USACO FEB11 Problem 'tri' Analysis

by Lewin Gan

We can see that a sub-triangle is defined by three traits

- Whether it is facing up or down
- Where the point of the tip is
- The side length of the triangle

For the first condition, we have two possibilities, for the second, we have about N^2, and for the third, we have about N. Therefore, the best we can possibly get if we tried every single sub-triangle would be about $2N^3$. Therefore, let us first try to get this solution.

We can deal with upside down and right side up triangles in similar ways. The naive solution would be $O(N^5)$, which would be for every point, compute the sub-triangle with side length K by summing up all the elements then dividing by the number of elements. This can easily be reduced to $O(N^4)$ by seeing that if we calculated the sub-triangle with length K - 1 at a point P, all we have to do to calculate the sub-triangle with length K at the point P is to add in the last row. To reduce this even further, we can take partial sums in each row, so that `sum [i][j]` contains `v_i1 + v_i2 + v_i3 + ... + v_ij`. Then, we can compute the sum of a part of the row in constant time. This will reduce the complexity to $O(N^3)$.

However, this $O(N^3)$ solution is still too slow for N <= 700. Let's look into the problem a little more. We want to compute the maximum *average* of a sub-triangle. This gives us a clever idea. If we have a sub- triangle of size 2K+1 (odd), we can split it up into four smaller sub-triangles, three with size K, and one with size K+1. In addition, if we have a sub-triangle of size 2K (even), we can split it up into four smaller sub-triangles, three with size K, and one with size K-1. Below is an example of how we can do this:

```
ODD:                        EVEN:
                                  []
      []                         [][]
     [][]                       [][][]
    ------                      ------
   \[][][]/                    []\[][]/[]
  []\[][]/[]                  [][]\[]/[][]
 [][]\[]/[][]                [][][]\/[][][]
```

Since we can split up these triangles into four smaller regions, the average of the bigger triangle must be less or equal to the maximum average of the four smaller sub-triangles. That means, we only really need to check triangles of size K...2K since any triangle with size greater than 2K can be split up into smaller sub-triangles that fit within our range. Therefore, since we only need to check K+1 values for our height, this reduces our complexity to $O(K N^2)$

Sample solution below:

```
#include <cstdio>
#include <algorithm>

#define FOR(i,a,b) for (i = a; i <= b; i++)
#define DFR(i,j,b) FOR(i,1,b) FOR(j,1,i)

using namespace std;

typedef long long ll;

FILE *fin = fopen ("tri.in", "r"), *fout = fopen ("tri.out", "w");

const int MAXN = 760;
const ll INF = (1ll << 50);

int N, K;
ll sum [MAXN][MAXN];

inline void maxc (ll &a, ll b) {if (b > a) a = b;}
inline int min (int a, int b) {return a < b ? a : b;}

int main () {
    fscanf (fin, "%d %d", &N, &K);
    int i, j, k, val, upper;
    DFR (i, j, N) {
        fscanf (fin, "%d", &val);
        sum [i][j] = val + sum [i][j - 1];
    }

    ll max = -INF, up, down, div;
    DFR (i, j, N) {
        up = 0, down = 0, div = 0;

        // right side up triangles
        upper = N - i + 1;
        if (upper >= K) FOR (k, 1, min (2 * K, upper)) {
            div += k;
            down += sum [i + k - 1][j + k - 1] - sum [i + k - 1][j - 1];
            if (k >= K) {maxc (max, down / div);}
        }

        // upside down triangles
        div = 0;
        upper = min (j, i - j + 1);
        if (upper >= K) FOR (k, 1, min (2 * K, upper)) {
            div += k;
            up += sum [i - k + 1][j] - sum [i - k + 1][j - k];
            if (k >= K) {maxc (max, up / div);}
        }
    }

    fprintf (fout, "%lld\n", max);
    return 0;
}
```