ASSUMPTION UNIVERSITY

Vincent Mary School of Science and Technology

Department of Computer Science

SC 6310

DESIGN AND ANALYSIS OF ALGORITHMS

TERM PROJECT REPORT

1033 Labyrinth

Timus Online Judge

Submit to

Asst. Prof. Dr. Thitipong Tanprasert

by

5919449 Kiratijuta Bhumichitr

Semester 2/2016

OUTLINES

▪ Problem Definition

1033 Labyrinth (Difficulty: 230)

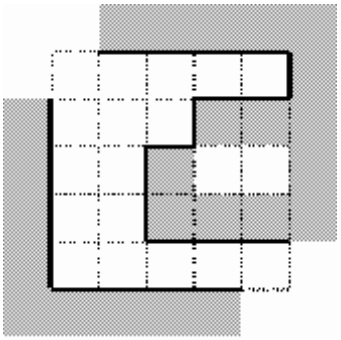
Vladimir Pinaev, Ural Collegiate Programming Contest '99

Time limit: 1.0 second
Memory limit: 64 MB

Administration of the labyrinth has decided to start a new season with new wallpapers. For this purpose, they need a program to calculate the surface area of the walls inside the labyrinth. This job is just for you!

The labyrinth is represented by a matrix $N \times N$ ($3 \le N \le 33$, you see, '3' is a magic digit!). Some matrix cells contain a dot character ('.') that denotes an empty square. Other cells contain a diesis character ('#') that denotes a square filled by monolith block of stone wall. All squares are of the same size 3×3 meters.

The walls are constructed around the labyrinth (except for the upper left and lower right corners, which are used as entrances) and on the cells with a diesis character. No other walls are constructed. There always will be a dot character at the upper left and lower right corner cells of the input matrix.



Your task is to calculate the area of visible part of the walls inside the labyrinth. In other words, the area of the walls' surface visible to a visitor of the labyrinth. Note that there's no holes to look or to move through between any two adjacent blocks of the wall. The blocks are considered to be adjacent if they touch each other in any corner. See picture: visible walls inside the labyrinth are drawn with bold lines. The height of all the walls is 3 meters.

Input

The first line of the input contains the single number N. The next N lines contain N characters each. Each line describes one row of the labyrinth matrix. In each line only dot and diesis characters will be used and each line will be terminated with a new line character. There will be no spaces in the input.

Output

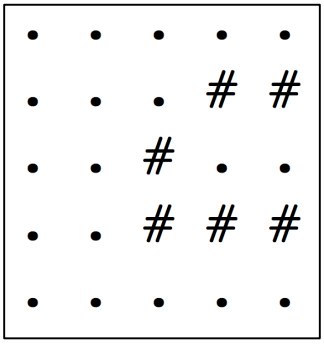
Your program should print to the output a single integer — the exact value of the area of the wallpaper needed.

Sample input & output

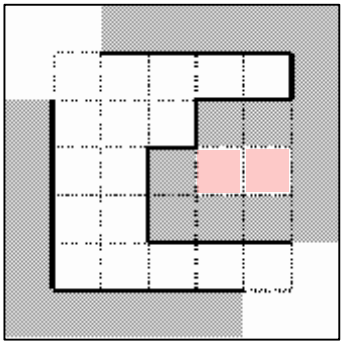
5 . . . . # # . . # . . . . # # #	198

▪ Problem Analysis

The labyrinth matrix size is limited to $N \times N$ where N is greater than or equal to 3 but less than or equal to 33. The labyrinth matrix cells can be either dot character – *empty square* or hash character – *wall*, see the figure (a). There always will be two entrances; upper-left and lower-right. The entrance cells can only be an empty square.

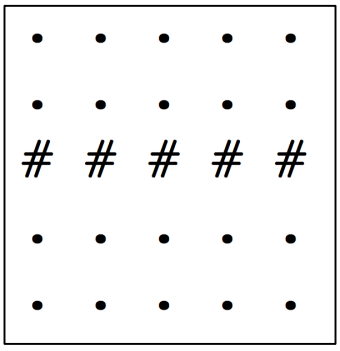


(a)



(b)

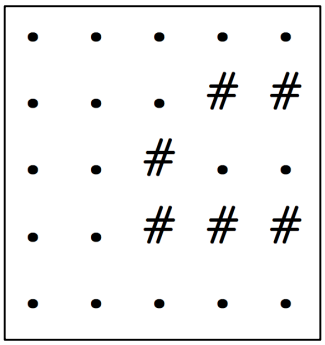
In order to calculate the area of the visible part of the walls, we basically need to walk through every possible empty square from the entrances and count the number of corresponding of the walls. Each wall is worth 9 wallpaper-unit. The labyrinth matrix might contain some unreachable area, see the figure (b) – the red squares. Also, the upper-left entrance may not have a reachable path to the lower-right entrance, see the figure (c). Meaning that, if we walk from the upper-left entrance and the lower-right entrance is not reachable, we have to walk from lower-right entrance for the possible wallpaper-unit.



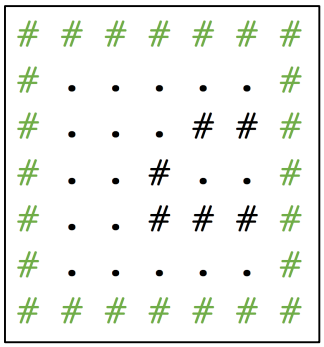
(c)

▪ Problem Solution

Programming language: Python 2.7
In order to easily count the number of corresponding wall, I will attach the 4-side wall as the figure (e) shown below;



(d)



(e)

Prepare the 2-dimensional array, size *(N+2)*×*(N+2)*, filled with hash character – *all wall*.

```
m = [['#']*(n+2) for i in range(n+2)]
```

Start from *i+1* till *N-1*, assign each input line with a specific range *1* and *N-1*.

```
for i in range(n)
    m[i+1][1:n+1] = list(raw_input())
```

There are 4-way to traverse from the current position; left, up, right, and down. The next position is valid for stepping in if it is not a wall.

```
path = [(-1,0), (0,1), (1,0), (0,-1)]
```

There are 2 entrances; upper-left and lower-right. In order to count every wall, I basically need to walk from the upper-left or lower-right, and note the number of wall that I found in each walk-step. I will use *Dijkstra*'s algorithm or a variant of it is known as **uniform-cost-search** to search from the entrance. The traversal pseudocode is given below.

```
1  Q ← empty-sequence
2  Q[source] ← (coordinate-x, coordinate-y)
3  while Q is not empty:
4      s ← Q[0]
5      for each i of [(-1,0), (0,1), (1,0), (0,-1)]:
6          new_x ← s[0]+i[0]
7          new_y ← s[1]+i[1]
8          add (new_x, new_y) to Q
9      remove s from Q
```

After applied the above algorithm, the below method is the complete *walkfrom* method. Each walk-step, I will mark it as visited step by *'P'* — line number 8. If the further step is a wall, I will basically increment *sm* variable — line number 14.

```
1  def walkfrom(x,y):
2      global sm
3      Q = []
4      Q.append((x,y))
5      while Q != []:
6          s = Q[0]
7          if m[s[0]][s[1]] == '.':
8              m[s[0]][s[1]] = 'P'
9              for i in range(4):
10                  x,y = s[0]+path[i][0],s[1]+path[i][1]
11                 if m[x][y] == '.':
12                     Q.append((x,y))
13                 elif m[x][y] == '#':
14                     sm += 1
15         del Q[0]
```

Since, there are 2 entrances; upper-left and lower-right. I need to make sure that I will not miss any available walls from the bottom part by checking the lower-right entrance whether I did mark the visited step or not.

```
sm = 0
walkfrom(1,1)
if m[n][n] == '.':
    walkfrom(n,n)
```

After all walk-step from *walkfrom* method, the result will be in *sm* variable. 1-sm value is equal to 1-wall found. I have to remove 4 walls from the 2 entrances as well. Each wall is then worth 9 wallpaper-unit.

```
print (sm-4)*9
```

The full source code is shown below.

```
n = input()
m = [['#']*(n+2) for i in range(n+2)]

for i in range(n):
    m[i+1][1:n+1] = list(raw_input())

path = [(-1,0), (0,1), (1,0), (0,-1)]

def walkfrom(x,y):
    global sm
    Q = []
    Q.append((x,y))
    while Q != []:
        s = Q[0]
        if m[s[0]][s[1]] == '.':
            m[s[0]][s[1]] = 'P'
            for i in range(4):
                x,y = s[0]+path[i][0],s[1]+path[i][1]
                if m[x][y] == '.':
                    Q.append((x,y))
                elif m[x][y] == '#':
                    sm += 1
        del Q[0]

sm = 0
walkfrom(1,1)
if m[n][n] == '.':
    walkfrom(n,n)
print (sm-4)*9
```

For time complexity, the worst-case performance is O(N^2).

The Timus online judge result.

ID	Date	Author	Problem	Language	Judgement result	Test #	Execution time	Memory used
7391527	20:07:02 18 May 2017	Code On	1033. Labyrinth	Python 2.7	Accepted		0.031	356 KB