Keith and the Egyptian Tomb (300)

Uday Shankar

The most obvious way to approach this problem is to just loop through all possible subsquares and find their sums. Unfortunately, this algorithm takes $O(N^5)$ time, so although it might work fine for the sample case, it certainly is too slow for $N = 10000$.

Clearly, a better approach is needed. Instead of looping through all possible subsquares, let's instead count how many times the value in each cell is counted. If we can generate this count for each cell in $O(M)$ time, we can finish off the problem quickly by looping through the matrix supplied in the input file, multiplying each number by its count, and summing - all in $O(MN^2)$ time.

Let's try to develop a formula for the count, so that we can just plug in the location of the cell (i, j), where $(1, 1)$ is the top-left corner, and compute the count in constant time. To develop this formula, we consider subsquares of all possible side lengths $1 \leq a \leq N$. Also note that we need only consider the cells where $i \leq j \leq \frac{N}{2}$; the rest of the counts can be computed using symmetries across the two main diagonals and the vertical and horizontal lines that divide the matrix into four equal squares.

For purposes of the following argument, it may help to visualize the matrix of numbers as a 10000x10000 chessboard and the subsquares as rigid square frames of side length a.

Using these assumptions and simplifications, we note that if $a \leq i$, then the cell (i, j) can be placed in any of the cells inside the subsquare of side length a without the subsquare's frame colliding with the edges; hence, it should be counted a^2 times. If $i < a \leq j$, then the subsquare has full range of motion in the y-direction, but the range of motion in the x-direction is limited by the distance between the cell and the left side edge of the matrix. This distance is precisely i. Since the subsquare can be moved such that the cell (i, j) takes up all a positions in the subsquare in the y-direction, but only i positions in the x-direction, the cell should be counted ai times. Finally, if $i < j < a$, then the subsquare has only range of motion in the x- and y-directions of i and j respectively, so in this case, the cell should be counted ij times. Summing over all values of a and using the symmetry of the sum, this becomes

$$2\left(\sum_{a=1}^{i} a^2 + i\sum_{a=i+1}^{j} a + ij\sum_{a=j+1}^{\frac{N}{2}} 1\right)$$

where constants have been pulled out of the sum. Note that this formula

only works for even N, a small adjustment is necessary to accommodate odd N. All of these sums have well-known explicit formulas, so using those gives the constant time expression:

$$\frac{i(i+1)(2i+1)}{6} + i\left(\frac{j(j+1)}{2} - \frac{i(i+1)}{2}\right) + ij\left(\frac{N}{2} - j\right)$$

All that remains is to loop through all the cells in the matrix and multiply the values by this computed count and sum. Sample C code that finds the solution in around 8 seconds on my computer is given below. Note that an unsigned 64-bit integer type (unsigned long long is guaranteed this width in C) is necessary to avoid overflow.

```
#include <stdio.h>

#define N 10000

int min(int a, int b) {
  return a < b ? a : b;
}

unsigned long long count(int i, int j) {
  unsigned long long in = min(i + 1, N - i);
  unsigned long long jn = min(j + 1, N - j);
  if(in > jn) { // swaps in and jn if in > jn
    in ^= jn;
    jn ^= in;
    in ^= jn;
  }
  unsigned long long ret = 0;
  ret += (in * (in + 1) * (2 * in + 1)) / 6;
  ret += in * ((jn * (jn + 1)) / 2 - (in * (in + 1)) / 2);
  ret += (in * jn) * ((N + 1)/2 - jn);
  ret *= 2;
  return ret;
}



int main() {
  FILE *in = fopen("egypt.in", "r");
  unsigned long long ans = 0;
  unsigned int num;
  for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
      fscanf(in, "%d", &num);
      ans += count(i, j) * num;
```

```
    }
  }
  printf("%llu\n", ans);
  return 0;
}
```