a OPEN12 BRONZE Cows in a Row

**Solution Notes:** This problem can be solved by "brute force", by simply trying to remove each possible cow ID from the line, checking after each one whether it gives the best answer (the longest consecutive block of equal cow IDs). Below is Travis Hance's solution using this idea. Although the running time of this method is O(N^2) (which is plenty fast for the limits in this problem), note that it is possible to solve the problem even faster, in only O(N) time. The idea behind the faster solution is to scan through the array in just one pass, remembering the two most recent distinct IDs you have seen, as well as a count of each one. For example, if the array is 31221254 and we are located at the third "2", then our current state will tell us that we have just scanned across three 2s and two 1s (giving a consecutive block size of 3, if we delete the 1s).

#include <cstdio>

int id[1005];

int get\_largest\_block(int n, int idignore) {

int maxBlockSize = 0;

int curBreed = -1;

int curSize = 0;

for(int i = 0; i < n; i++) {

if(id[i] != idignore) {

if(curBreed == id[i]) {

curSize++;

} else {

curBreed = id[i];

curSize = 1;

}

if(curSize > maxBlockSize)

maxBlockSize = curSize;

}

}

return maxBlockSize;

}

int main() {

freopen("cowrow.in","r",stdin);

freopen("cowrow.out","w",stdout);

int n;

scanf("%d", &n);

for(int i = 0; i < n; i++) {

scanf("%d", &id[i]);

}

int ans = 0;

for(int i = 0; i < n; i++) {

int size = get\_largest\_block(n, id[i]);

if(size > ans)

ans = size;

}

printf("%d\n", ans);

}

b OPEN12 SILVER Bookshelf (Silver)

**Solution Notes (Mark Gordon):** Suppose we decide that we want to place k books on the last bookshelf (and we've already checked that they can fit on one shelf). Then the height of the bookshelves is max(H[n-k+1], H[n-k+2], ..., H[n]) plus the cost of constructing shelves for the initial n-k books. This screams dynamic programming!

Let C(x) to be the minimum height of the bookshelves after placing the first x books (and let C(0) = 0). Then the initial observation allows us to construct the recurrence C(x) = min({C(y) + max(H[y+1], H[y+2], ..., H[x]) : 0 <= y < x and sum(W[y+1], W[y+2], ..., W[x]) <= L})

This immediately lends itself to an O(N^3) solution. By computing the max and sum terms as we iterate over y we can actually reduce the solution down to O(N^2) to look something like:

C(x) = 0

for i = 1 to N

hmax = 0

for j = i-1 to 0 step -1

wsum = wsum + W[j+1]

hmax = max(hmax, H[j+1])

if wsum <= L

C(x) = min(C(x), hmax + C(j))

However, for the gold version of the problem this is not enough. We'll use the same basic structure for the new solution but we'll be able to bring the runtime down to O(N log N) using two observations.

First, we can maintain which sections of the array correspond to different hmax terms (hmax from the solution above) efficiently as we iterate over the books. When we move past book i the set of positions with hmax H[i] is exactly those with hmax no larger than H[i] when we were at i-1. As hmax is non-increasing with j we can simply keep a list of the intervals of books with the same hmax and update the list by merging the intervals with hmax <= H[i] into our newly created interval. Because two indicies can never be 'unmerged' it follows that we can do at most N-1 merge operations over the course of the algorithm. We also must take care to erase indicies from the first interval if the wsum term gets too large.

The second observation is that C(x) is non-decreasing. That is, we can't get bookshelves of smaller height by adding more books. This means that we should only consider books that begin an interval formed in the previous step. Therefore each time we alter the set of intervals we erase any old costs of initial elements from a sorted set and insert any new costs. Then C(x) is simply the smallest element in this set. Below is my solution to this problem.

int W[MAXN];

int H[MAXN];

long long DP[MAXN];

int SA[MAXN];

int main() {

int N, L; cin >> N >> L;

int\* S = SA;

int rsz = 0;

int wsum = DP[0] = 0;

multiset<long long> bst;

for(int i = 1, j = 1; i <= N; i++) {

cin >> H[i] >> W[i];

for(S[rsz++] = 1; rsz > 1 && H[i - S[rsz - 1]] <= H[i]; rsz--) {

bst.erase(bst.find(H[i - S[rsz - 1]] + DP[i - S[rsz - 1] - S[rsz - 2]]));

S[rsz - 2] += S[rsz - 1];

}

bst.insert(H[i] + DP[i - S[rsz - 1]]);

wsum += W[i];

for(; wsum > L; j++) {

wsum -= W[j];

bst.erase(bst.find(H[j + S[0] - 1] + DP[j - 1]));

if(--S[0] == 0) {

++S; rsz--;

} else {

bst.insert(H[j + S[0]] + DP[j]);

}

}

DP[i] = \*bst.begin();

}

cout << DP[N] << endl;

}

c OPEN12 GOLD Tied Down

**Solution Notes (Bruce Merry):**

The first reaction may be to use the [winding number](http://en.wikipedia.org/wiki/Winding_number): if the rope has a non-zero winding number around a pole, then that pole obviously needs to be removed. However, this is not necessarily sufficient: in the sample case, the rope has a zero winding number around each pole, yet at least one pole must be removed. What's more, the sample case shows that testing each pole independently will not be good enough, because either pole can be left in place if the other is removed.

Thus, something smarter is required. The limit of 10 poles is a strong hint: we should have time to just try removing every subset, and then test whether the remaining poles leave Bessie free.

The geometry can get very messy, so let's try to simplify things. How can we use the fact that all the poles are in a line? Well, whatever happens entirely on one side of the line is irrelevant, since no matter how complex the shape it can always be straightened out without crossing any of the poles. So the only interesting segments are those that cross the line of poles. We can thus describe the rope by the sequence of points at which it crosses the pole line. The exact y coordinate is not relevant: only the position relative to the poles matters.

Now that we've simplified the representation of the rope, how can we move it around to release Bessie? Well, if the rope crosses at some point, and immediately afterwards crosses back at the same point, this is a loop that is not wrapped around any pole and can be pulled straight, making the two crossings disappear. If we represent each crossing point as a letter and the rope as a string (no pun intended) then this means we can delete any pair of adjacent letters that are the same e.g. abcxxdef -> abcdef. Conversely, we can of course insert two of the same letter anywhere into the sequence, but that turns out not to be useful.

Testing whether a particular set of poles allows Bessie to escape is now easy: take the starting string, and remove pairs of adjacent equal letters until either the string is empty (Bessie escapes) or there are no more pairs to remove (Bessie is stuck). This can be done in a single pass using a stack: each time a letter is seen, either pop from the stack if the new letter matches the top of the stack, or push the new letter onto the stack if not. The algorithm thus requires O(2^N.M) time.

#include <fstream>

#include <algorithm>

#include <complex>

#include <vector>

using namespace std;

typedef complex<int> pnt;

static int cross(const pnt &a, const pnt &b) { return imag(conj(a) \* b); }

static int cross(const pnt &a, const pnt &b, const pnt &c)

{

return cross(b - a, c - a);

}

int main()

{

ifstream in("tied.in");

ofstream out("tied.out");

int N, M;

int bx, by;

in >> N >> M >> bx >> by;

pnt B(bx, by);

vector<pnt> pnts(N);

for (int i = 0; i < N; i++)

{

int x, y;

in >> x >> y;

pnts[i] = pnt(x, y);

}

vector<pnt> rope(M + 1);

for (int i = 0; i <= M; i++)

{

int x, y;

in >> x >> y;

rope[i] = pnt(x, y);

}

int px = pnts[0].real();

vector<int> cuts;

for (int i = 0; i < M; i++)

{

if ((rope[i].real() > px) ^ (rope[i + 1].real() > px))

{

int c = 0;

for (int j = 0; j < N; j++)

if (cross(rope[i], rope[i + 1], pnts[j]) > 0)

c++;

if (rope[i + 1].real() > px)

c = N - c;

cuts.push\_back(c);

}

}

vector<int> st;

st.reserve(cuts.size() + 1);

int ans = N;

for (int b = 1; b < (1 << N); b++)

{

int G = 0;

vector<int> grp(N + 1);

grp[0] = 0;

for (int i = 0; i < N; i++)

{

if (b & (1 << i))

G++;

grp[i + 1] = G;

}

st.clear();

for (int i = 0; i < (int) cuts.size(); i++)

{

int g = grp[cuts[i]];

if (!st.empty() && g == st.back())

st.pop\_back();

else

st.push\_back(g);

}

if (st.empty())

ans = min(ans, N - \_\_builtin\_popcount(b));

}

out << ans << endl;

}

d OPEN12 GOLD Balanced Cow Subsets

**Solution Notes (Bruce Merry):**

This problem would be a relatively straightforward case of dynamic programming if it weren't for the very large limit on the amount of milk each cow produces. On the other hand, N is fairly small, so an exponential-time approach is likely to be feasible.

The obvious approach is to test every subset S in turn. If the sum of S is A, then try all subsets of S to see if one of them has a sum of A/2. Iterating over all subsets of all subsets takes 3^N steps, which is a little too large for the time available.

To reduce the time required, we can use a "meet-in-the-middle" approach. Split the cows evenly into two sets and paint one set brown and the other white. Suppose set S is balanced, and can be split into subsets A and B with the same sum. Then sum(brown in A) - sum(brown in B) = sum(white in B) - sum(white in A) i.e. the brown cows in S can be split into two sets and the white cows in S can also be split into two sets, where the degree of "unbalance" is the same. We can now try to reverse this process: for every subset of the brown cows, compute all the possible unbalance values and store them; similarly for the white cows. Then for each possible unbalance value, pair up all the brown subsets with that value with all the white subsets with that value to make balanced brown-and-white sets of cows.

What is the computational efficiency? The slow part is going to be the final matching. It's not obvious how to obtain a tight bound, but an approximation is to note that there are O(3^(N/2)) ways to partition subsets of the brown cows, and in the worst case one of these partitions may be matched up with all subsets of the white cows, giving O(3^(N/2).2^(N/2)) = O(6^(N/2)) time, which is good enough.

*Note from Neal Wu:*

Another optimization that can be made is when trying to match a brown subset with a white subset, one can choose the smaller of the two, iterate over every unbalance of that subset, and then look for that same unbalance in the other subset by doing a lookup into a hash table. This results in a faster but more complex to analyze runtime, which can be shown to be (1 + sqrt 1.5)^N, a bit better than (sqrt 6)^N.

#include <fstream>

#include <algorithm>

#include <map>

#include <vector>

#include <utility>

using namespace std;

typedef pair<int, int> pii;

typedef long long ll;

vector<pii> half(const vector<int> &S)

{

vector<pii> ans;

int N = S.size();

for (int i = 0; i < (1 << N); i++)

for (int j = i; ; j = (j - 1) & i)

{

int sum = 0;

for (int k = 0; k < N; k++)

{

if (j & (1 << k))

sum -= S[k];

else if (i & (1 << k))

sum += S[k];

}

if (sum >= 0)

ans.push\_back(pii(sum, i));

if (j == 0)

break;

}

sort(ans.begin(), ans.end());

ans.resize(unique(ans.begin(), ans.end()) - ans.begin());

return ans;

}

int main()

{

ifstream in("subsets.in");

ofstream out("subsets.out");

int N;

in >> N;

vector<int> SL, SR;

for (int i = 0; i < N; i++)

{

int x;

in >> x;

if (i & 1)

SL.push\_back(x);

else

SR.push\_back(x);

}

vector<pii> L = half(SL);

vector<pii> R = half(SR);

int p = 0;

int q = 0;

int LS = L.size();

int RS = R.size();

vector<bool> good(1 << N);

while (p < LS && q < RS)

{

if (L[p].first < R[q].first)

p++;

else if (L[p].first > R[q].first)

q++;

else

{

int p2 = p;

int q2 = q;

while (p2 < LS && L[p2].first == L[p].first)

p2++;

while (q2 < RS && R[q2].first == R[q].first)

q2++;

for (int i = p; i < p2; i++)

for (int j = q; j < q2; j++)

{

good[L[i].second | (R[j].second << SL.size())] = true;

}

p = p2;

q = q2;

}

}

int ans = count(good.begin() + 1, good.end(), true);

out << ans << endl;

}