ENCARNACION, Stephen Mary S.

CS 21 – Lab 2

# CS 21 Replacement Machine Problem - Maze Solver

## Documentation Video Link:

https://drive.google.com/file/d/1NYnaixDC7chupQdKNwKyYv1bhTPwtZ9L/view?usp=sharing

## 202002988.asm (Maze Solver): How it Works

We first initialize two arrays, each with 108 elements. One will be used to store the input, and the other to store the distances from the start point to every other traversable point. The second array will initially contain 108 zeroes, since traversing has not yet begun. Since we are working with 6 rows and 18 columns, we can think of our array as a 2d array.

We then ask the user for input, 6 times. For each input, we have a string that we will iterate through, storing each character to our array, until it is filled. We also keep track of the 'S' and 'F' characters in our input, since these characters signify the start and finish points of the maze. We store the addresses of each, so that we can locate the start and finish points of both mazes.

The starting point of the second maze will be set to have the value of 1. From the starting point of the first maze, we will check for every traversable and unvisited spot in the eight directions and give a value to it, and increment our index, and this will continue until the finish point of the second maze contains a nonzero value, meaning that it has been already visited.

We can now obtain the shortest distance from the start to the end point, which will be printed later. Now, remember that the second maze contains distances from the start point. We start at the finish point of the second maze, and let's say it has the value 18. We check the eight directions for 17, then we go to that point. Using this index, we can go to its corresponding index in the first maze, and change the value from '0' to '1'. We keep checking and moving, until we are at a spot that contains the value of 1. This means that we are already at the start point, and that we have traversed from the finish point to the start point.

After modifying the first maze, it now contains the solution to our problem, the shortest path from start to finish. We now print this maze, along with the shortest distance.

# Python Equivalent

Getting user input and creating a 2-d array.

```
board = []
row = []

for i in range(6):
    row1 = input().split()
    for j in range(18):
        row.append(row1[0][j])

    board.append(row)
    row = []
```

Locating the start and end points. Setting up the second 2-d array.

```

for i in range(6):
    for j in range(18):
        if board[i][j] == 'S':
            board[i][j] = '0'
            start = i,j

        if board[i][j] == 'F':
            end = i,j
            board[i][j] = '0'

bfs_board = [
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
]

i,j = start
bfs_board[i][j] = 1
```

Defining a function that will facilitate movement through the 2d array, and modify its contents if the conditions are met.

```
# we fill our bfs board, we can get the shortest distance from the start point to every traversable point in the board
#visit every traversable point
def pathing(k):
    for i in range(6):
        for j in range(18):
            if bfs_board[i][j] == k:
                def moveNE():
                    if i>0 and j<17 and bfs_board[i-1][j+1] == 0 and board[i-1][j+1] == "0":
                        bfs_board[i-1][j+1] = k + 1

                def moveNW():
                    if i>0 and j>0 and bfs_board[i-1][j-1] == 0 and board[i-1][j-1] == "0":
                        bfs_board[i-1][j-1] = k + 1

                def moveSE():
                    if i<5 and j<17 and bfs_board[i+1][j+1] == 0 and board[i+1][j+1] == "0":
                        bfs_board[i+1][j+1] = k + 1

                def moveSW():
                    if i<5 and j>0 and bfs_board[i+1][j-1] == 0 and board[i+1][j-1] == "0":
                        bfs_board[i+1][j-1] = k + 1

```

```
        def moveUp():
          if i>0 and bfs_board[i-1][j] == 0 and board[i-1][j] == "0":
            bfs_board[i-1][j] = k + 1

        def moveDown():
          if i<5 and bfs_board[i+1][j] == 0 and board[i+1][j] == "0":
            bfs_board[i+1][j] = k + 1

        def moveLeft():
          if j>0 and bfs_board[i][j-1] == 0 and board[i][j-1] == "0":
            bfs_board[i][j-1] = k + 1

        def moveRight():
          if j<17 and bfs_board[i][j+1] == 0 and board[i][j+1] == "0":
            bfs_board[i][j+1] = k + 1

        moveNE() #move ne
        moveNW() #move nw
        moveSE() #move se
        moveSW() #move sw
        moveUp() #move up
        moveDown()  #move down
        moveLeft() #move left
        moveRight() #move right


```

Travel the 2d array until the finish point is reached, while keeping track of the shortest distance.

```

k = 0
while bfs_board[end[0]][end[1]] == 0:
    k += 1
    pathing(k)

shortest_path = k

i, j = end
k = bfs_board[i][j] #bfs_board[end]
```

Create functions that will help in retracing our steps, from the finish point to the start point.

```python
#while we are not yet back to the start point, try to retrace our steps, while modifying the input board so that it will contain the solution
while k > 1:
    def checkNE():
        if i > 0 and j < 17 and bfs_board[i-1][j + 1] == k-1:
            return True
    def checkNW():
        if i > 0 and j > 0 and bfs_board[i-1][j - 1] == k-1:
            return True

    def checkSE():
        if i <5 and j < 17 and bfs_board[i+1][j + 1] == k-1:
            return True

    def checkSW():
        if i <5 and j >0 and bfs_board[i+1][j - 1] == k-1:
            return True

    def checkUp():
        if i > 0 and bfs_board[i - 1][j] == k-1:
            return True

    def checkDown():
        if i < 5 and bfs_board[i + 1][j] == k-1:
            return True

    def checkLeft():
        if j > 0 and bfs_board[i][j - 1] == k-1:
            return True

    def checkRight():
        if j < 17 and bfs_board[i][j + 1] == k-1:
            return True
```

Try to move in the eight directions

```python
    #check ne
    if checkNE():
        i,j = i-1, j+1
        board[i][j] = 1
        k-=1

    #check nw
    elif checkNW():
        i,j = i-1, j-1
        board[i][j] = 1
        k-=1

    #check se
    elif checkSE():
        i,j = i+1, j+1
        board[i][j] = 1
        k-=1

    #check sw
    elif checkSW():
        i,j = i+1, j-1
        board[i][j] = 1
        k-=1


    #check up
    elif checkUp():
        i, j = i-1, j
        board[i][j] = 1
        k-=1

    #check down
    elif checkDown():
        i, j = i+1, j
        board[i][j] = 1
        k-=1

```

```python
  #check left
  elif checkLeft():
    i, j = i, j-1
    board[i][j] = 1
    k-=1
  #check right
  elif checkRight():
    i, j = i, j+1
    board[i][j] = 1
    k -= 1
print()
```

We now have the solution to our maze. Print the 2d array then print the shortest path.

```python
board[start[0]][start[1]] = "S"
board[end[0]][end[1]] = "F"

for i in board:
  for j in i:
    print(j, end="")
  print()
print(shortest_path)
```

# 202002988.asm (Maze Solver) In-depth Explanation

We first define some macros and .eqv that will be frequently used in the program. The .eqv start will be used to contain the address of the character ‘S’ in the input board, while the .eqv end will be used to contain the address of the character ‘F’. The .eqv board will be used to contain the base address of the input board, while the .eqv bfsboard will be used to contain the base address of the second array.

The .macro save_register saves various registers to the stack, and these are to be used in functions, along with the restore_register, which restores registers. The .macros in the second image simply do what their name says, ending the program, printing a newline, reading a string, printing a character, and printing an integer.

```
.eqv start, $s5
.eqv end, $s6
.eqv board, $s7
.eqv bfsboard, $t5

.macro save_register
        addi $sp, $sp, -24
        sw $s0, 0($sp)
        sw $s1, 4($sp)
        sw $s2, 8($sp)
        sw $s3, 12($sp)
        sw board, 16($sp)
        sw bfsboard, 20($sp)
.end_macro


.macro restore_register
        lw bfsboard, 20($sp)
        lw board, 16($sp)
        lw $s3, 12($sp)
        lw $s2, 8($sp)
        lw $s1, 4($sp)
        lw $s0, 0($sp)
        addi $sp, $sp, 24
.end_macro
```

```
.macro exit #ends the program
        li      $v0, 10
        syscall
.end_macro

.macro newline #prints a newline
        li      $a0, 0xA
        li      $v0, 11
        syscall
.end_macro

.macro read_str # reads input
        li $v0, 8
        la $a0, line
        li $a1, 20
        syscall
.end_macro

.macro print_char(%n)
        addi    $a0, %n, 0
        li      $v0, 11
        syscall
.end_macro

.macro print_int(%n)
        addi    $a0, %n, 0
        li      $v0, 1
        syscall

.end_macro
```

Onto the .data segment, we set line: .space 20. For each line of input later, we are expecting 19 characters, the 18 characters and a newline. We then initialize the input_board and bfs_board to have 108 elements, and those are all zeroes.

```
.data
	line: .space 20
	input_board: .word 0:108 #initialize the bfs board, which will be key in solving the problem
	bfs_board: .word 0:108 #initialize the bfs board, which will be key in solving the problem
```

Let's now proceed to the main program. We first load the address of our array, input_board, to board (register $s7). We then copy this to the register $t2. We then jump and link to input 6 times.

```
.text
main:

	la board, input_board #set the address of our array
	move $t2, board #we save the address of the array so that we can traverse through it
	#run the input function 6 times, thus filling the array with 108 elements
	jal input
	jal input
	jal input
	jal input
	jal input
	jal input
```

Now, for the input function, we first save the necessary registers, then set the value of \$s2 to 48, which is the character '0' in ASCII. We the ask an input, then iterate through the string given, checking for each character and stores them to our array. This is repeated 18 times, since we have 18 elements per "row". Also, if the character detected is equal to 83/'S' in ASCII, or 70/'F' in ASCII, we go to either found_start or found_end, and store the address to start (\$s5) or end (\$s6), and change the character from 'S'/'F' to '0', by changing the value of the register to 48.

```
input:
        addi $sp, $sp, -12
        sw $s0, 0($sp)
        sw $s1, 4($sp)
        sw $s2, 8($sp)

        li $s2, 48
        read_str
        loop_store:
        beq $s0, 18, end_input
        lb $s1, line($s0)

        beq $s1, 83, found_start
        beq $s1, 70, found_end

        return:
        sw $s1, ($t2)
        addi $t2, $t2, 4
        addi $s0, $s0, 1
        j loop_store

        end_input:
        lw $s2, 8($sp)
        lw $s1, 4($sp)
        lw $s0, 0($sp)
        addi $sp, $sp, 12
        jr $ra

        found_start: #we store the starting point, and change 'S' to '0'
        la start, ($t2)
        move $s1, $s2
        j return


        found_end: #we store the starting point, and change 'F' to '0'
        la end, ($t2)
        move $s1, $s2
        j return
```

Once we have set up the two arrays, it is now time to proceed. We load the base address of our array bfs_board to bfsboard($t5). Remember that the start contains the address of the start point in the first array. Since the two arrays are 108 words apart, we multiply 108 by 4 = 432, then we add that to our start point. We now have the address of the starting point of the second array. Following the same logic, we can use the address of the ending point of the first array in order to locate the ending point of the second array. We set the value of the starting point of bfs_board to 1, then we prepare to explore bfs_board.

Note that $t0 has the address of the ending point of bfs_board. At first, the ending point of bfs_board contains the value of 0, meaning that it is unexplored yet. While it is 0, we increment $a0 by 1 ($a0 will keep track of the value k), and go to the function pathing.

```
la bfsboard, bfs_board
li $t0, 432 #the difference between bfs_board and input_board
move $t1, start
add $t0, $t0, $t1

#we go to the starting point in the bfs board, and initialize it to 1
addi $s1, $0, 1
sw $s1, ($t0)

#explore the bfs board
li $a0, 0

li $t0, 432 #the difference between bfs_board and input_board
move $t1, end
add $t0, $t0, $t1 #store the ending point of bfs_board

#while the ending is unexplored, loop
explore_loop:
lw $s1, ($t0)
bnez $s1, proceed_now
addi $a0, $a0, 1
jal pathing
j explore_loop
```

The pathing function. We first store the necessary registers, then create a nested for loop. We can think of it as

for i in range(6):

for j in range(18):

We can use i and j to access the elements in our array, since we are thinking of it as a 2d array. The formula to convert it to an address is: base address + (i * 16 + j) * 4.  We check if bfs_board[i][j] = k. If not, we will increment j by 1. At first, k will be equal to 1. When bfs_board[i][j] = k, it means that we are now in a valid spot, then we try visit the eight directions with their respective functions. Once we have tried to visit the eight directions, we then increment j by 1, and once i becomes equal to 6, it is time to end the function.

```
pathing:
	addi $sp, $sp, -24
	sw $s0, 0($sp)
	sw $s1, 4($sp)
	sw $s2, 8($sp)
	sw $s3, 12($sp)
	sw $s4, 16($sp)
	sw $ra, 20($sp)

	#looping
	#for i in range(6) i = $s0
		#for j in range(18) j = $s1

	li $s0, 0

	for1:
	li $s1, 0

	for2:
	beq $s1, 18, check_outer

	#to locate index, index = base address + (i * 16 + j) * 4
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#bfs_board[i][j] == k check
	add $s2, $s2, bfsboard
	lw $s3, ($s2)
	bne $s3, $a0, pathing_increment

	jal moveNE
	jal moveNW
	jal moveSE
	jal moveSW
	jal moveUp
	jal moveDown
	jal moveLeft
	jal moveRight

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

	check_outer:
	addi $s0, $s0, 1
	beq $s0, 6, pathing_end
	j for1

	pathing_end:
	lw $ra, 20($sp)
	lw $s4, 16($sp)
	lw $s3, 12($sp)
	lw $s2, 8($sp)
	lw $s1, 4($sp)
	lw $s0, 0($sp)
	addi $sp, $sp, 12
	jr $ra
```

Now, for the eight movement functions, we first save the registers, then set $s3 to have the value of $a0. We take note of the respective constraints of each function. An i with a value of less than or equal to 0 means that movement upwards is prohibited, an i with a value of greater than or equal to 5 means that movement downwards is prohibited, a j with a value of less than or equal to 0 means that movement to the left is prohibited, and a j with a value of greater than or equal to 17 means that movement to the right is prohibited. If we fail any of the conditions here, the movement function ends prematurely. If movement is allowed, we then make the necessary changes to our i and j value, then convert it to an address using the given formula base address + (i * 16 + j) * 4. We then check if bfs_board[i][j] is equal to 0, meaning that it is unvisited. If it has a nonzero value, then that means it is already visited, thus we terminate the function. If bfs_board[i][j] is equal to 0, we then check for board[i][j], if it has the value 48/'0' in ASCII. At this point, the elements in the array board are all either '0' or 'X'. If board[i][j] is a '0', then we can proceed, else we terminate the function and return. If Board[i][j] is indeed a '0', we increment $s3 by 1 (k+1), then store it to bfs_board[i][j]. We then terminate the function, restoring the registers and returning back. bfs_board will slowly fill up to contain the distance from the start point to the various points.

```
moveNE:
	save_register
	move $s3, $a0

	blez $s0, endNE
	bge $s1, 17, endNE

	addi $s0, $s0, -1
	addi $s1, $s1, 1


	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add board, board, $s2 #board[i][j]
	add bfsboard, bfsboard, $s2 #bfs_board[i][j]

	lw $s2, (bfsboard)
	bnez  $s2, endNE

	lw $s2, (board)
	bne $s2, 48, endNE

	add $s3, $s3, 1
	sw $s3, (bfsboard)

	endNE:
	restore_register
	jr $ra
```

```
moveNW:
	save_register

	move $s3, $a0

	blez $s0, endNW
	blez $s1, endNW

	addi $s0, $s0, -1
	addi $s1, $s1, -1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add board, board, $s2
	add bfsboard, bfsboard, $s

	lw $s2, (bfsboard)
	bnez  $s2, endNW

	lw $s2, (board)
	bne $s2, 48, endNW

	add $s3, $s3, 1
	sw $s3, (bfsboard)

	endNW:
	restore_register
	jr $ra
```

```
moveSE:
	save_register

	move $s3, $a0

	bge $s0, 5, endSE
	bge $s1, 17, endSE

	addi $s0, $s0, 1
	addi $s1, $s1, 1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bnez  $s2, endSE

	lw $s2, (board)
	bne $s2, 48, endSE

	add $s3, $s3, 1
	sw $s3, (bfsboard)

	endSE:
	restore_register
	jr $ra
```

```
moveSW:
	save_register
	move $s3, $a0

	bge $s0, 5, endSW
	blez $s1, endSW

	addi $s0, $s0, 1
	addi $s1, $s1, -1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bnez  $s2, endSW

	lw $s2, (board)
	bne $s2, 48, endSW

	add $s3, $s3, 1
	sw $s3, (bfsboard)

	endSW:
	restore_register
	jr $ra
```

```
moveUp:
	save_register
	move $s3, $a0

	blez $s0, endUp
	addi $s0, $s0, -1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add board, board, $s2
	add bfsboard, bfsboard, $s

	lw $s2, (bfsboard)
	bnez  $s2, endUp

	lw $s2, (board)
	bne $s2, 48, endUp

	add $s3, $s3, 1
	sw $s3, (bfsboard)

	endUp:
	restore_register
	jr $ra
```

```
moveDown:
	save_register
	move $s3, $a0

	bge $s0, 5, endDown
	addi $s0, $s0, 1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add board, board, $s2
	add bfsboard, bfsboard,

	lw $s2, (bfsboard)
	bnez  $s2, endDown

	lw $s2, (board)
	bne $s2, 48, endDown

	add $s3, $s3, 1
	sw $s3, (bfsboard)

	endDown:
	restore_register
	jr $ra
```

```
moveLeft:
	save_register
	move $s3, $a0

	blez $s1, endLeft
	addi $s1, $s1, -1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add board, board, $s2
	add bfsboard, bfsboard, $s

	lw $s2, (bfsboard)
	bnez  $s2, endLeft

	lw $s2, (board)
	bne $s2, 48, endLeft

	add $s3, $s3, 1
	sw $s3, (bfsboard)

	endLeft:
	restore_register
	jr $ra
```

```
moveRight:
	save_register
	move $s3, $a0

	bge $s1, 17, endRight
	addi $s1, $s1, 1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bnez  $s2, endRight

	lw $s2, (board)
	bne $s2, 48, endRight

	add $s3, $s3, 1
	sw $s3, (bfsboard)

	endRight:
	restore_register
	jr $ra
```

Eventually, bfs_board[end point] will contain a value, thus becoming visited. $a0 (k) contains the shortest distance from the start point to the end point, so we store it to $a3.

We have the address of the end point. We try to convert it in terms of i and j, so we set $t0 to end – board. (address of end point – base address of board). We then divide this by 4

$t0 / 18 will be our i, and $t0 mod 18 will be our j.

We then store to $a0 bfs_board[i][j]. Remember that bfs_board[start point] was initialized to contain the value 1. If $a0 is less than or equal to 1, that means we are back to the starting point, so we then go to the ending part of the main function. Else, we try to retrace our steps with the following functions.

```
	proceed_now:
	#find the coordinates of the end point, find i and j
	move $a3, $a0 #we move to $a3 the shortest distance

	subu $t0, end, board
	divu $t0, $t0, 4
	mflo $t0

	divu $t6, $t0, 18 #i
	mflo $t6

	divu $t7, $t0, 18 #j
	mfhi $t7

	addi $t0, end, 432 #ending point in bfs board, go back to starting point
	lw $a0, ($t0) #store bfs_board[i][j]

	retrace:
	ble $a0, 1, ending

	#can only take one of the following moves, so we set up a safety measure
	#a move is taken once $a0 is decremented
```

There is a path from the start to the end point. We can retrace our steps in 8 possible directions, but only 1 can be taken. We set $a1 to have the value of $a0. For a successful movement check, $a0 will be decremented by 1. After each movement check call, we try to check if $a0 and $a1 are still equal. If they are not, then that means the function is successful, and we have retraced our step, so we go back and try to retrace again.

```
#can only take one of the following moves, so we set up a safety measure
#a move is taken once $a0 is decremented

move $a1, $a0

jal checkNE
bne $a0, $a1, retrace

jal checkNW
bne $a0, $a1, retrace

jal checkSE
bne $a0, $a1, retrace

jal checkSW
bne $a0, $a1, retrace

jal checkUp
bne $a0, $a1, retrace

jal checkDown
bne $a0, $a1, retrace

jal checkLeft
bne $a0, $a1, retrace

jal checkRight
bne $a0, $a1, retrace
```

We have here the 8 movement check functions. We first save the necessary registers, then store our i and j. We also store the value of k-1 to $a3. We first check if movement upwards, downwards, left or right is allowed, by checking the values of i and j. If movement is not allowed, then we go to the end of the function. Else, we make the necessary changes to i and j.  We convert it by using base address + (i * 16 + j) * 4 to check board[i][j] and bfs_board[i][j]

We then check if bfs_board[i][j] is only 1 movement away from the original point. If not, then that means we have to end the function. If yes, then that means we can retrace our step to here. We go to board[i][j], then store the value ‘1’ to it. These will continue on, until we have gone back to our start point.

```
checkSE:
	save_register
	move $s0, $t6 #store i
	move $s1, $t7 #store j
	subi $s3, $a0, 1 # k-1

	bge $s0, 5, checkSE_end
	bge $s1, 17, checkSE_end
	addi $s0, $s0, 1
	addi $s1, $s1, 1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#adding base addresses
	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bne $s2, $s3 checkSE_end
	#board[i][j] = '1', from '0' (ASCII)
	lw $s2, (board)
	addi $s2, $s2, 1
	sw $s2, (board)
	subi $a0, $a0, 1
	addi $t6, $t6, 1
	addi $t7, $t7, 1

	checkSE_end:
	restore_register
	jr $ra
```

```
checkSW:
	save_register
	move $s0, $t6 #store i
	move $s1, $t7 #store j
	subi $s3, $a0, 1 # k-1

	bge $s0, 5, checkSW_end
	blez $s1, checkSW_end
	addi $s0, $s0, 1
	addi $s1, $s1, -1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#adding base addresses
	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bne $s2, $s3 checkSW_end
	#board[i][j] = 1
	lw $s2, (board)
	addi $s2, $s2, 1
	sw $s2, (board)
	subi $a0, $a0, 1
	addi $t6, $t6, 1
	addi $t7, $t7, -1

	checkSW_end:
	restore_register
	jr $ra
```

```
checkNE:
	save_register
	move $s0, $t6 #store i
	move $s1, $t7 #store j
	subi $s3, $a0, 1 # k-1

	blez $s0, checkNE_end
	bge $s1, 17, checkNE_end
	addi $s0, $s0, -1
	addi $s1, $s1, 1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#adding base addresses
	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bne $s2, $s3 checkNE_end
	#board[i][j] = 1, from 0 (ASCII)
	lw $s2, (board)
	addi $s2, $s2, 1
	sw $s2, (board)
	subi $a0, $a0, 1
	addi $t6, $t6, -1
	addi $t7, $t7, 1

	checkNE_end:
	restore_register
	jr $ra
```

```
checkNW:
	save_register
	move $s0, $t6 #store i
	move $s1, $t7 #store j
	subi $s3, $a0, 1 # k-1

	blez $s0, checkNW_end
	blez $s1, checkNW_end
	addi $s0, $s0, -1
	addi $s1, $s1, -1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#adding base addresses
	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bne $s2, $s3 checkNW_end
	#board[i][j] = 1, from 0 (ASCII)
	lw $s2, (board)
	addi $s2, $s2, 1
	sw $s2, (board)
	subi $a0, $a0, 1
	addi $t6, $t6, -1
	addi $t7, $t7, -1

	checkNW_end:
	restore_register
	jr $ra
```

```
checkUp:
	save_register
	move $s0, $t6 #store i
	move $s1, $t7 #store j
	subi $s3, $a0, 1 # k-1

	blez $s0, checkUp_end
	addi $s0, $s0, -1
	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#adding base addresses
	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bne $s2, $s3 checkUp_end

	#board[i][j] = 1, from 0 (ASCII)
	lw $s2, (board)
	addi $s2, $s2, 1
	sw $s2, (board)

	subi $a0, $a0, 1
	addi $t6, $t6, -1

	checkUp_end:
	restore_register
	jr $ra
```

```
checkDown:
	save_register
	move $s0, $t6 #store i
	move $s1, $t7 #store j
	subi $s3, $a0, 1 # k-1

	bge $s0, 5, checkDown_end
	addi $s0, $s0, 1
	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#adding base addresses
	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bne $s2, $s3 checkDown_end

	#board[i][j] = 1, from 0 (ASCII)
	lw $s2, (board)
	addi $s2, $s2, 1
	sw $s2, (board)

	subi $a0, $a0, 1
	addi $t6, $t6, 1

	checkDown_end:
	restore_register
	jr $ra
```

```
checkLeft:
	save_register
	move $s0, $t6 #store i
	move $s1, $t7 #store j
	subi $s3, $a0, 1 # k-1
	blez $s1,checkLeft_end
	addi $s1, $s1, -1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#adding base addresses
	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bne $s2, $s3 checkLeft_end

	#board[i][j] = 1, from 0 (ASCII)
	lw $s2, (board)
	addi $s2, $s2, 1
	sw $s2, (board)

	subi $a0, $a0, 1
	addi $t7, $t7, -1

	checkLeft_end:
	restore_register
	jr $ra
```

```
checkRight:
	save_register
	move $s0, $t6 #store i
	move $s1, $t7 #store j
	subi $s3, $a0, 1 # k-1

	bge $s1, 17, checkRight_end
	addi $s1, $s1, 1

	#$s2 contains the index
	mul $s2, $s0, 18
	add $s2, $s2, $s1
	mul $s2, $s2, 4

	#adding base addresses
	add board, board, $s2
	add bfsboard, bfsboard, $s2

	lw $s2, (bfsboard)
	bne $s2, $s3 checkRight_end

	#board[i][j] = 1, from 0 (ASCII)
	lw $s2, (board)
	addi $s2, $s2, 1
	sw $s2, (board)

	subi $a0, $a0, 1
	addi $t7, $t7, 1

	checkRight_end:
	restore_register
	jr $ra
```

```
checkDown:
```

Eventually, $a0, a.k.a. bfs_board[i][j] will be equal to 1. We are now at the start point, and we have successfully created a path that can be seen in board. It is now time to print our answers. First, we print the solved maze, then the shortest distance.

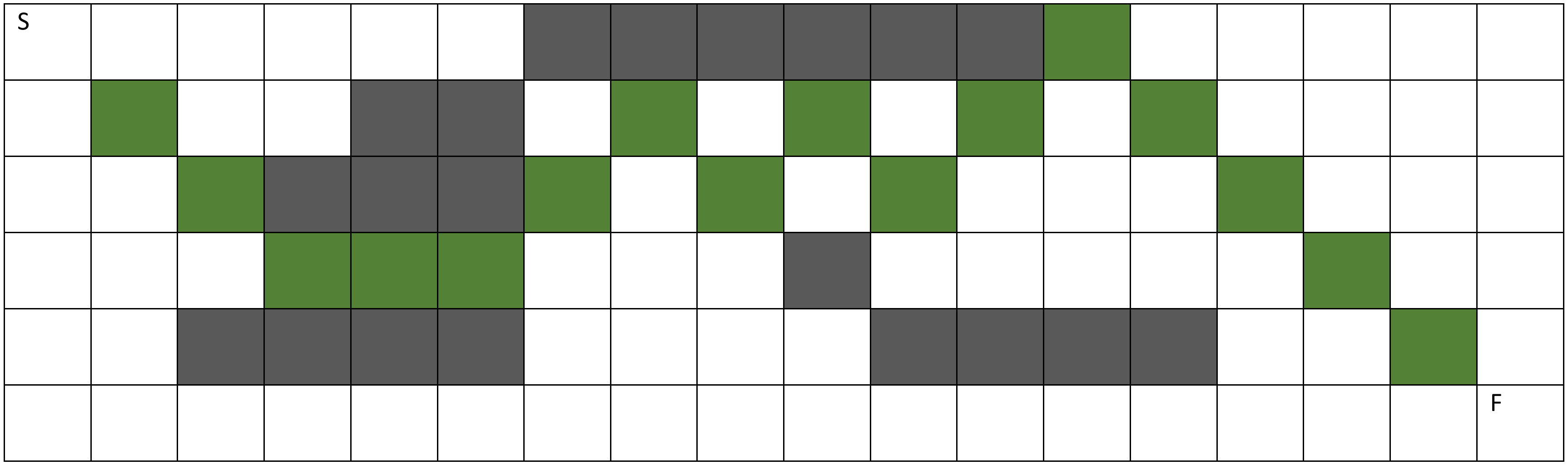
```
jal print_board
print_int($a3) #print the shortest distance
exit
```

For the print_board function, we begin by saving the necessary registers. What happens here is like a nested for loop. For i in list, for j in i. We print 18 elements, then enter a newline, then 18 elements again, then newline, until all 108 elements of the maze have been printed. $t2 initially contains the base address of the board, and we increment it by 4 as we traverse through the array. Also, $t2 will be checked if it is equal to start/end. Remember that our board was changed so that characters ‘S’/’F’ have been replaced. If $t2 is equal to start/end, then we print the characters ‘S’/’F

```
print_board: #prints the solved board
	addi $sp, $sp, -12
	sw $s0, 0($sp)
	sw $s1, 4($sp)
	sw $s2, 8($sp)
	move $t2, board
	#nested loop
	#basically a for i in list, for j in i, just like printing a list of lists
	line_loop:
	newline
	li $s1, 0
	beq $s0, 6, end_print

	print_loop:
	beq $s1, 18, loopback
	lw $s2, ($t2)

	beq $t2, start, restore_start #we restore the 'S'
	beq $t2, end, restore_end	#we restore the 'F'

	return_print:
	print_char($s2)
	addi $s1, $s1, 1
	addi $t2, $t2, 4
	j print_loop

loopback:
addi $s0, $s0,1
j line_loop

end_print:
lw $s2, 8($sp)
lw $s1, 4($sp)
lw $s0, 0($sp)
addi $sp, $sp, 12
jr $ra

restore_start:
li $s2, 83
j return_print

restore_end:
li $s2, 70
j return_print
```

# Test Cases

## #1.




Input:

S00000XXXXXX000000

0000XX000000000000

000XXX000000000000

000000000X00000000

00XXXX0000XXXX0000

00000000000000000F

Output:

S00000XXXXXX100000

0100XX010101010000

001XXX101010001000

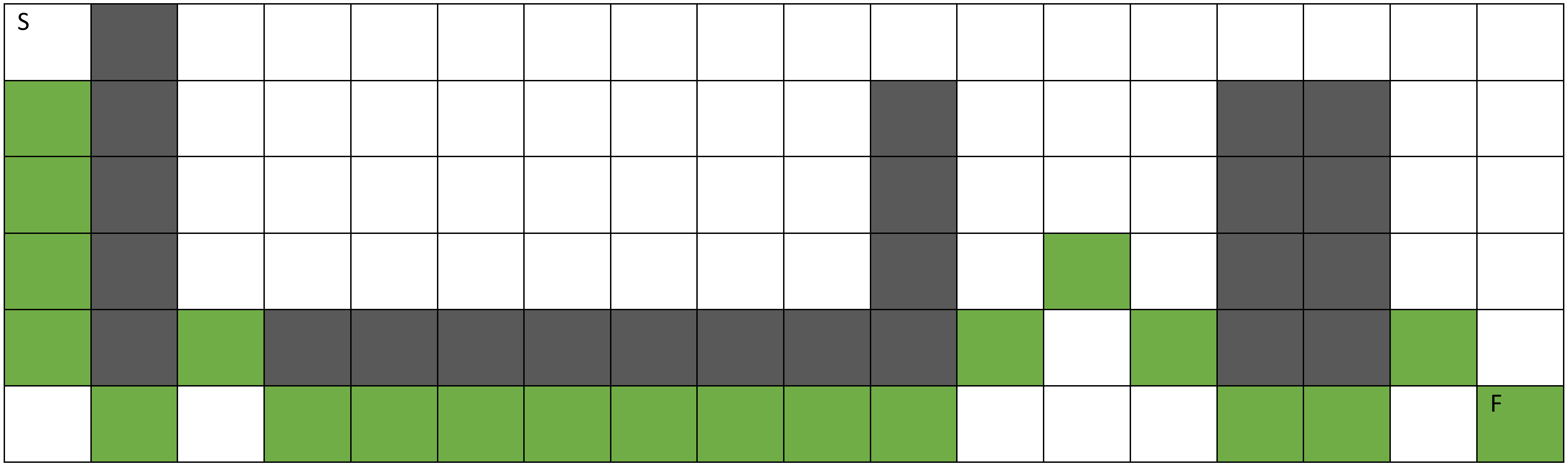
000111000X00000100

00XXXX0000XXXX0010

00000000000000000F

17

# #2.




Input:

SX0000000000000000

0X00000000X000XX00

0X00000000X000XX00

0X00000000X000XX00

0X0XXXXXXXX000XX00

00000000000000000F

Output:

SX0000000000000000

1X00000000X000XX00

1X00000000X000XX00

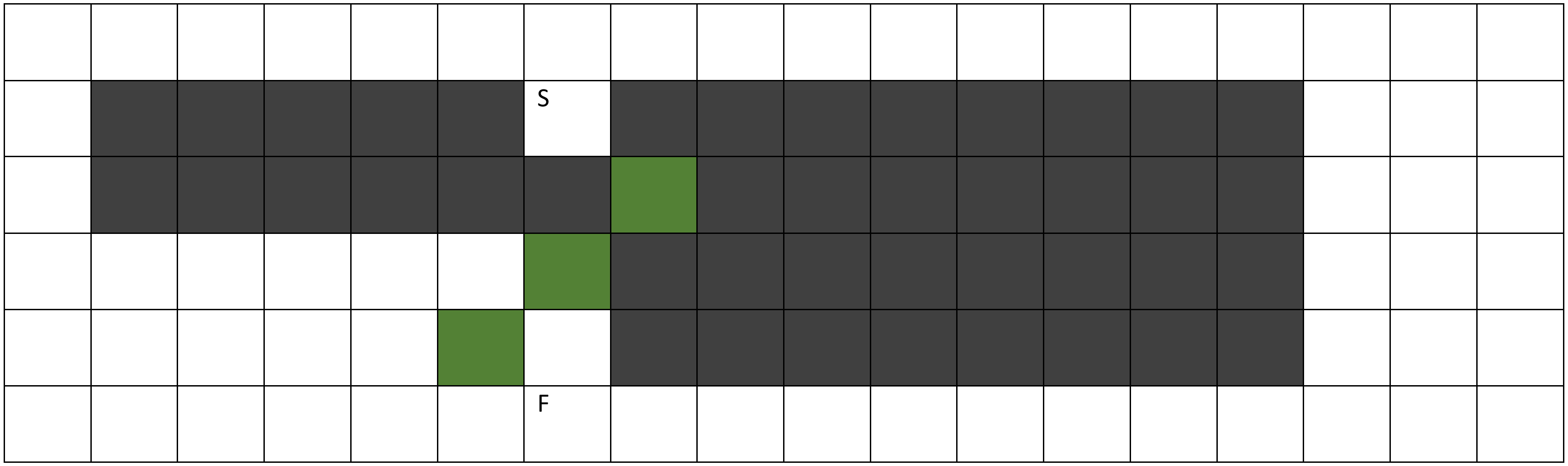
1X00000000X010XX00

1X1XXXXXXXX101XX10

01011111111000110F

21

# #3.




Input:

000000000000000000

0XXXXXSXXXXXXXX000

0XXXXXX0XXXXXXX000

0000000XXXXXXXX000

0000000XXXXXXXX000

000000F00000000000

Output:

000000000000000000

0XXXXXSXXXXXXXX000

0XXXXXX1XXXXXXX000

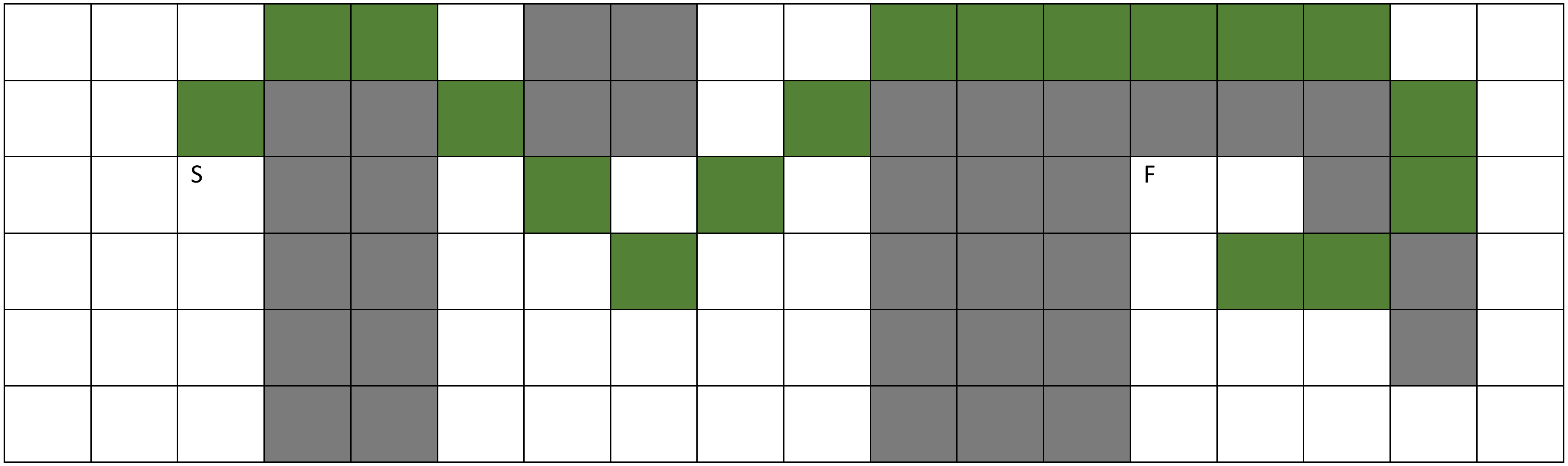
0000001XXXXXXXX000

0000010XXXXXXXX000

000000F00000000000

4

## #4.




Input:

000000XX0000000000

000XX0XX00XXXXXX00

00SXX00000XXXF0X00

000XX00000XXX000X0

000XX00000XXX000X0

000XX00000XXX00000

Output:

000110XX0011111100

001XX1XX01XXXXXX10

00SXX01010XXXF0X10

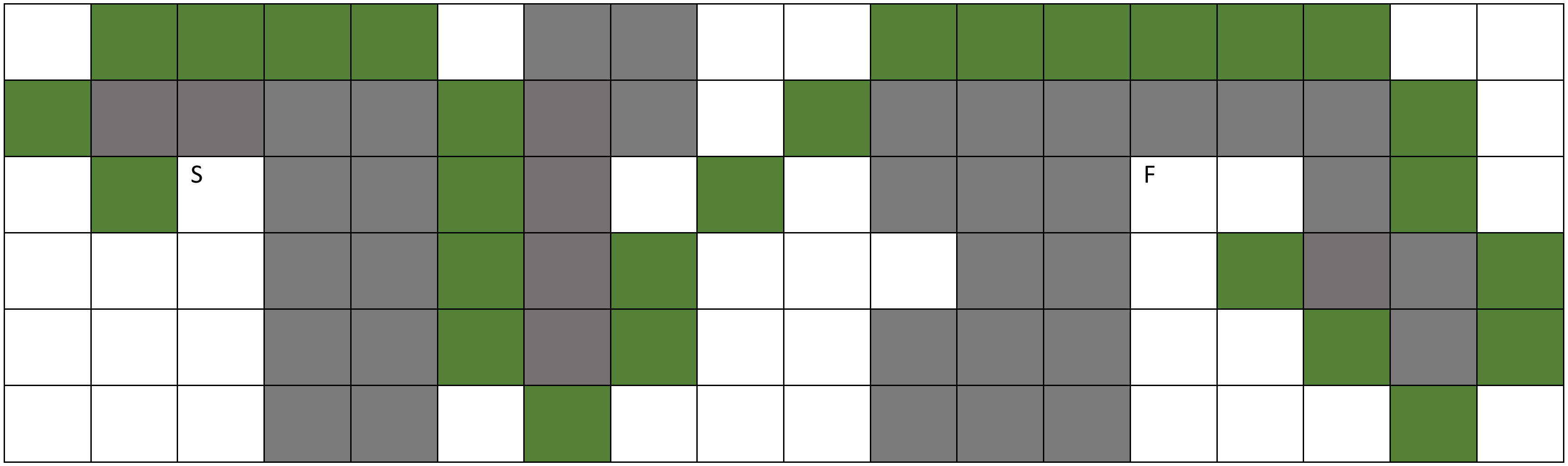
000XX00100XXX011X0

000XX00000XXX000X0

000XX00000XXX00000

19

# #5.




Input:

000000XX0000000000

0XXXX0XX00XXXXXX00

00SXX0X000XXXF0X00

000XX0X0000XX00XX0

000XX0X000XXX000X0

000XX00000XXX00000

Output:

011110XX0011111100

1XXXX1XX01XXXXXX10

01SXX1X010XXXF0X10

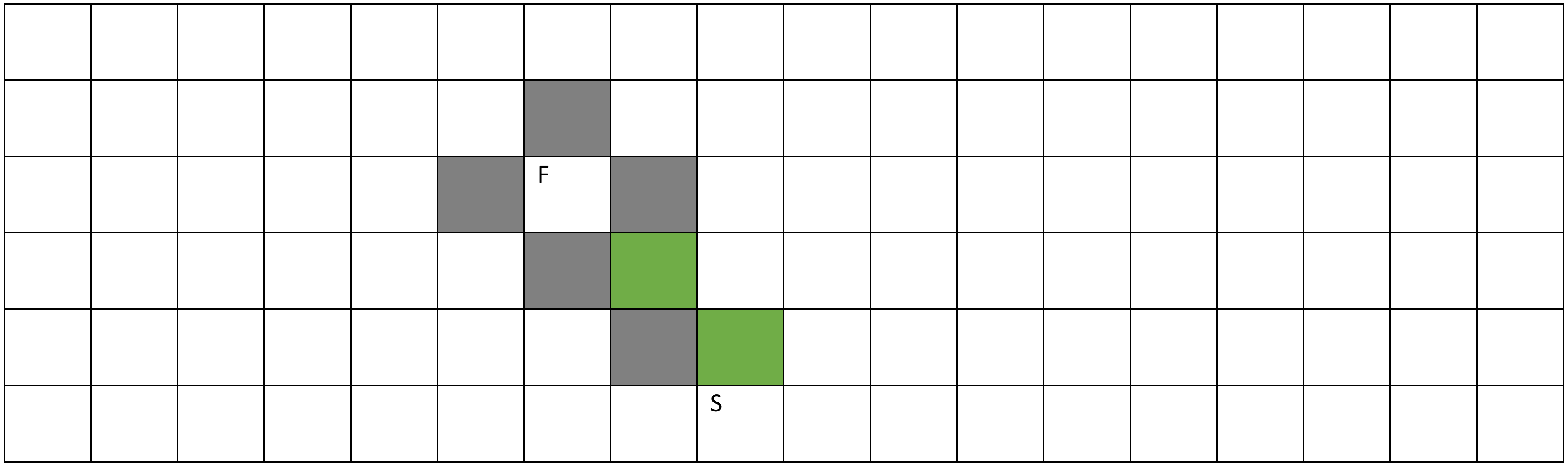
000XX1X1000XX01XX1

000XX1X100XXX001X1

000XX01000XXX00010

29

# #6.




Input:

000000000000000000

000000X00000000000

00000XFX0000000000

000000X00000000000

0000000X0000000000

00000000S000000000

Output:

000000000000000000

000000X00000000000

00000XFX0000000000

000000X10000000000

0000000X1000000000

00000000S000000000

3

**Python Code:**

```
board = []

row = []


for i in range(6):

  row1 = input().split()

  for j in range(18):

    row.append(row1[0][j])


  board.append(row)

  row = []


for i in range(6):

  for j in range(18):

    if board[i][j] == 'S':

      board[i][j] = '0'

      start = i,j


    if board[i][j] == 'F':

      end = i,j

      board[i][j] = '0'


bfs_board = [

  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

]


i,j = start

bfs_board[i][j] = 1



# we fill our bfs board, we can get the shortest distance from the start point to every traversable point in the board

#visit every traversable point

def pathing(k):

  for i in range(6):

    for j in range(18):

      if bfs_board[i][j] == k:

        def moveNE():

          if i>0 and j<17 and bfs_board[i-1][j+1] == 0 and board[i-1][j+1] == "0":

            bfs_board[i-1][j+1] = k + 1



        def moveNW():

          if i>0 and j>0 and bfs_board[i-1][j-1] == 0 and board[i-1][j-1] == "0":

            bfs_board[i-1][j-1] = k + 1



        def moveSE():

          if i<5 and j<17 and bfs_board[i+1][j+1] == 0 and board[i+1][j+1] == "0":

            bfs_board[i+1][j+1] = k + 1



        def moveSW():

          if i<5 and j>0 and bfs_board[i+1][j-1] == 0 and board[i+1][j-1] == "0":

            bfs_board[i+1][j-1] = k + 1


        def moveUp():

          if i>0 and bfs_board[i-1][j] == 0 and board[i-1][j] == "0":

            bfs_board[i-1][j] = k + 1


        def moveDown():

          if i<5 and bfs_board[i+1][j] == 0 and board[i+1][j] == "0":

            bfs_board[i+1][j] = k + 1


        def moveLeft():

          if j>0 and bfs_board[i][j-1] == 0 and board[i][j-1] == "0":

            bfs_board[i][j-1] = k + 1


        def moveRight():

          if j<17 and bfs_board[i][j+1] == 0 and board[i][j+1] == "0":

            bfs_board[i][j+1] = k + 1


        moveNE() #move ne

        moveNW() #move nw

        moveSE() #move se

        moveSW() #move sw

        moveUp() #move up

        moveDown()  #move down

        moveLeft() #move left
```

```
        moveRight() #move right




k = 0

while bfs_board[end[0]][end[1]] == 0:

    k += 1

    pathing(k)


shortest_path = k


i, j = end

k = bfs_board[i][j] #bfs_board[end]


#while we are not yet back to the start point, try to retrace our steps, while modifying the input board so that it will contain the solution

while k > 1:

  def checkNE():

    if i > 0 and j < 17 and bfs_board[i-1][j + 1] == k-1:

      return True

  def checkNW():

    if i > 0 and j > 0 and bfs_board[i-1][j - 1] == k-1:

      return True


  def checkSE():

     if i <5 and j < 17 and bfs_board[i+1][j + 1] == k-1:

      return True


  def checkSW():

   if i <5 and j >0 and bfs_board[i+1][j - 1] == k-1:

    return True


def checkUp():

  if i > 0 and bfs_board[i - 1][j] == k-1:

    return True


def checkDown():

  if i < 5 and bfs_board[i + 1][j] == k-1:

      return True


def checkLeft():

  if j > 0 and bfs_board[i][j - 1] == k-1:

      return True


def checkRight():

  if j < 17 and bfs_board[i][j + 1] == k-1:

      return True



#check ne

if checkNE():

  i,j = i-1, j+1

  board[i][j] = 1

  k-=1


#check nw

elif checkNW():

  i,j = i-1, j-1

  board[i][j] = 1

  k-=1


#check se

elif checkSE():

  i,j = i+1, j+1

  board[i][j] = 1

  k-=1


#check sw

elif checkSW():

  i,j = i+1, j-1

  board[i][j] = 1

  k-=1



#check up

elif checkUp():

  i, j = i-1, j

  board[i][j] = 1

  k-=1


#check down

elif checkDown():

  i, j = i+1, j

  board[i][j] = 1

  k-=1


#check left
```

```
elif checkLeft():
  i, j = i, j-1
  board[i][j] = 1
  k-=1
#check right
elif checkRight():
  i, j = i, j+1
    board[i][j] = 1
    k -= 1
print()

board[start[0]][start[1]] = "S"
board[end[0]][end[1]] = "F"
for i in board:
  for j in i:
    print(j, end="")
  print()
print(shortest_path)
```