Your task is to implement a simulation of a `change directory` command. This command changes the current working directory to the specified one.

The initial working directory is root i.e. `/`. You are given a list of `cd` commands `commands`.

There are multiple options for command arguments.

- `cd /` - changes the working directory to the root directory.
- `cd .` - stays in the current directory.
- `cd ..` - moves the working directory one level up. In the root directory, `cd ..` does nothing.
- `cd <subdirectory>` - moves to the specified subdirectory within the current working directory. `<subdirectory>` is a string consisting of only lowercase English letters.

All specified directories exist. Return the absolute path from the root to the working directory after executing all `cd` commands in the given order. `/` should be used as separators.

Note: You are not expected to provide the most optimal solution, but a solution with time complexity not worse than `O(commands.length² × max(commands[i].length))` *will fit within the execution time limit.*

i stack

i cd..
if stack empty, no change
else pop

```
def changeDir(commands):
path_stack = []

for c in commands:

arg = c[3:].strip()

if arg == "/":
    path_stack = []

elif arg == ".":
    continue

elif arg == "..":
    if path_stack:
        path.stack.pop()

else
    path_stack.push(arg)
```

In Python, `strip()` removes any leading and trailing whitespace characters (such as spaces, tabs, or newlines) from a string. For example, if you do `command.strip()`, it will return a version of `command` without any whitespace at the very start or end.

start with "/", and use "/" to seperate each dir

```
if path_stack :
    return "/" + "/".join(path_stack)
else :
    return "/"
```

Imagine you are playing a gravity-based puzzle game that involves clearing obstacles to allow an irregularly-shaped figure to fall to the bottom.

You are given a rectangular matrix `board` representing the game board, which only contains the following types of cells:

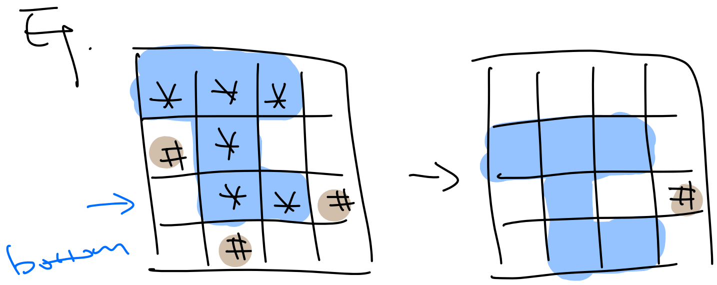
- `'-'` represents an empty cell,
- `'#'` represents an obstacle,
- `'*'` represents part of the figure.

It is guaranteed that the figure consists of one piece, where all parts are connected by the sides.

Your task is to simulate how the figure should fall, and find the minimum number of obstacles that should be removed to let the figure finally touch the bottom of the board with at least one of its cells.

Note: You are not expected to provide the most optimal solution, but a solution with time complexity not worse than `O(board.length · board[0].length)` *will fit within the execution time limit.*

Eg.



M[1][0], m[3][1] are removed.

1° find bottom row of the figure.

2° find shape of the figure,
each cell go down by #row - bottom
put the path cells of each cell into a set.

3° check # of obstacles in the set.

```
def removes(map):
    col = len(map)
    row = len(map[0])
    figure = []
    bottom = 0
    for i in range(row):
        for j in range(col):
            if map[i][j] == "*":
                figure.append((i,j))
                bottom = max(bottom, i)

    dist = row - bottom
    path = set(figure)
    for (i,j) in figure:
        for d in range(dist):
            path.add((i+d),j)
```

```
res = 0
for (i, j) in path:
    if map[i][j] == "#":
        res += 1

return res
```

GPTo1

```python
def min_obstacles_to_reach_bottom(board):
    rows = len(board)
    cols = len(board[0]) if rows > 0 else 0

    # 1. Collect the figure cells and find the top row of the figure
    figure_cells = []
    top = rows
    for r in range(rows):
        for c in range(cols):
            if board[r][c] == '*':
                figure_cells.append((r,c))
                top = min(top, r)

    # 2. Build "shape" in local coordinates r' = r - top
    shape = []
    for (r,c) in figure_cells:
        shape.append((r - top, c))
    # Height of bounding box
    h = max(rp for (rp,_) in shape) + 1

    # 3. We will track a running "union" of obstacle-coordinates we've “hit”
    #    as the figure goes from offset=top up to offset=i.
    #    cost[i] = size of that union if we stop at offset i.
    #    unionCoverageSet is a set of (row,col) obstacle cells encountered so far.
    union_coverage = set()
    cost = [0]*(rows+1)  # cost[i] means cost up to offset i, for i >= top

    # Initialize cost[top] by adding all obstacles for offset = top
    for (rp, cp) in shape:
        board_r = top + rp
        if 0 <= board_r < rows:
            # If there's an obstacle, add to the union
            if board[board_r][cp] == '#':
                union_coverage.add((board_r, cp))
    cost[top] = len(union_coverage)

    # Fill in cost for offsets top+1..rows-1 by “sliding” down row-by-row
    for i in range(top+1, rows):
        # When going from offset (i-1) to i:
        #   The shape’s new covered row is i-1 + h (the row that has just come into view),
        #   and the old top row (i-1) is “above” now—but we do NOT remove it
        #   from the union, because we need the total union of obstacles ever encountered.
        new_row = (i-1) + h  # row newly covered at offset i
        if new_row < rows:
            # Check each shape cell whose local row offset is h-1
            # but strictly you can just check *all* shape offsets that fall into new_row
            for (rp, cp) in shape:
                if rp == h-1:
                    board_r = new_row
                    if board[board_r][cp] == '#':
                        union_coverage.add((board_r, cp))

        cost[i] = len(union_coverage)

    # 4. Among all possible final offsets i that actually place
    #    at least one shape cell on bottom row, pick the min cost[i].
    #    Condition: i + rp = rows - 1 => i = (rows-1) - rp
    valid_offsets = set()
    for (rp, cp) in shape:
        bottom_offset = (rows - 1) - rp
        if bottom_offset >= top and bottom_offset < rows:
            valid_offsets.add(bottom_offset)

    answer = min(cost[i] for i in valid_offsets) if valid_offsets else 0
    return answer
```

Given an array of strings `words`, find the number of pairs where either the strings are equal or one string ends with another. In other words, find the number of such pairs `i`, `j` (`0 ≤ i < j < words.length`) that `words[i]` is a suffix of `words[j]`, or `words[j]` is a suffix of `words[i]`.

Example

- For `words = ["back", "backdoor", "gammon", "backgammon", "comeback", "come", "door"]`, the output should be `solution(words) = 3`.

  The relevant pairs are:

  1. `words[0] = "back"` and `words[4] = "comeback"`.
  2. `words[1] = "backdoor"` and `words[6] = "door"`.
  3. `words[2] = "gammon"` and `words[3] = "backgammon"`.

- For `words = ["cba", "a", "a", "b", "ba", "ca"]`, the output should be `solution(words) = 8`.

  The relevant pairs are:

  1. `words[0] = "cba"` and `words[1] = "a"`.
  2. `words[0] = "cba"` and `words[2] = "a"`.
  3. `words[0] = "cba"` and `words[4] = "ba"`.
  4. `words[1] = "a"` and `words[2] = "a"`.
  5. `words[1] = "a"` and `words[4] = "ba"`.
  6. `words[1] = "a"` and `words[5] = "ca"`