Problem name: Mascot Song

Language: English

Solution

This problem was all about smart simulation with the help of the simple data structure – an array. Using the definition of a block, it directly follows that some block ends at position i and the next one starts at position $i+1$ if and only if $A_i \geq A_{i+1}$. Now, for all $1 \leq i < n$, let us call a consecutive pair $(i, i+1)$ **bad** if $A_i \geq A_{i+1}$ and let $b(A)$ – the total number of bad pairs of sequence A. Now, the most importan (and obvious) fact is that the number of blocks of sequence A equals $1 + b(A)$.

The solution for 30 points is to simulate both types of queries directly, with constant complexity for query of type 1 and linear complexity for query of type 2. Number of bad pairs after each query can be computed in linear time and the time complexity of this algorithm is $O(q \cdot n)$.

For the further 30 points (only queries of type 1) it should be noted that query "1 x y" can only affect consecutive pairs $(x-1, x)$ and $(x, x+1)$, if they exists. Therefore, if we precompute $b(A)$ at the beginning, we can check in constant time if $b(A)$ changed and how; after that, we update $b(A)$ and A_x. Time complexity of this solution is $O(n+q)$.

For the full score, we need to handle the queries of type 2. However, note that the query "2 z" has the same effect as switching first z elements with last $n-z$ elements. During this process, we "break" one consecutive pair - (A_z, A_{z+1}) and form another - (A_n, A_1). The rest $n-3$ consecutive pairs stay the same; therefore, we can update $b(A)$ in constant time. In order to keep up with the indices, we will have one extra variable – $first$ which will denote the original index of the currently first element of sequence A; for example, if "2 z" is the first query, the sequence would look like $(A_{z+1}, A_{z+2}, \dots, A_n, A_1, A_2, \dots A_z)$ and we would have $first = z+1$. With this notation, the current position x is actually the position $first + x - 1$ of the original array (all operations are done modulo n). Time complexity of this solution is $O(n+q)$ and memory complexity is $O(n)$.

```
#include <cstdlib>
#include <cstdio>

const int MaxN = 200200;

int n, q, groupNum, first, last;
int A[MaxN];

int main()
{
        scanf("%d", &n);
        for (int i = 0; i < n; i++)
                scanf("%d", &A[i]);

        first = 0;
        last = n - 1;
        groupNum = 1;
        for (int i = 0; i < n - 1; i++)
                if (A[i] >= A[i + 1]) groupNum++;

        scanf("%d", &q);
        for (int query = 0; query < q; query++)
        {
                int type, x, y, z;
                scanf("%d", &type);

                if (type == 1)
                {
                        scanf("%d%d", &x, &y);
```

```
                        int curr = (first + x - 1) % n;
                        int prev = (curr - 1 + n) % n;
                        int next = (curr + 1) % n;

                        // .... A[prev] A[curr] A[next] ...  --->  A[prev] y A[next]
                        if (curr != first && A[prev] < A[curr] && A[prev] >= y)
                                groupNum++;
                        if (curr != first && A[prev] >= A[curr] && A[prev] < y)
                                groupNum--;
                        if (curr != last && A[curr] < A[next] && y >= A[next])
                                groupNum++;
                        if (curr != last && A[curr] >= A[next] && y < A[next])
                                groupNum--;

                        A[curr] = y;
                }
                else
                {
                        scanf("%d", &z);
                        if (z != n)
                        {
                                int left = (first + z - 1) % n;
                                int right = (left + 1) % n;

                                // A[first] .... A[left] | A[right] ... A[last] ---> A[right]
//... A[last] | A[first] ... A[left]
                                if (A[left] >= A[right])
                                        groupNum--;
                                if (A[last] >= A[first])
                                        groupNum++;

                                first = right;
                                last = left;
                        }
                }

                printf("%d\n", groupNum);
        }

        return 0;
}
```

Problem name: Tickets — Language: English

Solution

This problem is ad-hoc problem where in order to come up with the solution we need to carefully analyze the given method for creating the gray code.

The solution for 30 points is to simulate described method and print the K^{th} string in the list. Both time and space complexity of this solution is $O(2^n)$.

The solution for 100 points is a little bit more complicated. First, we can see that if $K \leq 2^{n-1}$ (if the number that we are looking for is in the first half of the gray code), then the first bit in the result is 0, and if $K > 2^{n-1}$, then the solution starts with 1. Furthermore, now when we have found the first bit in the solution, let's erase the first bit from all numbers in the gray code. What we are left with is an array where the first half is the gray code of all numbers consisting of $n-1$ bits, and the second half is same as the first half, just reversed. So if we are looking for $K \leq 2^{n-1}$ then what we are left with is the problem of finding the K^{th} element in the gray code of all numbers consisting of $n-1$ bits, and if $K > 2^{n-1}$ then we need to find the $(K-2^{n-1})^{th}$ element in the reversed gray code of order $n-1$, which is the same as searching for $(2^n - K + 1)^{th}$ element in the gray code of order $n-1$. To get all the bits, we repeat this process n times.

So, the algorithm for 100 points is:

1. While $(n > 0)$
2. If $K \leq 2^{n-1}$ print 0
3. Else /* $K > 2^{n-1}$ */
4. print 1
5. $K = 2^n - K + 1$
6. $n = n - 1$

The time complexity of this solution is $O(n)$ and space complexity is constant.

```
#include <stdio.h>
#include <string>
using namespace std;

int main(){
    unsigned long long n, k; scanf("%llu%llu", &n, &k);

    string sol = "";
    for (int i = 0, N = n; i < N; i++){
        if (k <= (1LLU << (n - 1))){
            sol += "0";
        }
        else {
            sol += "1";
            k = (1LLU << n) - k + 1;
        }
        n--;
    }

    printf("%s\n", sol.c_str());
    return 0;
}
```

Problem name: Andres Iniesta Language: English

Solution

Many problems that require finding optimal solution are solvable by using dynamic programming approach. Here, we are going to present solution in time complexity $O(N \cdot M \cdot K^2)$.

Let's define matrix $left[N][M][K]$, where $left[r][c][l]$ represents maximal number of visible cells left from cell (r, c) if Iniesta can destroy up to l obstacles that are located left of him (same row with lower index of column). It not hard to see that following recursive formula is correct (if cell (r, c) contains obstacle, then we can destroy 1 less obstacle left of cell that cell):

$$left[r][c][l] = \begin{cases} left[r][c-1][l-1] + 1, & if\ there\ is\ obstacle\ at\ cell\ (r,c) \\ left[r][c-1][l] + 1, & otherwise \end{cases}$$

Where initial values are 0 or 1 based on that if cell contain obstacle or not.

In the similar way we can define matrixes $right[N][M][K], up[N][M][K], down[N][M][K]$. Formulas for constructing those matrix are very similar.

Now we have matrixes $left, right, up$ and $down$, but we still have to combine then to get a solution. We are going to make matrix $column[N][M][K]$, where $column[r][c][l]$ represents maximal number of visible cells in the same column where cell (r, c) is located in the matrix if we can destroy up to l obstacles in column c. We can find $column[r][c][l]$ by trying all combinations destroying p obstacles up from cell (r, c) and $l - p$ down of that cell. If cell (r, c) in the matrix contains an obstacle, then we can destroy 1 less obstacle when we observe all combinations.

$$column[r][c][l] = \begin{cases} \max\limits_{p=0..l-1} (left[r][c-1][p] + right[r][c+1][l-p] + 1), (r,c)\ has\ obstacle \\ \max\limits_{p=0..l} (left[r][c-1][p] + right[r][c+1][l-p] + 1), \quad otherwise \end{cases}$$

In similar way we can define matrix $row[N][M][K]$, where $row[r][c][l]$ represents maximal number of visible cells that are located in same row as cell (r, c) if Iniesta can destroy up to l obstacles in that same row.

Finally we are going to combine matrixes $column$ and row to get a result. For every cell in matrix we are going to calculate maximal field of view if we stand on that cell and can destroy K cells.

For each cell we should try all combinations of destroying obstacles, where combinations differ from each other by number of destroyed obstacles in row and in column. So if cell (r, c) doesn't contain an obstacle we can conclude that

$$solution[r][c] = \max_{l=0..K} column[r][c][l] + row[r][c][K-l] - 1$$

We have -1 in last formula because we are counting cell (r, c) 2 times, first time in sum $column[r][c][l]$ and second time in sum $row[r][c][K - l]$.

If cell (r, c) contains an obstacle, that means we have to destroy that obstacle. Formula becomes slightly different than the last one.

After calculating solution for each cell, we have to pick the best one. Calculated matrixes $left, right, up, down$ and $solution$ is done in time complexity $O(N \cdot M \cdot K)$, but matrixes $column$ and row are computed in time complexity $O(N \cdot M \cdot K^2)$ so time complexity of whole solution is $O(N \cdot M \cdot K^2)$.

```
#define _CRF_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_DEPRECATE

#include <stdio.h>
```

```
const int MaxN = 210;
const int MaxM = 210;
const int MaxK = 55;

int n, m, k;
char map[MaxN][MaxM];
char map2[MaxN][MaxN];

short column[MaxM][MaxN][MaxK];
short row[MaxN][MaxM][MaxK];

short left[MaxN][MaxK];
short right[MaxN][MaxK];

short max(short a, short b) { return a > b ? a : b; }
short min(short a, short b) { return a < b ? a : b; }

void CalculateRow(int m, char *map, int _k, short row[MaxN][MaxK])
{
	// left
	left[0][0] = map[0] == '.';
	for (int i = 1; i < m; ++i) {
		if (map[i] == '*') left[i][0] = 0;
		else left[i][0] = left[i - 1][0] + 1;
	}

	for (int k = 1; k <= _k; ++k) {
		left[0][k] = 1;
		for (int i = 1; i < m; ++i) {
			if (map[i] == '*') left[i][k] = left[i - 1][k - 1] + 1;
			else left[i][k] = left[i - 1][k] + 1;
		}
	}

	// right
	right[m - 1][0] = map[m - 1] == '.';
	for (int i = m - 2; i >= 0; --i) {
		if (map[i] == '*') right[i][0] = 0;
		else right[i][0] = right[i + 1][0] + 1;
	}

	for (int k = 1; k <= _k; ++k) {
		right[m - 1][k] = 1;
		for (int i = m - 2; i >= 0; --i) {
			if (map[i] == '*') right[i][k] = right[i + 1][k - 1] + 1;
			else right[i][k] = right[i + 1][k] + 1;
		}
	}

	// merge
	for (int i = 0; i < m; ++i) {
		if (map[i] == '*') row[i][0] = 0;
		else row[i][0] = left[i][0] + right[i][0] - 1;
	}

	for (int k = 1; k <= _k; ++k) {
		row[0][k] = right[0][k];
		row[m - 1][k] = left[m - 1][k];

		for (int i = 1; i < m - 1; ++i) {
			int newK = k;
			if (map[i] == '*') newK = k - 1;

			for (int leftK = 0; leftK <= newK; ++leftK) {
				int rightK = newK - leftK;
				row[i][k] = max( row[i][k], left[i - 1][leftK] + right[i +
1][rightK] + 1 );
			}
```

```
                }

        }
}

int main()
{
        scanf("%d%d%d", &n, &m, &k);
        for (int i = 0; i < n; ++i) {
                scanf("%s", map[i]);
        }

        // rotate matrix (columns to rows)
        for (int i = 0; i < n; ++i) {
                for (int j = 0; j < m; ++j)
                        map2[j][i] = map[i][j];
        }

        for (int i = 0; i < n; ++i)
                CalculateRow(m, map[i], k, row[i]);
        for (int i = 0; i < m; ++i)
                CalculateRow(n, map2[i], k, column[i]);

        short oldret = -1;
        short retR = -1;
        short retC = -1;
        short rowObstacles = -1;
        short columnObstacles = -1;

        short ret = 0;
        for (int i = 0; i < n; ++i) {
                for (int j = 0; j < m; ++j) {
                        for (int rowK = 0; rowK <= k; ++rowK) {
                                int columnK = k - rowK;

                                if (map[i][j] == '*') {
                                        if (rowK != 0)
                                                ret = max(ret, row[i][j][rowK] +
column[j][i][columnK + 1] - 1);
                                        if (columnK != 0)
                                                ret = max(ret, row[i][j][rowK + 1] +
column[j][i][columnK] - 1);
                                }
                                else {
                                        ret = max(ret, row[i][j][rowK] + column[j][i][columnK] -
1);
                                }

                                if (oldret < ret) {
                                        oldret = ret;
                                        retR = i;
                                        retC = j;
                                        rowObstacles = rowK;
                                        columnObstacles = columnK;
                                }
                        }
                }
        }
        printf("%d\n", ret);

        /*
        printf("Row = %d\n", retR);    printf("Column = %d\n", retC);
        printf("Mapa[%d][%d] = %c\n", retR, retC, map[retR][retC]);
        printf("Row obstacles = %d\n", rowObstacles);
        printf("Column obstacles = %d\n", columnObstacles);
        */

        return 0;
}
```