# USACO JAN14 Problem 'skilevel' Analysis

**by Nathan Pinsker**

The structure of this problem suggests, at first, the straightforward approach of performing flood fills at each of the MN points in the grid to find our desired value of D at each point. For each point P, we can binary search on the minimum value of D such that starting from P allows a cow to reach at least T points. (This works because increasing the maximum elevation we can change by between two points only helps us.) However, this method can take as long as O((MN)^2) times a logarithmic factor for the binary search, which is too slow given our bounds.

A bit more thinking reveals the following simple insight: the difficulties of nearby points are often correlated! For example, if we know the difficulty of a point P is D, then every point that we reach from P in our previous flood fill will also have difficulty at most D. (We know we can get from any of these T other points to any other one.) It is possible that these other points actually have a lower difficulty, so this insight doesn't end up helping our runtime much. However, this does at least motivate another approach! If we could somehow find the points with lowest difficulty first, then we could use that information to efficiently find the points with higher difficulty.

We can do this by thinking of the MN points as points in a graph. The possible edges in this graph are between adjacent pair of points, and their edge weights are equal to the difference between those points' elevations. We add edges in increasing order of weight to our graph, and keep track of each connected component. (This means if we have added edges up to weight D, then each component will just represent a set of points that we can get between using elevation differences of at most D.) Thus, we can assign a difficulty to all points in a component precisely when its component's size is at least T.

The only graph operations we need to support are adding an edge, checking if two vertices are connected, and querying the size of a component. All of these are super quick using union find!

Below is Travis Hance's code.

```
#include <cstdio>
#include <algorithm>
using namespace std;

#define NMAX 500

int elevation[NMAX][NMAX];

struct node {
    int x, y;

    node* pa;
    int rank, comp_size, min_weight;
};
node nodes[NMAX][NMAX];

struct edge {
    node *a, *b;
    int weight;
    edge() {}
    edge(node *a, node *b, int weight) : a(a), b(b), weight(weight) {}
    bool operator<(edge const& e) const {
        return weight < e.weight;
    }
};
```

```
edge edges[2 * NMAX * (NMAX-1)];

void uf_merge(node* a, node* b) {
    if (a->rank < b->rank) {
        a->pa = b;
        b->comp_size += a->comp_size;
    } else {
        b->pa = a;
        a->comp_size += b->comp_size;
        if (a->rank == b->rank) {
            a->rank++;
        }
    }
}

node* uf_find(node* a) {
    node* b = a->pa;
    if (b->pa != b) {
        uf_find(b);
    }
    if (a->min_weight == -1) {
        a->min_weight = (b->min_weight == -1 ? b->pa->min_weight : b->min_weight);
    }
    a->pa = b->pa;
    return a->pa;
}

int main() {
#ifndef HOME
    freopen("skilevel.in", "r", stdin);
    freopen("skilevel.out", "w", stdout);
#endif

    int m, n, t;
    scanf("%d %d %d", &m, &n, &t);
    for (int y = 0; y < m; y++) {
        for (int x = 0; x < n; x++) {
            scanf("%d", &elevation[x][y]);

            nodes[x][y].x = x;
            nodes[x][y].y = y;
            nodes[x][y].pa = &nodes[x][y];
            nodes[x][y].rank = 0;
            nodes[x][y].comp_size = 1;
            nodes[x][y].min_weight = -1;
        }
    }

    int nedges = 0;
    for (int y = 0; y < m; y++) {
        for (int x = 0; x < n; x++) {
            if (y < m-1) {
                edges[nedges++] = edge(&nodes[x][y], &nodes[x][y+1],
                    abs(elevation[x][y] - elevation[x][y+1]));
            }
            if (x < n-1) {
                edges[nedges++] = edge(&nodes[x][y], &nodes[x+1][y],
                    abs(elevation[x][y] - elevation[x+1][y]));
            }
        }
    }

    sort(edges, edges + nedges);
    for (int i = 0; i < nedges; i++) {
        edge e = edges[i];
        node* a = uf_find(e.a);
        node* b = uf_find(e.b);
        if (a != b) {
            if (a->comp_size + b->comp_size >= t) {
                if (a->comp_size < t) {
                    a->min_weight = e.weight;
                }
                if (b->comp_size < t) {
                    b->min_weight = e.weight;
                }
            }
            uf_merge(a, b);
        }
    }

    long long total = 0;
    for (int y = 0; y < m; y++) {
        for (int x = 0; x < n; x++) {
            int is_start;
            scanf("%d", &is_start);
            if (is_start) {
                // has side-effect of setting min_weight
                uf_find(&nodes[x][y]);

                total += (long long) nodes[x][y].min_weight;
            }
        }
    }

    printf("%lld\n", total);
}
```