```python
#ADVENT OF CODE DAY 12 SOLUTION (BFS/GRAPH TRAVERSAL)
from collections import deque
#grid positions listed as y, x instead of x, y. This is the case until the very end.
queue = deque()
visited = {}
field = [] #will be a list of lists, with the inidces as coordinates, i.e. field[4][3] is y=4, x=3 (zero-indexed)
START_POINT = (0,0) #set these values later
END_POINT = (0,0)

def process():
    numOfLines = 41 #get this from an opening input() statement
    for count in range(numOfLines):
        line = input()
        field.append([operation(letter, count, inner) for inner, letter in enumerate(line)]) #where operation works on the character AND its position
    BFS()

def BFS():
    queue.append(END_POINT) #this searches backwards, finding a path from the end to the start.
                            #Change this to START_POINT to search forwards
    while (len(queue)) > 0:
        for spot in searchAround(queue[0]):
            visited[spot] = queue[0]
            queue.append(spot)
            if spot == START_POINT:
                traverse(spot)
        queue.popleft()

def traverse(start): #start is a y,x tuple. Since we searched backwards,
                     #we traverse forwards to build the final path
    currentLocation = start
    path = [start]
    while (currentLocation != END_POINT):
        currentLocation = visited[currentLocation]
        path.append(currentLocation)
    print([(spot[1], spot[0]) for spot in path]) #the path, in x/y coordinates
    print("Shortest path length:" + str(len(path)) + " nodes visited.") #subract one for number of steps taken
    exit()

def searchAround(centre): #centre is an x,y coordinate pair as a tuple
    GRID_HEIGHT = len(field)
    GRID_WIDTH = len(field[0])
    output = []
    if centre[0] > 0: #search to the left, if not on the left edge
        out = check_and_append(centre[0] - 1, centre[1], centre[0], centre[1]) #if condition() depends on the current centre, must also pass its position in
        if out is not None:
            output.append(out)
        #if centre[1] > 0: #if diagonals are allowed, these need to be a thing for all 4 diagonals, only one shown here (this checks down-right)
        #check_and_append(centre[0]- 1, centre[1] - 1)
    if centre[0] < GRID_HEIGHT - 1: #search to the right, if not on the right edge
        out = check_and_append(centre[0] + 1, centre[1], centre[0], centre[1])
        if out is not None:
            output.append(out)
    if centre[1] > 0:
        out = check_and_append(centre[0], centre[1] - 1, centre[0], centre[1])
        if out is not None:
            output.append(out)
    if centre[1] < GRID_WIDTH - 1:
        out = check_and_append(centre[0], centre[1] + 1, centre[0], centre[1])
        if out is not None:
            output.append(out)
    return output
```

```
def check_and_append(x, y, centrex, centrey):
    if (not visited.get((x,y))): #if we haven't visited (x,y) yet, consider this path/case
        if condition(x, y, centrex, centrey): #where condition() may depend on currently visitING x, y, or the original centre from which we came.
                        #condition should be true iff the spot we're checking could be part of the path.
            return (x, y)

def condition(y, x, centrey, centrex): #specific to the advent of code solution,
                                       #will need to change this to match our problem.
    return field[y][x] >= field[centrey][centrex] - 1
process()
#END ADVENT OF CODE SOLUTION

#DJIKSTRA'S ALGORITHM
def minimum(dicti):
    min_key = list(dicti.keys())[0]
    for i in list(dicti.keys)[1:]:
        if dicti[i] < dicti[min_key]:
            min_key = i
    return min_key

def dijkstra(airports, lines, start, end):
    unexplored = {airport : float('inf') for airport in airports}
    unexplored[start] = 0
    while len(unexplored) != 0:
        explore = minimum(unexplored)
        if explore == end:
            break
        else:
            for path in lines.items():
                if path[0][0] == explore:
                    if path[0][1] in unexplored.keys():
                        check_time = unexplored[path[0][0]] + path[1]
                        if check_time < unexplored[path[0][1]]:
                            unexplored[path[0][1]] = check_time
                elif path[0][1] == explore:
                    if path[0][0] in unexplored.keys():
                        check_time = unexplored[path[0][1]] + path[1]
                        if check_time < unexplored[path[0][0]]:
                            unexplored[path[0][0]] = check_time
            del unexplored[explore]
    return(unexplored[explore])

airports = ['A', 'B', 'C', 'D', 'E']
lines = {
    ('A', 'B') : 4,
    ('A', 'C') : 2,
    ('B', 'C') : 1,
    ('B', 'D') : 2,
    ('C', 'D') : 4,
    ('C', 'E') : 5,
    ('E', 'D') : 1,
}
start = 'A'
end = 'D'

print(dijkstra(airports, lines, start, end))
```

```
def two_lines_intersect(x1, y1, x2, y2, x3, y3, x4, y4): #where the first line goes from (x1, y1) to (x2, y2)
#and the second line goes from (x3, y3) to (x4, y4). works even if one line is vertical
    try:
        px= ( (x1*y2-y1*x2)*(x3-x4)-(x1-x2)*(x3*y4-y3*x4) ) / ( (x1-x2)*(y3-y4)-(y1-y2)*(x3-x4) ) #intersection X coordinate
        py= ( (x1*y2-y1*x2)*(y3-y4)-(y1-y2)*(x3*y4-y3*x4) ) / ( (x1-x2)*(y3-y4)-(y1-y2)*(x3-x4) ) #intersection Y coordinate
        return (x1 < px < x2) and (x3 < px < x4), (px, py) #answer[0] is "do they intersect within bounds",
        #answer[1] is "where do they intersect", even if it's out of bounds.
    except ZeroDivisionError: #lines have the same slope (no single intersection)
        return False, False #though they may be the same line

def prime_check(n):
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

def left_predicate(x1, y1, x2, y2, px, py): #where the line is x1 -> x2 and the point is at (px, py)
    return (x2 - x1) * (py - y1) - (y2 - y1)*(px - x1) > 0

#####GRAHAM's SCAN ALGORITHM#####
# A Python3 program to find convex hull of a set of points. Refer
# https://www.geeksforgeeks.org/orientation-3-ordered-points/
# for explanation of orientation()

from functools import cmp_to_key

# A class used to store the x and y coordinates of points
class Point:
    def __init__(self, x = None, y = None):
        self.x = x
        self.y = y

# A global point needed for sorting points with reference
# to the first point
p0 = Point(0, 0)

# A utility function to find next to top in a stack
def nextToTop(S):
    return S[-2]

# A utility function to return square of distance
# between p1 and p2
def distSq(p1, p2):
    return ((p1.x - p2.x) * (p1.x - p2.x) +
            (p1.y - p2.y) * (p1.y - p2.y))

# To find orientation of ordered triplet (p, q, r).
# The function returns following values
# 0 --> p, q and r are collinear
# 1 --> Clockwise
# 2 --> Counterclockwise
def orientation(p, q, r):
    val = ((q.y - p.y) * (r.x - q.x) -
           (q.x - p.x) * (r.y - q.y))
    if val == 0:
        return 0  # collinear
    elif val > 0:
        return 1  # clock wise
    else:
        return 2  # counterclock wise
```

```python

# A function used by cmp_to_key function to sort an array of
# points with respect to the first point
def compare(p1, p2):

    # Find orientation
    o = orientation(p0, p1, p2)
    if o == 0:
        if distSq(p0, p2) >= distSq(p0, p1):
            return -1
        else:
            return 1
    else:
        if o == 2:
            return -1
        else:
            return 1

# Prints convex hull of a set of n points.
def convexHull(points, n):

    # Find the bottommost point
    ymin = points[0].y
    min = 0
    for i in range(1, n):
        y = points[i].y

        # Pick the bottom-most or choose the left
        # most point in case of tie
        if ((y < ymin) or
            (ymin == y and points[i].x < points[min].x)):
            ymin = points[i].y
            min = i

    # Place the bottom-most point at first position
    points[0], points[min] = points[min], points[0]

    # Sort n-1 points with respect to the first point.
    # A point p1 comes before p2 in sorted output if p2
    # has larger polar angle (in counterclockwise
    # direction) than p1
    p0 = points[0]
    points = sorted(points, key=cmp_to_key(compare))

    # If two or more points make same angle with p0,
    # Remove all but the one that is farthest from p0
    # Remember that, in above sorting, our criteria was
    # to keep the farthest point at the end when more than
    # one points have same angle.
    m = 1  # Initialize size of modified array
    for i in range(1, n):

        # Keep removing i while angle of i and i+1 is same
        # with respect to p0
        while ((i < n - 1) and
        (orientation(p0, points[i], points[i + 1]) == 0)):
            i += 1

        points[m] = points[i]
        m += 1  # Update size of modified array

    # If modified array of points has less than 3 points,
    # convex hull is not possible
    if m < 3:
        return

    # Create an empty stack and push first three points
```

```
    # to it.
    S = []
    S.append(points[0])
    S.append(points[1])
    S.append(points[2])

    # Process remaining n-3 points
    for i in range(3, m):

        # Keep removing top while the angle formed by
        # points next-to-top, top, and points[i] makes
        # a non-left turn
        while ((len(S) > 1) and
        (orientation(nextToTop(S), S[-1], points[i]) != 2)):
            S.pop()
        S.append(points[i])

    # Now stack has the output points,
    # print contents of stack
    while S:
        p = S[-1]
        print("(" + str(p.x) + ", " + str(p.y) + ")")
        S.pop()

# Driver Code
input_points = [(0, 3), (1, 1), (2, 2), (4, 4),
                (0, 0), (1, 2), (3, 1), (3, 3)]
points = []
for point in input_points:
    points.append(Point(point[0], point[1]))
n = len(points)
convexHull(points, n)
#####END GRAHAM SCAN ALGORITHM

### Making and traversing trees
class GenericTreeNode(object):
    def __init__(self, children, measurement): #where measurement is something we need
    to keep track of like fun score
        self.children = children
        self.measurement = measurement
    def getChildren():
        return self.children
    def getGrandchildren():
        return [child.getChildren() for child in children]
#for a question like the CEO question where we;re given the parent,
#put them in a list, and have a list of tree nodes, it's not ideal but what else do you
do?

###Prime factorization of any integer
import math
def primeFactors(n):
    while n % 2 == 0:
        print (2)
        n = n // 2
    for i in range(3,int(math.sqrt(n))+1,2):
        while n % i== 0:
            print(i),
            n = n // i
    if n > 2:
        print(n)


def strange_input_example():
#for when the problem diesn't say how many cases/lines there are
    for line in sys.stdin:
        try:
            process(line)
```

```python

#finds subsets of an array that add to a number N
def get_subsets_adding_to_n(array, n):
    dp = [1] + [0]*n
    curr = 0
    for i in range(0, len(array)):
        curr += array[i]
        for  j in range(min(n, curr), array[i]-1, -1):
            dp[j] += dp[j - array[i]]
    return dp[-1]
#gets the optimal number of cuts to turn a rectangle into squares
#note: this won't be fast enough for large numbers (over 100 or so)
#unless they share some factors, but it's the best we got
memo = {}
def optimal_rectangle_cut(i,j):
    import math
    #base cases: o-width line or already a square
    if (i == j) or (i <= 0) or (j <= 0):
        return 0
    gcd = math.gcd(i,j)
    width = max(i,j) // gcd
    height = min(i,j) // gcd
    if (height == 1): #remember that we just took the gcd
        return width - 1
    if (height, width) in memo:
        return memo[(height, width)]
    hcut = 1 + min([optimal_rectangle_cut(width, count) + optimal_rectangle_cut(width, height - count) for count in range(1, height // 2 + 1)])
    vcut = 1 + min([optimal_rectangle_cut(count, height) + optimal_rectangle_cut(width - count, height) for count in range(1, width // 2 + 1)])
    memo[(height, width)] = min(hcut, vcut)
    return memo[(height, width)]

#rainwater problem (dual-pointer example)
hs = [int(i) for i in input().split(sep=" ")]
#find the highest point in the map
def find_highest(lo=0, hi=len(hs)):
    maxh = 0
    maxh_point = 0
    for count in range(lo, hi):
        if hs[count] > maxh:
            maxh = hs[count]
            maxh_point = count
    return maxh, maxh_point

score = 0
high_left, highpoint_left = hs[0], 0
high_right, highpoint_right = find_highest(0, len(hs))
for count in range(len(hs)):
    if hs[count] > high_left: #are we higher than any point to our left? if so, update the left side wall
        high_left, highpoint_left = hs[count], count
    if count == highpoint_right: #is this the highest point to the right? if so, find a new high point to the right of the current position
        high_right, highpoint_right = find_highest(lo=count+1)
    #those two should both happen quite often, if we crest the highest point on the map we'll need to set that as the left wall and find a new right wall
    score += max(min(high_left, high_right) - hs[count], 0)
    print(max(min(high_left, high_right) - hs[count], 0))
print(score)
```

```
#where W is the max weight of the backpack, wt is an array of weights, and val is an array of values
def knapSack(W, wt, val):
    n=len(val)
    table = [[0 for x in range(W + 1)] for x in range(n + 1)]

    for i in range(n + 1):
        for j in range(W + 1):
            if i == 0 or j == 0:
                table[i][j] = 0
            elif wt[i-1] <= j:
                table[i][j] = max(val[i-1]
+ table[i-1][j-wt[i-1]],  table[i-1][j])
            else:
                table[i][j] = table[i-1][j]
    return table[n][W]

# Function to Check if a substring is a palindrome
def is_palindrome(string, i, j):

    while i < j:
        if string[i] != string[j]:
            return False
        i += 1
        j -= 1
    return True

#Function to find the minimum number of cuts needed for palindrome partitioning
def min_pal_partition(string, i, j):
    # Base case: If the substring is empty or a palindrome, no cuts needed
    if i >= j or is_palindrome(string, i, j):
        return 0
    ans = 10**70 #absurdly high number
    # Iterate through all possible partitions and find the minimum cuts needed
    for k in range(i, j):
        count = min_pal_partition(string, i, k) + \
            min_pal_partition(string, k + 1, j) + 1
        ans = min(ans, count)
    return ans
```