The Grid Search editorial

Problem's link: https://www.hackerrank.com/challenges/the-grid-search/problem
There are two approaches demonstrated here:

- The brute force approach
- The dynamic programming approach

The brute force approach

Running the larger matrix G element by element, and check if there is a match for the smaller matrix P (using that element at the top left of both matrices)

example

$$G = \begin{bmatrix} 123412 \\ 561212 \\ 123634 \\ 781288 \end{bmatrix} \quad P = \begin{bmatrix} 12 \\ 34 \end{bmatrix}$$

below the behavior of the brute force approach:

123412 561212 123634 781288	1**2**3412 561212 123634 781288	12**3**412 561212 123634 781288	123**4**12 561212 123634 781288	1234**1**2 561212 123634 781288	123412 **5**61212 123634 781288	123412 5**6**1212 123634 781288	123412 561**2**12 123634 781288
No match	No match	No match	No match	No match	No match	No match	No match
123412 5612**1**2 123634 781288							
match							

C++ code of the function ″grid_search
Time complexity: O(R*C*r*c)

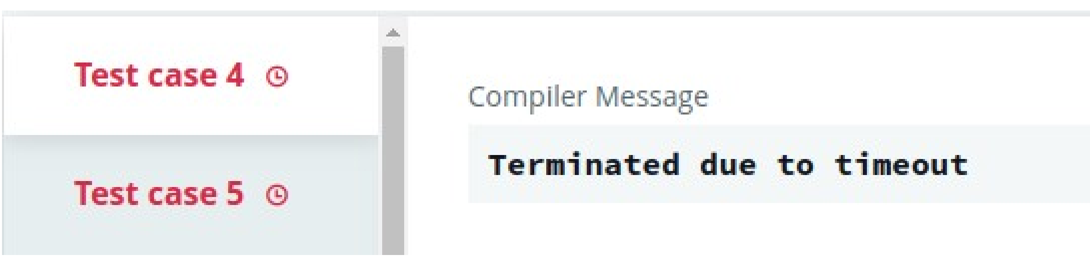
```cpp
bool check_match (vector<string> G, vector<string> P, int r, int c, int rg, int cg){
  for (int rp = 0 ; rp < r ; ++rp)
    for (int cp = 0 ; cp < c ; ++cp)
      if (P[rp][cp] != G[rg + rp][cg + cp]) return false;
  return true;
}

string grid_search(vector<string> G, vector<string> P, int R, int C, int r, int c) {
  for (int rg = 0 ; rg < R-r+1 ; ++rg)
    for (int cg = 0 ; cg < C-c+1 ; ++cg)
      if (check_match(G, P, r, c, rg, cg)) return "YES";

  return "NO";
}
```

Whole C++ code

```cpp
/*
* Brute force approach: O(t * R*C*r*c)
* Terminated due to timeout
*/

#include <bits/stdc++.h>
using namespace std;

bool check_match (vector<string> G, vector<string> P, int r, int c, int rg, int cg)
{
  for (int rp = 0 ; rp < r ; ++rp)
    for (int cp = 0 ; cp < c ; ++cp)
      if (P[rp][cp] != G[rg + rp][cg + cp]) return false;
  return true;
}

string grid_search(vector<string> G, vector<string> P, int R, int C, int r, int c)
{
  for (int rg = 0 ; rg < R-r+1 ; ++rg)
    for (int cg = 0 ; cg < C-c+1 ; ++cg)
      if (check_match(G, P, r, c, rg, cg)) return "YES";

  return "NO";
}
```

```
int main(){
  int t;
  cin >> t;
  cin.ignore(numeric_limits<streamsize>::max(), '\n');

  for (int t_itr = 0; t_itr < t; t_itr++) {
    int R, C;
    cin >> R >> C;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    vector<string> G(R);
    for (int i = 0; i < R; i++) {
      string G_item;
      getline(cin, G_item);
      G[i] = G_item;
    }

    int r, c;
    cin >> r >> c;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    vector<string> P(r);
    for (int i = 0; i < r; i++) {
      string P_item;
      getline(cin, P_item);
      P[i] = P_item;
    }

    string result = grid_search(G, P, R, C, r, c);
    cout << result << "\n";
  }

  return 0;
}
```

With this code, we got a timeout error (our program run too slow for tests cases #4 and #5)



Dynamic programming approach

The idea is to create a $r \times R$ matrix of pairs of {Boolean, array of all positions of each line of the smaller grid P, in each line of the greater grid G }.

The rows of the matrix represent the lines of P.

The columns of the matrix represent the lines of G.

	0	1	2	j	….	R−1
0	False,	True,	..	..	..	
i	...	...	True,	..	..	..
...	..	..	..	True,	..	..
r−1	..	..	..	..	True,	..

If a line i of P exists once (or more than once) in a line j of G, than the value of the matrix at cell (i, j) will be { $true$, $all\,the\,positions\,of\,line\,i$ in j).

To know if there is a match of P in G, we check if all lines of P exists consecutively in each line of G.

So, for each diagonal of the $r \times R$ matrix:

- If there is a cell with a $false$ value (it means, the current line of P doesn’t exist in the current line of G), so, no need to continue with that diagonal.
- Otherwise, compute the number of occurrence of each position in that diagonal.
- At the end, check the number of occurrence of each position:
  - if it’s equal to the number of P ’s lines, return "YES"

example

$$\boldsymbol{G}=\begin{bmatrix}123412\\561212\\123634\\781288\end{bmatrix} \quad \boldsymbol{P}=\begin{bmatrix}12\\34\end{bmatrix}$$

12 is in positions:

- 0 and 4 of line #1 of G
- 2 and 4 of line #2 of G
- 0 line #3 of G
- 2 line #4 of G

34 is in position:

- 2 line #1 of G
- 4 line #3 of G

`dp:`

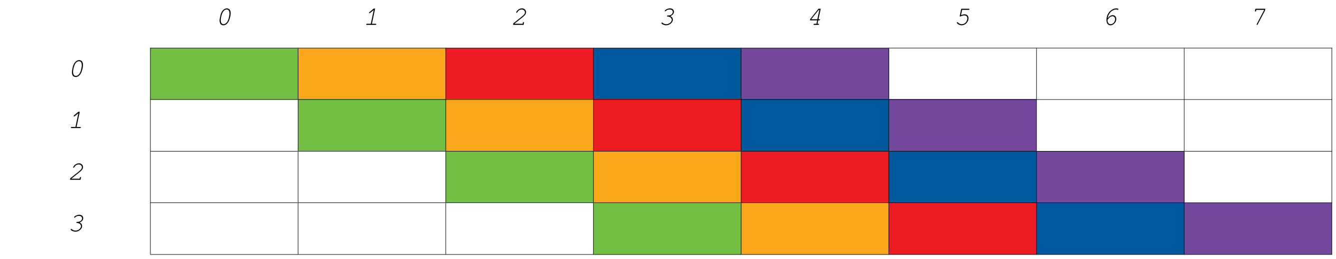
	0	1	2	3
0	true, 0 4	true, 2 4	true, 0	true, 2
1	true, 2	false,	true, 4	false,

For the 1st diagonal $(0,0)(1,1)$, the line 34 doesn’t exist in line 561212, so no need to check.
For the 2nd diagonal $(0,1)(1,2)$:

- the line 12 exists in line 561212 at positions 2 and 4
- the line 34 exists in line 123634 at position 4
- as you see, the occurrence of position 4 is 2 for this diagonal, which is the number of line of P. **we can conclude that P exists in G**.

At the beginning, we have to find the number of all diagonals of the $r \times R$ gird. We mean by all diagonals, the ones those contain all P’s lines.

For example, an 4×8 matrix has 5 diagonals



Diagonal #1 (green)

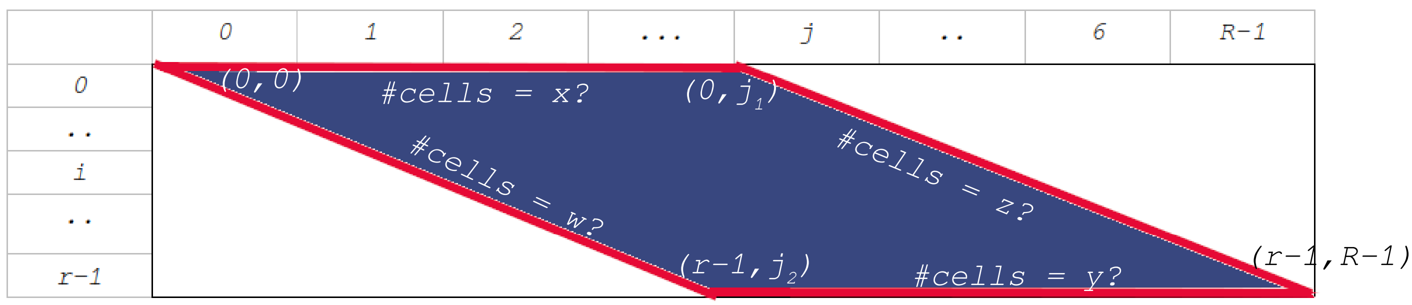
Diagonal #2 (gold)

Diagonal #3 (red)

Diagonal #4 (blue)

Diagonal #5 (purple)

In general, for a $r\times R$ grid, the number of diagonals is the number of cells (x or y) by columns:



We know the coordinates of the four points constructing the shape, so, it's easy to find w and z .

$w=r-1-0+1=r=j_2-0+1$

$z=r-1-0+1=r=R-1-j_1+1$

we have to find the values of x and y . also, we must to prove that $x=y$ s.

let's find the value of x :

$x=j_1-0+1=j_1+1$

we have $r=R-j_1$ <=> $j_1=R-r$

so, $x=R-r+1$

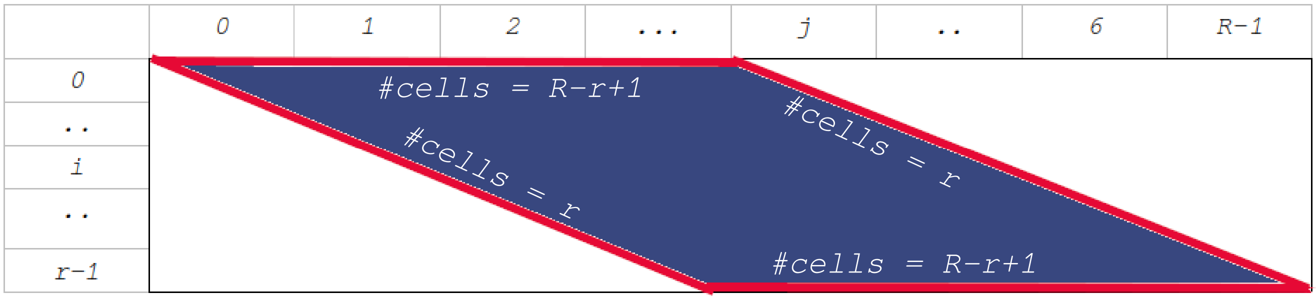
let's do the same for y :

$y=R-1-j_2+1$

we have $r=j_2+1$ <=> $j_2=r-1$

so, $y=R-1-r+1+1$ <=> $y=R-r+1$

we proved that $x=y$, **so the #diagonals of the** $r\times R$ **matrix is** $R-r+1$


0
1
2
...
j
..
6
R−1
0
..
i
..
r−1
#cells = R−r+1
#cells = r
#cells = r
#cells = R−r+1

C++ code of the function ″grid_search
Time complexity: O(R*C*r*c)

```
string grid_search(vector<string> G, vector<string> P, int R, int C,
int r, int c) {
  // Declare the r x R matrix
  vector<vector<pair<bool, vector<int>>>> dp(r, vector<pair<bool,
vector<int>>>(R));

  // Fill it
  for (int i = 0 ; i < r ; ++i){
    vector<vector<int>> tmp(R);
    for (int j = 0 ; j < R ; ++j){
      size_t p = G[j].find(P[i]);
      while (p != string::npos){
        tmp[j].push_back(p);
        dp[i][j] = {true, tmp[j]};
        p = G[j].find(P[i], p + 1);
      }
    }
  }

  // Run over all diagonals
  for (int d = 0 ; d < R-r+1 ; ++d){

    // hash map to compute the number of occurence of each position
    map<int, int> occ;

    // Run over diagonal (d)
    for (int i = 0 ; i < r ; ++i){
      if (dp[i][i+d].first){
        // Compute the number of occurence of the position (p) in (d)
        for (auto p: dp[i][i+d].second)
          occ[p]++;
      }
      else
        break;
    }

    for (auto cnt: occ)
      // If #occurence of a position is equal to (r), (P) exists in (G)
      if (cnt.second == r) return "YES";
  }

  return "NO";

}
```