***Fill The Cube***

Problem Description  
  
A company manufactures walls which can be directly implanted at the site. The company uses small square bricks of material C and material D which have similar looks but have huge difference in quality. The company manufactures walls of square shapes only to optimize their costs.  
  
A novice employee created a square wall using bricks of material C and D. However, the client had asked the wall to be made of only high-quality material - material C.  
  
To solve this problem, they will place the wall in a special furnace and heat it such that the material D melts and only material C remains. Material C brick will move down due to gravity if a material D brick below it melts. The new empty space created will be filled by new material C square walls. They also want to use biggest possible C square wall while building the final wall. For this they will position the wall in the furnace in an optimal way i.e. rotate by 90-degrees any number of times, if required, such that the biggest space possible for new material C wall is created. No rotations are possible when the furnace starts heating.  
  
Given the structure of the original wall created by the novice employee, you need to find out the size of the new C square wall which can be fitted in the final wall which will be delivered to the client.  
  
Constraints  
  
1 < N < 100  
  
Input  
  
First Line will provide the size of the original wall N.  
  
Next N lines will provide the type of material (C and D) used for each brick by the novice employee.  
  
Output  
  
Size of the biggest possible C square wall which can be fitted in the final wall.  
  
Time Limit  
  
1  
  
  
  
Examples  
  
Example 1  
  
Input  
  
4  
  
C D C D  
  
C C D C  
  
D D D D  
  
C D D D  
  
Output  
  
3  
  
Explanation  
  
If the wall is placed with its left side at the bottom, space for a new C wall of size 2x2 can be created. This can be visualized as follows  
  
D C D D  
  
C D D D  
  
D C D D  
  
C C D C  
  
The melted bricks can be visualized as follows  
  
- - - -  
  
- C - -  
  
C C - -  
  
C C - C  
  
Hence, the maximum wall size that can be replaced is 2x2.  
  
If the wall is placed as it is with its original bottom side at the bottom, space for a new C wall of size 3x3 can be created. Post melting, this can be visualized as follows.  
  
- - - -  
  
C - - -  
  
C - - -  
  
C C C C  
  
Hence, the maximum wall size that can be replaced is 3x3 in this approach.  
  
Since no rotations followed by heating is going to a yield a space greater than 3x3, the output is 3.  
  
Example 2  
  
Input  
  
7  
  
C D D C D D D  
  
C D D C D D D  
  
D D D D D D C  
  
D C D C D D D  
  
D D D C D C D  
  
C D D C D C C  
  
C D C D C C C  
  
Output  
  
5  
  
Explanation  
  
If the wall is placed with its left side at the bottom, a space for new C wall of size 5x5 can be created. This can be visualized as follows  
  
D D C D D C C  
  
D D D D C C C  
  
D D D D D D C  
  
C C D C C C D  
  
D D D D D D C  
  
D D D C D D D  
  
C C D D D C C  
  
When this orientation of the wall is heated, a space for new C wall of size 5x5 is created after the D bricks melt  
  
\_ \_ \_ \_ \_ \_ \_  
  
\_ \_ \_ \_ \_ \_ \_  
  
\_ \_ \_ \_ \_ \_C  
  
\_ \_ \_ \_ \_ \_ C  
  
\_ \_ \_ \_ \_ C C  
  
C C \_ C C C C  
  
C C C C C C C  
  
Whereas, if the rotation was not done, the wall formed after the D bricks melt will be as follows  
  
\_ \_ \_ \_ \_ \_ \_  
  
\_ \_ \_ \_ \_ \_ \_  
  
\_ \_ \_ C \_ \_ \_  
  
C \_ \_ C \_ \_ \_  
  
C \_ \_ C \_ \_ C  
  
C \_ \_ C \_ C C  
  
C C C C C C C  
  
When this orientation of the wall is heated, a space for new C wall of size 3x3 only is created after the D bricks melt  
  
Hence rotation is important and correct answer is 5x5  
  
Since no rotations followed by heating is going to a yield a space greater than 5x5, the output is 5.

import java.util.\*;

class Cube{

// anti-clockwise direction

public void rotateMatrix( int N, int mat[][])

{

// Consider all squares one by one

for (int x = 0; x < N / 2; x++) {

// Consider elements in group

// of 4 in current square

for (int y = x; y < N - x - 1; y++) {

// Store current cell in

// temp variable

int temp = mat[x][y];

// Move values from right to top

mat[x][y] = mat[y][N - 1 - x];

// Move values from bottom to right

mat[y][N - 1 - x]

= mat[N - 1 - x][N - 1 - y];

// Move values from left to bottom

mat[N - 1 - x][N - 1 - y] = mat[N - 1 - y][x];

// Assign temp to left

mat[N - 1 - y][x] = temp;

}

}

}

//For melting of matrix

/\* Melting is done by taking the first column

\* Then two variables top and down

\* top point to the first element of the first column

\* down points to the last element of the first column

\* top and down moves towards each other by iterating the elements in the first column from their position

\* if top founds a 'D' (ie) zero in our case, then it'll change it to -1 to denote space and move to next element

\* if top founds a 'C', top will get assigned to C and wait for down to find a 'D'

\* if down finds a 'D' iterating from the bottom , then down will change the 'D' to 'C' and top will change it's 'C' to space (ie) '-1'

\* and then top will continue to search for another C while continuously changing 'D' to space

\* Thus the first column will be melted and it'll be repeated for the remaining columns

\*/

public void melting(int N, int mat[][]) {

for(int i=0;i<N;i++) {

int top=0;

int down=N-1;

for(int loop=0;loop<N;loop++) {

if(top>down) {

break;

}

for(;top<N;top++) {

if(mat[top][i]==0) {

mat[top][i]=-1; //To denote the space as -1 after C drops

}

else if(mat[top][i]==1) {

break;

}

}

for(;down>0;down--) {

if(mat[down][i]==0) {

mat[down][i]=1; //changes down to 1 and top to -1 to make the matrix look melted

mat[top][i]=-1; //To denote the space as -1 after C drops

top++;

down--;

break;

}

}

}

}

}

/\*For finding the square wall to be fit inside the melted matrix

\* We'll create a 1D array of size N which is the wall\_size given as input

\* Assign all the elements of array from 1 to the input N ex: arr[0]=1, arr[1] = 2 ......arr[N-1]=N

\* We'll use each element of the array as the square wall size to fit into our melted matrix

\* and assign -1 to the respective array[i] if it fits inside our matrix

\* Atlast we'll iterate our fully checked array from the last index arr[N] and if arr[i] matches -1

\* then we'll return the correspinding array element position as the biggest square to be fitted to our matrix

\* To check if a square fits our matrix, we'll take the first column and iterate from top

\* We'll continue our iteration and count the no.of iterations of spaces in our first column

\* If we find a C instead of a space then we'll stop the iteration and check if no.of iteration == array[i]

\* If it satisfies then we'll increment a new variable called loop and check whether loop == array[i]

\* If it equals then we've successfully fitted a square and assign array[i]=-1 and move on the next element in our array

\* if not then we'll move on to next column till loop==array[i]

\* if no.of iterations < array[i] then we'll move to next column while setting loop back to 0 and check from first

\* whether we can fit a square. If not we'll not assign -1 to our array.

\*/

public int wall\_size(int N,int mat[][]){

int[] arr = new int[N];

for(int i=0;i<N;i++) {

arr[i]=(i+1);

}

for(int i=0;i<arr.length;i++) {

int loop=0;

for(int col=0;col<N;col++) {

int count =0;

for(int row=0;row<N;row++ ) {

if(count==arr[i]) {

break;

}

if(mat[row][col]==-1) {

count++;

//For Debugging

//System.out.println("column: "+col +"Row: "+row);

}

else if(mat[row][col]==1)

{

//For Debugging

//System.out.println("same");

break;

}

}

if(count<arr[i]) {

//For Debugging

//System.out.println("COUNT:"+count);

loop=0;

}

if(count==arr[i]) {

loop++;

if(loop==arr[i]) {

//For Debugging

//System.out.printf("\n Array: %d , column: %d, loop: %d, count: %d \n",arr[i],col,loop,count);

arr[i]=-1;

break;

}

}

}

}

for(int i=(arr.length)-1;i>=0;i--) {

if(arr[i]==-1) {

return (i+1);

}

}

return 0;

}

//for debugging printing of the matrix

public void print(int size,int[][] mat) {

System.out.println();

for(int i =0;i<size;i++ ) {

for ( int j=0;j<size;j++) {

System.out.print(" "+mat[i][j]+" ");

}

System.out.println();

}

}

}

//Main class

public class Cube\_Fill {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

int wall\_size = input.nextInt();

input.nextLine();

String[] bricks = new String[wall\_size];

int[][] bmatrix = new int[wall\_size][wall\_size];

int[][] bmatrix1 = new int[wall\_size][wall\_size];

//For converting the input to matrix form C=1 and D=0

for(int i=0;i<bricks.length;i++) {

bricks[i] = input.nextLine();

for(int j=0;j<bricks.length;j++) {

if(bricks[i].charAt(j)=='C')

bmatrix[i][j] = 1;

else

bmatrix[i][j]=0;

}

}

Cube cubObj = new Cube();

int[][] k = new int[2][2];

k[0][1]=1;

int[][] j = new int[2][2];

for(int i=0;i<k.length;i++) {

System.arraycopy(k[i] , 0, j[i], 0, k[0].length);

}

j[0][1] = 0;

System.out.println(k[0][1]);

//Copying bmatrix for rotation and storing it in another bmatrix1

for(int i=0;i<bmatrix.length;i++) {

System.arraycopy(bmatrix[i], 0, bmatrix1[i], 0, bmatrix[0].length);

}

cubObj.rotateMatrix(wall\_size, bmatrix1); //rotation happens

//cubObj.print(wall\_size, bmatrix); -- Print the matrix form of the input

//Melting

cubObj.melting(wall\_size, bmatrix);

cubObj.melting(wall\_size,bmatrix1);

//cubObj.print(wall\_size, bmatrix); -- Print the melted matrix

int notRotated = cubObj.wall\_size(wall\_size, bmatrix); //Identifies the wall\_size

int rotated = cubObj.wall\_size(wall\_size, bmatrix1); //Identifies the wall\_size of the rotated matrix

//To check the wall\_sizes and print the bigger one.

if(notRotated>=rotated) {

System.out.println(notRotated);

}

else {

System.out.println(rotated);

}

}

}