USACO DEC12 Problem 'gangs' Analysis

by Mark Gordon

This problem was inspired from the online, constant space majority algorithm. There are several ways to solve this problem, including a linear time algorithm.

The (roughly) cubic algorithm relies on a procedure that given the state of the field and remaining cows can determine how many cows from gang 1 can be left at the end. Given the existence of that algorithm we can easily compute the lexicographically first solution by repeatedly appending the cow from the smallest gang that keeps this number constant. The code fragment below demonstrates how this might be done.

```
int cur_gang = 0;
int cur_cows = 0;
int res = max_cows(cur_gang, cur_cows, G);
if(res > 0) {
  cout << "YES\n" << res << "\n";
  for(int i = 0; i < N; i++) {
    /* Find the smallest gang to place next. */
    int prev_cur_gang = cur_gang;
    int prev_cur_cows = cur_cows;
    for(int j = 0; ; j++) {
      if(G[j] == 0) {
        continue;
      }

      G[j]--;
      update_state(cur_gang, cur_cows, j);
      if(max_cows(cur_gang, cur_cows, G) == res) {
        cout << j + 1 << '\n';
        break;
      }

      /* Placing gang j next didn't work out.  Undo the changes. */
      G[j]++;
      cur_gang = prev_cur_gang;
      cur_cows = prev_cur_cows;
    }
  }
} else {
  cout << "NO" << endl;
}
```

The intuition for the cubic algorithm is that we always can send gang 1 cows last. Additionally we can send all of the largest other gang of cows first and then repeatedly send a cow from the largest remaining gang. Indeed if we do this then we always end up with the maximum number of cows from gang 1 left. But why is this so?

Consider that maximizing the number of gang 1 cows at the end is the same as minimizing the number of other cows on the field prior to when we send the gang 1 cows.

Initially we fill up field with the largest gang. Then we send members of the other gangs until there are no cows on the field or we run out of cows to send. In the later case we've clearly done the best we can.

In the former case the difference between the two largest remaining gangs is at most 1. Therefore we can avoid sending a cow from the same gang twice in a row. This prevents more than one cow from ever being on the field. Since the parity of the number of cows on the field must equal the number of cows we've sent this is clearly the best we can do.

The linear algorithm is a bit trickier, and for brevity this article does not contain a complete proof of its correctness. First, when there are only two gangs the solution is always to send all of gang 1 first and then all of gang 2.

Otherwise first we compute g, the number of cows from gang 1 that can be present at the end. If A[n] gives the number of cows remaining in gang n then the optimal solution always begins by sending A[1] - g cows from gang 1 and ends by sending g cows from gang 1.

For the rest of the algorithm we set A[1] to 0 and maintain the invariant that

```
A + {GANG_ON_FIELD} * COWS_ON_FIELD has no majority element
```

That is, including the cows currently on the field, there is no gang that has a strict majority of the remaining cows. This means there are essentially three important gangs we might consider placing next: the smallest index gang, the largest gang, or the gang currently on the field. We only need to check to determine the smallest index of these that doesn't violate our invariant. The algorithm can be simplified even further by realizing that the gang currently on the field is either exhausted or falls into one of the other categories.

Below is my code for the simple O(N^3 log N) algorithm

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdio.h>

using namespace std;

void update_state(int& cur_gang, int& cur_cows, int cow_gang) {
  if(cur_cows == 0) {
    cur_gang = cow_gang;
  }
  if(cur_gang == cow_gang) {
    cur_cows++;
  } else {
    cur_cows--;
  }
}

/* Compute the number of cows from the first gang can be on the field at the
 * end. */
int max_cows(int cur_gang, int cur_cows, vector<int> G) {
  /* Keep trying to place the gang from the largest gang left. */
  sort(G.begin() + 1, G.end());
  while(G.back() > 0) {
    for(int i = G.size() - 1; i > 0; i--) {
      update_state(cur_gang, cur_cows, i);
      G[i]--;
      if(G[i - 1] <= G[i]) {
        break;
      }
    }
  }
  /* Finish by placing all of Bessie's gang. */
  for(int i = 0; i < G[0]; i++) {
    update_state(cur_gang, cur_cows, 0);
  }
  return cur_gang == 0 ? cur_cows : 0;
}

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

  int N, M; cin >> N >> M;
  vector<int> G(M);
  for(int i = 0; i < N; i++) {
    cin >> G[i];
  }

  int cur_gang = 0;
  int cur_cows = 0;
  int res = max_cows(cur_gang, cur_cows, G);
  if(res > 0) {
    cout << "YES\n" << res << "\n";
    for(int i = 0; i < N; i++) {
      /* Find the smallest gang to place next. */
      int prev_cur_gang = cur_gang;
      int prev_cur_cows = cur_cows;
      for(int j = 0; ; j++) {
        if(G[j] == 0) {
          continue;
        }

        G[j]--;
        update_state(cur_gang, cur_cows, j);
        if(max_cows(cur_gang, cur_cows, G) == res) {
          cout << j + 1 << '\n';
          break;
        }

        /* Placing gang j next didn't work out.  Undo the changes. */
        G[j]++;
        cur_gang = prev_cur_gang;
        cur_cows = prev_cur_cows;
      }
    }
  } else {
    cout << "NO" << endl;
  }
  return 0;
}
```