USACO March 2006 Problem 'xlite' Analysis

by Bruce Merry

This is a surprisingly difficult dynamic programming problem. Firstly, we assume that the fork has its end tines present - if not, we can eliminate those slots and the corresponding ends of the light switches (since the end switches cannot be touched). If we work left-to-right, we can set all but the last T - 1 switches to any state we want - but then we cannot make any changes the last T - 1 switches. This suggests that we make the state of the last T - 1 switches the focus of our dynamic programming. This is possible, but it turns out that the implementation is slightly easier if we use the last T switches. In particular, for each final setting of the last T switches, we store the minimum total number of lights remaining.

If T = L, this is easy to compute. There are only two possible final settings (we either do a flip or we don't); in each case we count the lights. The remaining configurations we label as impossible to achieve e.g. with INT_MAX as the number of lights.

Now suppose that we increase L by one, adding an extra light at the end, and want to determine the minimum number of lights for a particular configuration. Consider what happens to the final switch: either it will be left alone, or it will be flipped by placing at the fork at the right-most position (there is not point doing this twice, since they will cancel out). The order of operations is irrelevant, so assume that this happens last if at all. We can loop over the two options (for a particular final configuration of the last T switches), and choose the option that gives the best result. Once this choice is made, we just have to find how best to minimise the number of lights on before the (possible) last step, considering only L - 1 lights, using the information we have already determined. The last T - 1 of these lights have now been fixed, so we just take the best choice for the T'th light from the end. Note that "best" means fewest lights remaining, but with ties broken by the number of flips.

To recover the switches that were flipped, we apply the same process, but working from the end towards the beginning. Alternatively, for each entry in the DP table we can store the entire set of lights to flip - there are only 50 bits to store, so the memory consumption is quite reasonable if they are packed 8 bits to the byte.

Running time is $O(L*2^T)$, which leaves plenty of breathing room.

An alternative is to search all possible subsets of the switches to flip. An upper bound can quickly be established by turning off all the lights except the last T - 1 (which is at most 6). This provides significant pruning, and some further pruning solves most of the test cases within the time limit.

```
#include <fstream>
#include <algorithm>
#include <utility>
#include <cassert>
#include <vector>
#include <iterator>
```

```
#include <stdint.h>

using namespace std;

#define MAXL 50
#define MAXT 7
#define MAX2T (1 << MAXT)

int main() {
    int L, T;
    uint64_t state = 0;
    uint8_t tines = 0;
    char ch;
    int x;
    pair<int, int> dp[MAXL + 1][MAX2T];
    bool usek[MAXL + 1][MAX2T];
    int mask, high;
    ifstream in("xlite.in");
    ofstream out("xlite.out");

    in >> L >> T;
    for (int i = 0; i < L; i++) {
        in >> ws >> ch;
        state = (state << 1) + (ch - '0');
    }
    for (int i = 0; i < T; i++) {
        in >> ws >> ch;
        tines = (tines << 1) + (ch - '0');
    }
    if (T > L) {
        out << "0\n";
        return 0;
    }

    assert(tines);
    while (!(tines & (1 << (T - 1)))) {
        state &= ~(1ULL << (L - 1));
        L--;
        T--;
    }
    while (!(tines & 1)) {
        tines >>= 1;
        state >>= 1;
        T--;
        L--;
    }

    if (T == 1) {
        out << __builtin_popcountll(state) << "\n";
        for (int i = 0; i < L; i++)
            if ((state >> (L - 1 - i)) & 1) out << i + 1 << "\n";
        return 0;
    }
    if (L < T)
    {
        out << "0\n";
        return 0;
    }

    mask = (1 << T) - 1;
    high = (1 << (T - 1));
    fill(dp[T], dp[T] + mask + 1, make_pair(INT_MAX, INT_MAX));
    x = state >> (L - T);
    dp[T][x] = make_pair(__builtin_popcount(x), 0);
    dp[T][x ^ tines] = make_pair(__builtin_popcount(x ^ tines), 1);
    for (int i = T + 1; i <= L; i++)
        for (int j = 0; j <= mask; j++) {
            bool toggle;
            int jm;
            int on = (state >> (L - i)) & 1;

            toggle = (j & 1) ^ on;
            jm = (toggle ? (j ^ tines) : j) >> 1;
            pair<int, int> adj(__builtin_popcount(j) -
__builtin_popcount(jm), toggle ? 1 : 0);

            dp[i][j] = make_pair(INT_MAX, INT_MAX);
            for (int k = 0; k < 2; k++) {
                int jk = k ? jm | high : jm;
                pair<int, int> cur = dp[i - 1][jk];
                if (cur.first == INT_MAX) continue;
                cur.first += adj.first;
                cur.second += adj.second;
                if (cur < dp[i][j])
                {
                    dp[i][j] = cur;
                    usek[i][j] = k;
                }
            }
        }

    int u = min_element(dp[L], dp[L] + mask + 1) - dp[L];
    vector<int> used;
    for (int i = L; i > T - 1; i--) {
        int oldu = u;
        if ((u & 1) ^ ((state >> (L - i)) & 1)) {
            used.push_back(i - T + 1);
            u ^= tines;
        }
        u >>= 1;
        if (usek[i][oldu]) u |= high;
    }
    out << used.size() << "\n";
    copy(used.begin(), used.end(), ostream_iterator<int>(out, "\n"));
    return 0;
}
```