USACO FEB12 Problem 'coupons' Analysis

by Nathan Pinsker

There are several different ways to approach this problem. One of them stems from the initial idea of picking the lowest-cost cow each time: use all coupons on the cheapest cows, then buy as many cows as possible without coupons. However, this doesn't quite work: if several cows are very cheap with or without a coupon, and other cows are cheap with a coupon but very expensive without one, we can intuitively see that we would like to use our coupons on the more expensive cows. This leads to the idea of "revoking" a coupon: for cow i, we can pay (P_i - C_i) in order to regain one of our coupons (because we are now buying cow i at the "expensive" price). After purchasing as many cows as possible with coupons, we store their (P_i - C_i) values in a heap. To purchase a remaining cow j, we can either pay P_j or C_j + (P_i - C_i), where cow i is the top cow in our heap. This ensures we are always using exactly as many coupons as we can. For each cow we add to our lineup, we are greedily paying the minimum possible amount for it, so this solution is clearly optimal.

Bruce Merry's solution (implementing this idea) is below:

```
#include <fstream>
#include <algorithm>
#include <queue>
#include <vector>

using namespace std;

typedef long long ll;

struct pqitem
{
    ll value;
    int index;

    bool operator<(const pqitem &b) const
    {
        return value > b.value;
    }

    pqitem() {}
    pqitem(ll value, int index) : value(value), index(index) {}
};

int main()
{
    ifstream in("coupons.in");
    ofstream out("coupons.out");
    int N, K;
    ll M;
    in >> N >> K >> M;

    vector<ll> P(N), C(N);
```

```
    for (int i = 0; i < N; i++)
    {
        in >> P[i] >> C[i];
    }

    typedef priority_queue<pqitem> pqtype;
    priority_queue<ll, vector<ll>, greater<ll> > recover;
    pqtype cheap;
    pqtype expensive;
    for (int i = 0; i < K; i++)
        recover.push(0LL);
    for (int i = 0; i < N; i++)
    {
        cheap.push(pqitem(C[i], i));
        expensive.push(pqitem(P[i], i));
    }

    vector<bool> used(N, false);
    int nused = 0;
    while (M > 0 && nused < N)
    {
        while (used[cheap.top().index])
            cheap.pop();
        while (used[expensive.top().index])
            expensive.pop();

        if (recover.top() + cheap.top().value < expensive.top().value)
        {
            const pqitem top = cheap.top();
            ll cost = recover.top() + top.value;
            if (M < cost)
                break;
            M -= cost;
            recover.pop();
            recover.push(P[top.index] - C[top.index]);
            used[top.index] = true;
        }
        else
        {
            const pqitem top = expensive.top();
            ll cost = top.value;
            if (M < cost)
                break;
            M -= cost;
            used[top.index] = true;
        }
        nused++;
    }
    out << nused << "\n";
    return 0;
}
```