USACO DEC06 Problem 'fewcoins' Analysis

by Bruce Merry

Let's start by looking at just the problem of deciding how to make the change, given what Farmer John is going to pay. This is a fairly well-known dynamic programming problem, which can be solved by using the following recurrence: the best way to make amount C with coins 1..N is either to do it with coins 1..N-1, or to make change C-VN with coins 1..N and add a VN.

Working out the most efficient way for FJ to pay an amount M is slightly trickier, because he has a limited number of each coin. Instead, we can use the following recurrence: the best way to make amount M with coins 1..N is to use 0<=k<=CN coins of VN plus C-k.VN with coins 1..N-1, taking the minimum over all k.

Thus, once we know how much FJ is going to pay, we can compute the total number of coins required. Of course, we can run the DPs just once each (up to a maximum value) rather than for every amount that FJ might pay. The only catch is that we need to determine the maximum. Of course, FJ cannot pay more than all the coins in his possession, but it is possible to do better. Let V be the largest coin value, and call this type of coin silver and all other coins brass. It is not difficult to prove that for any change amount of at least V^2, it is possible to use at least one silver coin (hint: consider the partial sums modulo V), and hence that in making change of at least 2V^2 there can be at least V silver coins. Now if Farmer John pays at least 2V^2, and hence pays with V coins, then a similar argument shows that some set of them adds up to a multiple of V and at most V^2, which can be cancelled out with the silver coins in the change. So the change amount can never exceed 2V^2. In fact it is possible to show tighter bounds (such as V^2),but that is left as an exercise.

If implemented directly, this gives an algorithm that is O((T+V^2)NC), which at ~10^9 operations will most likely be too slow. One area that can easily be optimised is the inner-most loop, which determines the optimum number of coins of each type for FJ to use. It looks for the minimum of old[t - k*v] + k, where i is the total under consideration, v is the value of the coin under consideration and k ranges from 0 to c inclusive (c being the number of v's). If we define tilted[i%v][i/v] = old[i]-i/v then the expression we seek becomes min(tilted[t % v][t / v - k]) + t / v. By moving the offset outside of the minimum and splitting up the cases by modulus, we reduce the problem to one of finding the minimum from a range of values. What is worth noting is that the ranges we will use the same array multiple times, and each time the range we seek is just shifted over by 1. There exist some extremely sophisticated algorithms that provide constant-time range-min queries, but they are overkill for this specialised situation. Instead, we can preprocess the array to determine, for each i, the smallest j > i such that arr[j] < arr[i]. We can do this by working through the array, keeping a stack of the i values for which we have not yet seen any such j. Each time we see a new value j, we can finalise stack entries for which arr[i] > arr[j], before pushing j onto the stack (a similar algorithm can be used to solve Empodia from IOI 2004). To determine the smallest value in each range as we proceed, we simply keep track of the smallest index. If the previous smallest drops out of the window, start from the left of the window and keep following the "next smaller" pointer until the new smallest is found.

With this optimisation, the search becomes amortised constant time per query, so the efficiency is O((T+V^2)N).

```
#include <fstream>
#include <algorithm>
#include <set>
#include <stack>
#include <iterator>
#include <cassert>
#include <complex>

using namespace std;

#define MAXN 100
#define MAXT 20000
#define MAXV 200
#define MAXC 1000000

#define BIG (INT_MAX / 4)

typedef complex<int> addr;

int main() {
    ifstream in("fewcoins.in");
    ofstream out("fewcoins.out");
    int N, T, M;
    set<int> seen;

    in >> N >> T;
    assert(1 <= N && N <= MAXN);
    assert(1 <= T && T <= MAXT);

    int V[N], C[N], ans[N];
    for (int i = 0; i < N; i++) {
        in >> V[i];
        assert(1 <= V[i] && V[i] <= MAXV);
        assert(!seen.count(V[i]));
        seen.insert(V[i]);
    }
    for (int i = 0; i < N; i++) {
        in >> C[i];
        assert(0 <= C[i] && C[i] <= MAXC);
    }
    M = 0;
    for (int i = 0; i < N; i++)
        M += V[i] * C[i];
    assert(M >= T);
    int maxV = *max_element(V, V + N);
    M = min(M, T + maxV * maxV);
    M++;

    int dpf[M];
    int dpu[M], dpu_old[M];
    addr next[M];
    fill(dpf + 1, dpf + M, BIG);
    dpf[0] = 0;
```

```
    for (int i = 0; i < N; i++)
        for (int j = V[i]; j < M; j++)
            dpf[j] <?= dpf[j - V[i]] + 1;

    fill(dpu + 1, dpu + M, BIG);
    dpu[0] = 0;
    for (int i = 1; i <= N; i++) {
        copy(dpu, dpu + M, dpu_old);
        fill(next, next + M, addr(-1, -1));
        int v = V[i - 1];
        int c = C[i - 1];
        addr step(v, 1);
        addr delta = step * c;
        for (int b = 0; b < v; b++) {
            stack<addr> nonext;
            addr lo(b, 0);
            for (addr cur = lo; cur.real() < M; cur += step) {
                while (!nonext.empty()
                       && dpu_old[nonext.top().real()] - nonext.top().imag()
>
                                       dpu_old[cur.real()] - cur.imag()) {
                    next[nonext.top().real()] = cur;
                    nonext.pop();
                }
                nonext.push(cur);
                if (cur.real() - delta.real() > lo.real())
                    lo = cur - delta;
                while (next[lo.real()].real() != -1) lo = next[lo.real()];

                dpu[cur.real()] = dpu_old[lo.real()] - lo.imag() +
cur.imag();
            }
        }
    }

    int bestc = BIG;
    for (int i = T; i < M; i++)
        bestc <?= dpu[i] + dpf[i - T];
    out << (bestc < BIG ? bestc : -1) << "\n";

    return 0;
}
```