 'o' is found at board[i][j]
128     j increment_indexx #if current character is not a peg, increment index
129
130     peg_detected:
171
172     increment_indexx:
173     addi $s1, $s1, 1
174     j forr2
175
176     check_outerr:
177     addi $s0, $s0, 1
178     beq $s0, 7, solver_end
179     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)

```
130     peg_detected:
131     check_up:
132     jal moveUp
133     beq $a2, 1, up #check if we can move up
134     j check_down #if we cant go up, check if we can go down
135
136     up:
137     li $a2, 0
138     jal tryMoveUp
139     beq $v1, 1, solver_end
140
141
142     check_down:
143     jal moveDown
144     beq $a2, 1, down #check if we can move up
145     j check_left #if we cant go down, check left
146
147     down:
148     li $a2, 0
149     jal tryMoveDown
150     beq $v1, 1, solver_end
151
152     check_left:
153     jal moveLeft
154     beq $a2, 1, left #check if we can move left
155     j check_right #if we cant go left, check_right
156
157     left:
158     li $a2, 0
159     jal tryMoveLeft
160     beq $v1, 1, solver_end
161
162     check_right:
163     jal moveRight
164     beq $a2, 1, right #check if we can move right
165     j increment_indexx #no more possible movements, increment index
166
167     right:
168     li $a2, 0
169     jal tryMoveRight
170     beq $v1, 1, solver_end
171
172     increment_indexx:
173     addi $s1, $s1, 1
174     j forr2
175
```

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)

```

224 moveUp:
225     addi $sp, $sp, -4
226     sw $ra, 0($sp)
227
228     blt $s0, 2, moveUp_end
229
230     mul $s2, $s0, 7
231     add $s2, $s2, $s1
232     mul $s2, $s2, 4
233
234     add $s2, $s2, board
235     addi $s2, $s2, -28
236     move $t1, $s2
237     lw $s2, ($s2) #board[i-1][j]
238
239     beq $s2, 111, upCondition1 #if #board[i-1][j] = 'o'
240
241     moveUp_end:
242     lw $ra, 0($sp)
243     addi $sp, $sp, 4
244     jr $ra
245
246     upCondition1:
247     addi $t0, $0, 1
248     addi $t1, $t1, -28
249     lw $t1, ($t1) #board[i-2][j]
250     beq $t1, 46, upCondition2 #if #board[i-2][j] = '.'
251     j moveUp_end
252
253     upCondition2:
254     li $a2, 1
255     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

```

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

```
300 moveDown:
301     addi $sp, $sp, -4
302     sw $ra, 0($sp)
303
304     bgt $s0, 4, moveDown_end
305
306     mul $s2, $s0, 7
307     add $s2, $s2, $s1
308     mul $s2, $s2, 4
309
310     add $s2, $s2, board
311     addi $s2, $s2, 28
312     move $t1, $s2
313     lw $s2, ($s2) #board[i+1][j]
314
315     beq $s2, 111, downCondition1 #if #board[i+1][j] = 'o'
316
317     moveDown_end:
318     lw $ra, 0($sp)
319     addi $sp, $sp, 4
320     jr $ra
321
322     downCondition1:
323     addi $t0, $0, 1
324     addi $t1, $t1, 28
325     lw $t1, ($t1) #board[i+2][j]
326     beq $t1, 46, downCondition2 #if #board[i+2][j] = '.'
327     j moveDown_end
328
329     downCondition2:
330     li $a2, 1
331     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

```
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

```
376 moveLeft:
377     addi $sp, $sp, -4
378     sw $ra, 0($sp)
379
380     blt $s1, 2, moveLeft_end
381
382     mul $s2, $s0, 7
383     add $s2, $s2, $s1
384     mul $s2, $s2, 4
385
386     add $s2, $s2, board
387     addi $s2, $s2, -4
388     move $t1, $s2
389     lw $s2, ($s2) #board[i][j-1]
390
391     beq $s2, 111, leftCondition1 #if #board[i][j-1] = 'o'
392
393     moveLeft_end:
394     lw $ra, 0($sp)
395     addi $sp, $sp, 4
396     jr $ra
397
398     leftCondition1:
399     addi $t0, $0, 1
400     addi $t1, $t1, -4
401     lw $t1, ($t1) #board[i][j-2]
402     beq $t1, 46, leftCondition2 #if #board[i][j-2] = '.'
403     j moveLeft_end
404
405     leftCondition2:
406     li $a2, 1
407     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

```
63 def moveLeft(row, col):
64     global grid
65     if col >= 2:
66         if grid[row][col] == 'o' and grid[row][col-1] == 'o' and grid[row][col-2] == '.':
67             grid[row][col] = '.'
68             grid[row][col-1] = '.'
69             grid[row][col-2] = 'o'
70     return True
```

## moveRight

```
452 moveRight:
453     addi $sp, $sp, -4
454     sw $ra, 0($sp)
455
456     bgt $s1, 4, moveRight_end
457
458     mul $s2, $s0, 7
459     add $s2, $s2, $s1
460     mul $s2, $s2, 4
461
462     add $s2, $s2, board
463     addi $s2, $s2, 4
464     move $t1, $s2
465     lw $s2, ($s2) #board[i][j+1]
466
467     beq $s2, 111, rightCondition1 #if #board[i][j+1] = 'o'
468
469     moveRight_end:
470     lw $ra, 0($sp)
471     addi $sp, $sp, 4
472     jr $ra
473
474     rightCondition1:
475     addi $t0, $0, 1
476     addi $t1, $t1, 4
477     lw $t1, ($t1) #board[i][j+2]
478     beq $t1, 46, rightCondition2 #if #board[i][j+2] = '.'
479     j moveRight_end
480
481     rightCondition2:
482     li $a2, 1
483     j moveRight_end
484
```

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

```
75 ▼ def moveRight(row, col):
76     global grid
77 ▼     if col <= 4 :
78 ▼         if grid[row][col] == 'o' and grid[row][col+1] == 'o' and grid[row][col+2] == '.':
79             grid[row][col] = '.'
80             grid[row][col+1] = '.'
81             grid[row][col+2] = 'o'
82             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 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.

```
130     peg_detected:
131     check_up:
132     jal moveUp
133     beq $a2, 1, up #check if we can move up
134     j check_down #if we cant go up, check if we can go down
135
136     up:
137     li $a2, 0
138     jal tryMoveUp
139     beq $v1, 1, solver_end
140
141     check_down:
142     jal moveDown
143     beq $a2, 1, down #check if we can move up
144     j check_left #if we cant go down, check left
145
146     down:
147     li $a2, 0
148     jal tryMoveDown
149     beq $v1, 1, solver_end
150
151
152     check_left:
153     jal moveLeft
154     beq $a2, 1, left #check if we can move left
155     j check_right #if we cant go left, check_right
156
157     left:
158     li $a2, 0
159     jal tryMoveLeft
160     beq $v1, 1, solver_end
161
162     check_right:
163     jal moveRight
164     beq $a2, 1, right #check if we can move right
165     j increment_indexx #no more possible movements, increment index
166
167     right:
168     li $a2, 0
169     jal tryMoveRight
170     beq $v1, 1, solver_end
171
172     increment_indexx:
173     addi $s1, $s1, 1
174     j forr2
175
```

## tryMoveUp:

```
181 tryMoveUp:
182     addi $sp, $sp, -4
183     sw $ra, 0($sp)
184
185     mul $s2, $s0, 7
186     add $s2, $s2, $s1
187     mul $s2, $s2, 4
188     add $s2, $s2, board
189
190     li $t1, 46
191     sw $t1, ($s2) #grid[i][j] = '.'
192
193     addi $s2, $s2, -28
194     sw $t1, ($s2) #grid[i-1][j] = '.'
195
196     addi $s2, $s2, -28
197     li $t1, 111
198     sw $t1, ($s2) #grid[i-2][j] = 'o'
199
200     jal solver #if solver(), return True
201     beq $v1, 1, movedUp
202
203     #else we undo the movement
204     li $t1, 46
205     sw $t1, ($s2) #grid[i-2][j] = '.'
206
207     li $t1, 111
208     addi $s2, $s2, 28
209     sw $t1, ($s2) #grid[i-1][j] = 'o'
210
211     addi $s2, $s2, 28
212     sw $t1, ($s2) #grid[i][j] = 'o'
213
214     lw $ra, 0($sp)
215     addi $sp, $sp, 4
216     jr $ra
217
218 movedUp:
219     #printmsg1
220     lw $ra, 0($sp)
221     addi $sp, $sp, 4
222     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 `$v1`. If it is equal to 1, then we simply go to the label `movedUp`, where we restore the necessary registers and go back to `$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

```
257 tryMoveDown:
258     addi $sp, $sp, -4
259     sw $ra, 0($sp)
260
261     mul $s2, $s0, 7
262     add $s2, $s2, $s1
263     mul $s2, $s2, 4
264     add $s2, $s2, board
265
266     li $t1, 46
267     sw $t1, ($s2) #grid[i][j] = '.'
268
269     addi $s2, $s2, 28
270     sw $t1, ($s2) #grid[i+1][j] = '.'
271
272     addi $s2, $s2, 28
273     li $t1, 111
274     sw $t1, ($s2) #grid[i+2][j] = 'o'
275
276     jal solver #if solver(), return True
277     beq $v1, 1, movedDown
278
279     #else we undo the movement
280     li $t1, 46
281     sw $t1, ($s2) #grid[i+2][j] = '.'
282
283     li $t1, 111
284     addi $s2, $s2, -28
285     sw $t1, ($s2) #grid[i+1][j] = 'o'
286
287     addi $s2, $s2, -28
288     sw $t1, ($s2) #grid[i][j] = 'o'
289
290     lw $ra, 0($sp)
291     addi $sp, $sp, 4
292     jr $ra
293
294     movedDown:
295     #printmsg2
296     lw $ra, 0($sp)
297     addi $sp, $sp, 4
298     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 `$v1`. If it is equal to 1, then we simply go to the label `movedDown`, where we restore the necessary registers and go back to `$ra`. If not, then we undo our modifications to the board, then restore the necessary registers and return to `$ra`.

## tryMoveLeft

```
333 tryMoveLeft:
334     addi $sp, $sp, -4
335     sw $ra, 0($sp)
336
337     mul $s2, $s0, 7
338     add $s2, $s2, $s1
339     mul $s2, $s2, 4
340     add $s2, $s2, board
341
342     li $t1, 46
343     sw $t1, ($s2) #grid[i][j] = '.'
344
345     addi $s2, $s2, -4
346     sw $t1, ($s2) #grid[i][j-1] = '.'
347
348     addi $s2, $s2, -4
349     li $t1, 111
350     sw $t1, ($s2) #grid[i][j-2] = 'o'
351
352     jal solver #if solver(), return True
353     beq $v1, 1, movedLeft
354
355     #else we undo the movement
356     li $t1, 46
357     sw $t1, ($s2) #grid[i][j-2] = '.'
358
359     li $t1, 111
360     addi $s2, $s2, 4
361     sw $t1, ($s2) #grid[i][j-1] = 'o'
362
363     addi $s2, $s2, 4
364     sw $t1, ($s2) #grid[i][j] = 'o'
365
366     lw $ra, 0($sp)
367     addi $sp, $sp, 4
368     jr $ra
369
370 movedLeft:
371     #printmsg3
372     lw $ra, 0($sp)
373     addi $sp, $sp, 4
374     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[i][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 `$v1`. If it is equal to 1, then we simply go to the label `movedLeft`, where we restore the necessary registers and go back to `$ra`. If not, then we undo our modifications to the board, then restore the necessary registers and return to `$ra`.

## tryMoveRight

```
409 tryMoveRight:
410     addi $sp, $sp, -4
411     sw $ra, 0($sp)
412
413     mul $s2, $s0, 7
414     add $s2, $s2, $s1
415     mul $s2, $s2, 4
416     add $s2, $s2, board
417
418     li $t1, 46
419     sw $t1, ($s2) #grid[i][j] = '.'
420
421     addi $s2, $s2, 4
422     sw $t1, ($s2) #grid[i][j+1] = '.'
423
424     addi $s2, $s2, 4
425     li $t1, 111
426     sw $t1, ($s2) #grid[i][j+2] = 'o'
427
428     jal solver #if solver(), return True
429     beq $v1, 1, movedRight
430
431     #else we undo the movement
432     li $t1, 46
433     sw $t1, ($s2) #grid[i][j+2] = '.'
434
435     li $t1, 111
436     addi $s2, $s2, -4
437     sw $t1, ($s2) #grid[i][j+1] = 'o'
438
439     addi $s2, $s2, -4
440     sw $t1, ($s2) #grid[i][j] = 'o'
441
442     lw $ra, 0($sp)
443     addi $sp, $sp, 4
444     jr $ra
445
446 movedRight:
447     #printmsg4
448     lw $ra, 0($sp)
449     addi $sp, $sp, 4
450     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[i][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 `$v1`. If it is equal to 1, then we simply go to the label `movedRight`, where we restore the necessary registers and go back to `$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.

```
112 check_movements:
113     li $s0, 0 #i
114     forr1:
115         li $s1, 0 #j
116         forr2:
117             beq $s1, 7, check_outerr
118
119             #check board[i][j]
120             mul $s2, $s0, 7
121             add $s2, $s2, $s1
122             mul $s2, $s2, 4
123
124             add $s2, $s2, board
125             lw $s2, ($s2)
126
127             beq $s2, 111, peg_detected #the character 'o' is found at board[i][j]
128             j increment_indexx #if current character is not a peg, increment index
129
130     peg_detected:
171
172     increment_indexx:
173         addi $s1, $s1, 1
174         j forr2
175
176     check_outerr:
177         addi $s0, $s0, 1
178         beq $s0, 7, solver_end
179         j forr1
```

```
130     peg_detected:
131     check_up:
132         jal moveUp
133         beq $a2, 1, up #check if we can move up
134         j check_down #if we cant go up, check if we can go down
135
136     up:
137         li $a2, 0
138         jal tryMoveUp
139         beq $v1, 1, solver_end
140
141     check_down:
142         jal moveDown
143         beq $a2, 1, down #check if we can move up
144         j check_left #if we cant go down, check left
145
146     down:
147         li $a2, 0
148         jal tryMoveDown
149         beq $v1, 1, solver_end
150
151
152     check_left:
153         jal moveLeft
154         beq $a2, 1, left #check if we can move left
155         j check_right #if we cant go left, check_right
156
157     left:
158         li $a2, 0
159         jal tryMoveLeft
160         beq $v1, 1, solver_end
161
162     check_right:
163         jal moveRight
164         beq $a2, 1, right #check if we can move right
165         j increment_indexx #no more possible movements, increment index
166
167     right:
168         li $a2, 0
169         jal tryMoveRight
170         beq $v1, 1, solver_end
171
172     increment_indexx:
173         addi $s1, $s1, 1
174         j forr2
175
```

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.

```
111 #check every movement of every peg
112 ▼ for i in range(7):
113 ▼   for j in range(7):
114 ▼     if grid[i][j] == 'o':
115
116         moved = moveUp(i,j)
117 ▼         if moved == True:
118
119             if solver():
120                 moves.append("up")
121                 post.append([i,j])
122                 return True
123 ▼             else: #restoration
124                 grid[i][j] = 'o'
125                 grid[i-1][j] = 'o'
126                 grid[i-2][j] = '.'
127
128         moved = moveDown(i,j)
129 ▼         if moved == True:
130
131             if solver():
132                 moves.append("down")
133                 post.append([i,j])
134                 return True
135 ▼             else: #restoration
136                 grid[i][j] = 'o'
137                 grid[i+1][j] = 'o'
138                 grid[i+2][j] = '.'
139
140
141 ▼         moved = moveLeft(i,j)
142         if moved == True:
143
144             if solver():
145                 moves.append("left")
146                 post.append([i,j])
147 ▼                 return True
148             else: #restoration
149                 grid[i][j] = 'o'
150                 grid[i][j-1] = 'o'
151                 grid[i][j-2] = '.'
152
153 ▼         moved = moveRight(i,j)
154         if moved == True:
155
156             if solver():
157                 moves.append("right")
158                 post.append([i,j])
159 ▼                 return True
160             else: #restoration
161                 grid[i][j] = 'o'
162                 grid[i][j+1] = 'o'
163                 grid[i][j+2] = '.'
164         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.”

```
77 main:
78     la board, game_board #set the address of our array
79     move $t0, board
80
81     #7 lines of input, build the board
82     jal input
83     jal input
84     jal input
85     jal input
86     jal input
87     jal input
88     jal input
89
90     jal solver
91
92
93     beq $v1, 1, yes
94     printNO
95     exit
96
97     yes:
98     printYES
99     exit
```

```
63 .macro printYES
64     li $v0, 4
65     la $a0, msg5
66     syscall
67 .end_macro
68
69 .macro printNO
70     li $v0, 4
71     la $a0, msg6
72     syscall
73 .end_macro
```

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

# Test Cases

<div>Input: xx...xx xxo..xx ..o.... ..oO.. ..... xx...xx xx...xx Output: YES</div>	<div>Input: xx...xx xxo..xx ..o.... E.oo.. ..... xx...xx xx...xx Output: YES</div>	<div>Input: xx...xx xxo..xx ..O.... ..oo.. ..... xx...xx xx...xx Output: NO</div>	<div>Input: xx...xx xxo..xx ..o.... ..oo.. ..... xx.E.xx xx...xx Output: NO</div>	<div>Input: xx.o.xx xxooOxx ..oo.. ...oo.. ..... xx...xx xx...xx Output: YES</div>
<div>Input: ...xE.o ...oo.. ...oo.. ..... ..... ..... ..... ..... Output: YES</div>	<div>Input: ..... ..... ..... ...E.. ..... ..... ..... ..... Output: NO</div>	<div>Input: ..... ..... ..... ..... ..... ..... ..... ....O. Output: YES</div>	<div>Input: ...xx.x ...xE.o ...oo.. x..oo.. ..xxx.. ...x... o..x... Output: NO</div>	<div>Input: xx.o.xx xxo.oxx ..ooE.. ..O.O.. ...O.. xx...xx xx...xx Output: YES</div>