Code Jam 2010 - Round 1C

Analysis: Making Chess Boards

Finding the largest chess board

First, we need a quick way to find the largest chess board. There is classic dynamic programming trick that goes like this. Let's compute, for each cell (i, j), the size of the largest square whose bottom-right corner is (i, j). Let's call this value larg[i][j]. It is easy to compute larg[i][0] and larg[0][j] -- they are always 1. For any other cell, the value of larg[i][j] is always at least 1, and it is larger only if the following condition holds:

```
if (board[i - 1][j] != board[i][j] &&
    board[i][j - 1] != board[i][j] &&
    board[i - 1][j - 1] == board[i][j]) {
  larg[i][j] = 1 + min(larg[i - 1][j],
                       larg[i][j - 1],
                       larg[i - 1][j - 1]);
}
```

In a single, linear-time, row-by-row scan, we can compute the values of larg[][] for all cells.

Finding the chess board to remove

Now that we have larg[][], it is very easy to find the first chess board that we should cut out. Its bottom-right corner is in the cell that has the largest possible value in larg[][]. If there are several such cells, we use the tie-breaking rules described in the problem and choose the one that comes first in lexicographic order of (i, j).

We can do this in linear time by scanning larg[][], but since we will have to do this many times, it is better to make a heap of triples of the form

```
(-larg[i][j], i, j)
```

and take the smallest element from that heap. This way, we are sorting all cells by decreasing size, then by increasing row, then by increasing column. As long as we can update this heap efficiently after cutting out a chess board, we can always retrieve the smallest element in O(log(m*n)) time. We could also use a balanced binary search tree instead of a heap.

Removing the chess board and updating larg[][]

Consider removing the first 6x6 chess board from the example input described in the problem statement. How should we update larg[][]? First of all, we can fill the 6x6 square of cells with zeros because there are no more chess boards to be removed from those locations. But that is not all. There are other cells that might need to be updated. Where are they, and how many of them are there?

Naively, we can simply recompute the values of all non-zero cells in larg[][] and continue. If we do that, we will have a $O(m^2*n^2)$ algorithm, which is too slow.

First of all, notice that we do not need to update rows above or to the left of the 6x6 square. Any square chess boards whose bottom-right corners are in those areas still exist and can be cut out later. The only chess boards that we need to worry about are those that overlap the 6x6 board that we have just removed. Also, notice that we have just removed the largest possible chess board, so we only need to care about remaining boards of size 6 or smaller. Where can their bottom-right corners lie in order for those boards to overlap *our* board? They must be in the 12x12 square whose center is at (i, j) -- the bottom-right corner of our board.

That's an area of size $4*6^2$. In fact, whenever we remove a board of size k-by-k, we only need to update an area of larg[][] of size at most 2k-by-2k. Since we can only remove each cell at most once, all of the updating work requires linear time in total; 4*m*n updates, to be precise.

Updating the heap

Each time we update larg[][], we must also update the heap that lets us find the next board to remove. This means finding and removing an old entry, as well as inserting a new entry. With pointers from cells to heap elements, or by using a balanced binary search tree, both steps can be done in O(m*n) time.

In total, this algorithm runs in O(n*m*log(n*m)) time, which is plenty fast for the problem's constraints.