USACO OPEN14 Problem 'fairphoto' Analysis

by Bruce Merry

Let T(b, p) be the number of cows of breed b in the photo to the left of point p. If the left and right end-points are L and R, then the photo is valid if there is some subset A of the breeds (with |A| >= K) such that

- T(b, R) = T(b, L) + c for each b in A
- T(b, R) = T(b, L) for each b not in A

For a given A and p, we can create a "signature" S(A, p) which consists of

- T(b, p) - T(b0, p) for each b in A, where b0 is the first element in A
- T(b, p) for each b not in A

With a little work, one can show that if S(A, L) = S(A, R) and L < R, then the photo from L to R is valid.

Now let us consider each possible right endpoint R, and consider how to find a matching L. We do not need to consider all subsets A: imaging starting with no cows in the photo, and gradually extending the left edge to include more cows. At some points a new breed will enter a photo, but no breed will ever disappear from the photo, so there can be at most O(B) distinct subsets of breeds in the photo. We can efficiently compute these subsets by keeping track of the rightmost point at which each breed appeared before the current point.

Now that we've picked R and A, and computed the signature S(A, R), how do we find a matching L? We precompute all values of S(A, L), and store them in a lookup table (such as a hash table, balanced tree or sorted list). We then just consult the table to find the corresponding minimum value of L. Similarly to before, we only need to consider O(B) possible values of A for each L.

The runtime for this is O(B^2N) for a hash table or O(B^2N (log N + log B)) for a sorted list: there are O(BN) precomputations and queries, and each signature has O(B) size. I found that the run time of both was very similar in practise.

An alternative is to consider all O(2^B) valid values for A in an outer loop. If one indexes not by the full signature but by a 64-bit hash of it, then the runtime becomes O(2^BN), but in the unlikely event of two different signatures hashing to the same 64-bit value, the answer may be incorrect or must be verified. Many who wrote exponential solutions in B received time outs; on occasion a low constraint is a red herring and hides an easily implemented more efficient solution.

Below is Mark Gordon's solution that implements the O(B^2N) algorithm described above.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <cstdio>
#include <cassert>
#include <map>

using namespace std;

#define MAXN 100010
#define GROUPS 8

int MB[MAXN][GROUPS];
int MF[MAXN][GROUPS];
int PS[MAXN][GROUPS];

int main() {
  freopen("fairphoto.in", "r", stdin);
  freopen("fairphoto.out", "w", stdout);

  int N, K; cin >> N >> K;
  vector<pair<int, int> > A(N);
  for(int i = 0; i < N; i++) {
    cin >> A[i].first >> A[i].second;
    A[i].second--;
  }
  sort(A.begin(), A.end());

  /* Construct backstep masks */
  for(int i = 0; i < GROUPS; i++) {
    MB[0][i] = 1 << A[0].second;
  }
  for(int i = 1; i < N; i++) {
    int bt = 1 << A[i].second;

    MB[i][0] = bt;
    for(int j = 1; j < GROUPS; j++) {
      if(MB[i - 1][j] & bt) {
        MB[i][j] = MB[i - 1][j];
      } else {
        MB[i][j] = bt | MB[i - 1][j - 1];
      }
    }
  }

  /* Construct forward step masks */
  for(int i = 0; i < GROUPS; i++) {
    MF[N - 1][i] = 1 << A[N - 1].second;
  }
  for(int i = N - 2; i >= 0; i--) {
    int bt = 1 << A[i].second;

    MF[i][0] = bt;
    for(int j = 1; j < GROUPS; j++) {
      if(MF[i + 1][j] & bt) {
        MF[i][j] = MF[i + 1][j];
      } else {
        MF[i][j] = bt | MF[i + 1][j - 1];
      }
    }
  }

  /* Construct partial sums */
  for(int i = 0; i < N; i++) {
    memcpy(PS[i + 1], PS[i], sizeof(PS[i]));
    ++PS[i + 1][A[i].second];
  }

  /* Compute the earliest starts for given masks and normalized partial sums.
*/
  map<vector<int>, int> cost_map;
  for(int i = N - 1; i >= 0; i--) {
    vector<int> V(1 + GROUPS);
    for(int j = K - 1; j < GROUPS; j++) {
      int base = -1;
      int m = V[GROUPS] = MF[i][j];
      if(__builtin_popcount(m) <= j) break;
      for(int k = 0; k < GROUPS; k++) {
        if(m & 1 << k) {
          if(base == -1) {
            base = PS[i][k];
          }
          V[k] = PS[i][k] - base;
        } else {
          V[k] = PS[i][k];
        }
      }
      cost_map[V] = A[i].first;
    }
  }

  /* Find best start points for each ending position. */
  int result = -1;
  for(int i = 0; i < N; i++) {
    vector<int> V(1 + GROUPS);
    for(int j = K - 1; j < GROUPS; j++) {
      int base = -1;
      int m = V[GROUPS] = MB[i][j];
      if(__builtin_popcount(m) <= j) break;
      for(int k = 0; k < GROUPS; k++) {
        if(m & 1 << k) {
          if(base == -1) {
            base = PS[i + 1][k];
          }
          V[k] = PS[i + 1][k] - base;
        } else {
          V[k] = PS[i + 1][k];
        }
      }

      map<vector<int>, int>::iterator it = cost_map.find(V);
      if(it != cost_map.end() && it->second < A[i].first) {
        result = max(result, A[i].first - it->second);
      }
    }
  }

  cout << result << endl;
  return 0;
}
```