USACO FEB10 Problem 'corral' Analysis

by Jelle van den Hooff

A first and very useful observation is that if a segment is completely covered by another segment, then we can simply discard it. In fact, solving the problem becomes a lot easier.

Once all covered segments are discarded, the remaining segments will look something like this:

```
[1111]  [333]          next[1] = 2    next[4] = 5
   [22222][444444]     next[2] = 3
           [55555555]  next[3] = 5
```

Now we call next[i] the segment that touches the i'th segment and extends farthest to the right. If the segments are sorted by starting point, then start[i] + length[i] ≤ start[i+1] + length[i+1]. This means that next[i] ≤ next[i+1]. So next[i+1] can be found by checking all possible segments after next[i] in linear time.

A warning: this all happens on a circle, so 'before' and 'after' become rather vague.

Assume we know one segment in the solution. Then the whole solution can be constructed by constantly adding next[last_segment] until we have a cycle, since next[last_segment] surely is the best possible segment that can come after last_segment.

If we pre-calculate next[i], next[next[i]], next[next[next[next[i]]]] (i.e., steps of size 2^n), we can simulate this process by trying every segment as starting point and then doing a binary search on the required number of steps and solve the problem in O(N log N).

My solution:

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

FILE *fin = freopen("corral.in", "r", stdin);
FILE *fout = freopen("corral.out", "w", stdout);

const int MAXM = 100000; // limits

int n, m; // input

struct seg {
  int s, e;
  bool operator<(const seg& o) const {
    if (s != o.s) return s < o.s;
    return e > o.e;
  }
} s[MAXM];
int next[MAXM][18]; // next[i][k] = optimal location after 2^k steps
                    // from i or - 1 if done

bool between(int a, int b, int c) { // cyclic a <= b < c
```

```
  if (a <= c) return a <= b && b <= c;
  else return a <= b || b <= c;
}

bool touch_right(int a, int b) {
  if (s[a].e < n) {
    if (b <= a) return false;
    return s[b].s <= s[a].e;
  } else {
    if (b > a) return true;
    return s[b].s <= s[a].e % n;
  }
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 0; i < m; i++) scanf("%d%d", &s[i].s, &s[i].e);

  for (int i = 0; i < m; i++) if (s[i].e == n) { // special case l=n
    printf("%d\n", 1);
    return 0;
  }

  for (int i = 0; i < m; i++) s[i].e += s[i].s;
  sort(s, s + m);

  // remove contained segments
  int t = m;
  m = 0;
  for (int i = 0; i < t; i++)
    if (m == 0 || s[i].e > s[m - 1].e)
        s[m++] = s[i];
  for (t = 0; s[t].e <= s[m - 1].e % n; t++)
    ;
  m -= t;
  for (int i = 0; i < m; i++)
      s[i] = s[i + t];

  t = 0;
  for (int i = 0; i < m; i++) {
    while (touch_right(i, (t + 1) % m))
      t = (t + 1) % m;
    if (t == i) {
      printf("-1\n");
      exit(0);
    }
    next[i][0] = t;
  }

  for (int k = 0; k < 17; k++) {
    for (int i = 0; i < m; i++) {
      int a = next[i][k];
      if (a == -1) {
        next[i][k + 1] = -1;
      } else {
        int b = next[a][k];
        if (b == -1 || between(i, b, a))       // done if i <= b <= a
           next[i][k + 1] = -1;
          else next[i][k + 1] = b;
      }
```

```
    }
  }

  int ans = m + 1;
  for (int i = 0; i < m; i++) {
    int c = next[i][0], x = 2;
    for (int k = 16; k >= 0; --k) {
      if (next[c][k] != -1 && !between(c, i, next[c][k])) {
        c = next[c][k];
        x += 1 << k;
      }
    }
    ans = min(ans, x);
  }

  printf("%d\n", ans);
  return 0;
}
```

Another solution looks at the graph that consist of edges i->next[i]. This graph can consist of multiple components, and each component has exactly one cycle (see Islands, IOI'08).

If an optimal solution uses any segment s in a given component, then every s in the cycle of that component can be used in an optimal solution:
An optimal solution that uses a segment s, can also use next[s]. So next[s] is contained in an optimal solution, and next[next[s]] too. Doing this long enough will always end in a cycle.

So we only need to check one starting point for each component (a segment in the cycle). This results in a linear solution, after sorting.

Implementation by David Benjamin:

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <algorithm>

// overflow shouldn't matter, but this is a little close
const long long MAXC = 1000000000 + 5;
const int MAXM = 100000 + 5;

struct edge {
    long long x, l;
    inline long long y() const { return x + l; }
};
edge edges[MAXM * 2];
bool dead[MAXM * 2];
long long c;
int m;

int next[MAXM * 2];

struct edge_cmp_x {
    bool operator() (const edge& a, const edge& b) {
        if (a.x != b.x)
            return a.x < b.x;
        return a.y() > b.y();
```

```
    }
};

// assumes everything already in [0, c)
inline bool contains(const edge& e, int x) {
    if (e.x <= x) {
        return e.y() >= x;
    } else { // e.x > x
        return e.y() >= (x + c);
    }
}

int start_from(int s) {
    int num = 1;
    int start_x = edges[s].x;
    // follow the next pointers until we get back to an edge that contains start_x
    do {
        num++;
        s = next[s];
    } while (!contains(edges[s], start_x));
    return num;
}

int rep_found[MAXM];
// find representative of cycle ending at s. use tok to uniquify among separate
// runs. returns -1 if representative already checked
int find_rep(int s, int tok) {
    while (!rep_found[s]) {
        rep_found[s] = tok;
        s = next[s];
    }
    if (rep_found[s] == tok)      // haha! I found a cycle
        return s;
    else                         // bah! someone else beat me to it
        return -1;
}

int main() {
    FILE * fin = fopen("corral.in", "r");
    FILE * fout = fopen("corral.out", "w");
    assert(fin != NULL); assert(fout != NULL);

    fscanf(fin, "%lld %d", &c, &m);
    for (int i = 0; i < m; i++)
        fscanf(fin, "%lld %lld", &edges[i].x, &edges[i].l);

    // sort by x and then decreasing length
    std::sort(edges, edges + m, edge_cmp_x());

    // duplicate edges to avoid nuisances with circular stuff
    for (int i = 0; i < m; i++) {
        edges[i + m] = edges[i];
        edges[i + m].x += c;
    }

    // delete all majorized edges
    int furthest = -1;
    for (int i = 0; i < m*2; i++) {
        if (edges[i].y() <= furthest)
            dead[i] = true;
```

```
        else
            furthest = edges[i].y();
    }
    int newm = 0;
    for (int i = 0; i < m; i++) {
        if (!dead[i] && !dead[i+m]) {
            edges[newm++] = edges[i];
        }
    }
    m = newm;

    // re-duplicate
    for (int i = 0; i < m; i++) {
        edges[i + m] = edges[i];
        edges[i + m].x += c;
    }

    // compute "next" edge
    memset(next, -1, sizeof(next));
    int nexti = 0; // the next "next" value to be finalized
    for (int i = 1; i < m*2; i++) {
        // the next edge is the last edge (x_next, y_next) such that x < x_next < y
        // we needn't check the first condition because the input should be a valid
cover
        while (edges[nexti].y() < edges[i].x) {
            next[nexti] = i-1;
            nexti++;
        }
    }

    // condense the duplicate thing
    for (int i = 0; i < m; i++) {
        // verify things match
        assert(next[i] >= 0 || next[i+m] >= 0);
        if (next[i] >= 0 && next[i+m] >= 0)
            assert((next[i] % m) == (next[i+m] % m));
        // set to whichever we got
        if (next[i+m] >= 0)
            next[i] = next[i+m];
        next[i] %= m;
    }

    // and now we hunt for the cycles and test representatives on each cycle
    int tok = 1;
    int best = m;
    for (int i = 0; i < m; i++) {
        int rep = find_rep(i, tok++);
        if (rep >= 0)
            best = std::min(best, start_from(rep));
    }

    fprintf(fout, "%d\n", best);
    return 0;
}
```