SACO 2006 Day 2 solutions

Bruce Merry

Lights

A brute-force solution operates left-to-right, and hits the toggle on each light that is on. Since hitting a toggle involves updating all lights to the right of it, this can be quite slow.

A faster solution can be found by noticing that for a pair of neighbouring lights A and B (with A left of B), the only switch that alters exactly one of the lights is the one on light B (the others flip either both or neither). Thus, if A and B start in opposite states, the switch on B must be flipped, and if they start in the same state, the switch on B must not be flipped. To handle the left-most light, imagine that there is a virtual green light to the left of it.

How not to be seen

A brute-force solution simply checks every possible placement of the $W \times H$ rectangle and counts the number of non-hiding places. However, some mathematics can speed this up.

Consider an optimal placement of the rectangle, and suppose that the row pattern for these H rows has r_0 zeros and r_1 ones, and that the column pattern for these W columns has c_0 zeros and c_1 ones. The number of non-hiding places is then $r_0c_0 + r_1c_1$. Suppose $c_0 < c_1$: then if there were a way to move the rectangle vertically to increase r_1 and decrease r_0, this would increase the number of non-hiding square. Similarly, if $c_0 > c_1$ then r_0 must be maximal (and r_1 minimal). Applying a similar argument to the columns shows that r_1 and c_1 are either maximal or minimal (it is possible that one is maximal and one is minimal).

It is thus sufficient to find the maximal and minimal values of r_1 and c_1. One can quickly find all values of r_1 by moving a "sliding window" over the row pattern, just adding one new value and subtracting one old value.

Putting things on top of other things

The first step is to build the grid i.e., construct a 2D array in which the blocked cells are marked. This will be more efficient that searching a list for the algorithm we propose.

For each grid cell, we compute and save the size of the largest square that can be constructed with that cell as the top-left corner; we call this function f. The answer is simply the sum of these values. Consider four adjacent grid cells:

A	B
C	D

If cell A is blocked, the largest square that can be built there is size zero (i.e., no square can be built). Otherwise, it can be at most one larger than the largest squares that can be built from B, C or D. Conversely, if squares of size S can be built at B, C and D, then a square of size $S + 1$ can be built at A. It follows that

$$f(A) = \min\{f(B), f(C), f(D)\} + 1$$

if cell A is not blocked.

By running a double loop starting from the bottom-right of the grid, we can fill in a 2D array of all the values of f, and only ever read back values that have already been computed.

Cave of Caerbannog

Many of the tunnels will never be used by the knights, because they can be circumvented using less dangerous tunnels. We start by building the subset of tunnels that are useful to the knights. Start with no tunnels at all. The least dangerous tunnel is clearly useful to the knights. The second-least dangerous tunnel is also useful, unless it links the same chambers as the previous tunnel. We can keep adding tunnels in increasing order of danger, discarding any for which there is already an alternative (and hence less dangerous) route.

The resulting set of tunnels will not have any loops in it. If the original cave was connected, then such a subset is called a *tree* (because the branches of a tree never form loops), and the particular tree that we have constructed happens to be the *Minimum Spanning Tree* (the tree that connects everything with least danger sum). The algorithm we used to generate the tree is called Kruskal's algorithm.

We have now reduced the problem from a general network to a tree. We can pick an arbitrary node of the tree and declare it to be the root: the tree then has a hierarchical structure like a binary search tree. To answer a query, we take the two chambers and follow edges up the tree until we reach a common ancestor; this defines the unique path between them.

If some of the paths are very long, this walking procedure will not be efficient enough. It can be optimised by precomputing pointers to the kth ancestor, where k is 1 (parent), 2 (grand-parent), 4, 8, etc. With each pointer, it is necessary to store the danger involved in moving up k levels. This allows us to take larger steps through the tree.