IOI'04

Enumerating Hurdles

September 16, 2004

1 The problem

In comparing two DNA sequences, biologists have found out that a powerful model is based on assigning integers to genes, then compare the number of "hurdles" (to be defined below) in each sequence. This approach was used in the 1980's to explain the genetic similarity between cabbage and turnip, in terms of evolutionary change.

Formally, we shall study permutations of the identity sequence $\sigma_I = \; 0\ 1\ 2\ \ldots\ N\ (N+1)\;$, such that 0 is always the first element, $(N+1)$ is always the last element and no two consecutive integers $\; x\ (x+1)\;$ exist. The central question is to find the number and the location of certain subsequences, called *hurdles*, which are defined below.

Define a *framed interval* of a sequence σ to be an interval of the form:

$$i\ \sigma_{j+1}\ \sigma_{j+2}\ \ldots\ \sigma_{j+k-1}\ i+k,$$

such that all integers between i and $i+k$ $(k \geq 1)$ belong to the interval $[i+1\ \ldots\ i+k-1]$. The *length* of the framed interval is the number of elements it contains including its endpoints. As a result, the length of the above framed interval is $k+1$. The *starting point* of the above framed interval is j, while the *finishing point* is $j+k$.

Hurdles are defined as framed intervals that contain no shorter framed intervals.

Your task is to write a program that, given a sequence of positive integers, finds the number of hurdles in the sequence. Your program shall also output the starting and finishing points of every hurdle.

Input: hurdles.in
The input is a permutation of the identity sequence σ_I that does not contain two consecutive integers in consecutive positions.

- The first line of the input contains the number of the elements that appear in the sequence, namely $N+2$.
- Each of the next $N+2$ lines of the input contains an integer that represents the value of the corresponding element of the sequence.

As an example, the input on sequence $\sigma_1 = \; 0\ 3\ 5\ 4\ 6\ 2\ 1\ 7\;$ would be:

Example input:
```
8
0
3
5
4
6
2
1
7
```

Output: hurdles.out
The first line of the output will contain a number that represents the number of hurdles, say M, found in the input sequence.

The next M lines will contain each a pair of numbers i, j, separated by a blank space, representing the starting and finishing points of each hurdle, subject to $i, j \quad \{1,\ N+2\}$. The hurdles must be sorted in increasing order of their starting points i.

The output of the above example should be:

Example output:
```
1
2 5
```
Now, let's see why the above output is correct. The entire input sequence σ_1 is a framed interval. Moreover, we have the framed interval 3 5 4 6, which can be reordered as 3 4 5 6. According to the definition, the interval 3 5 4 6 is a hurdle. This is the only hurdle in the above sequence: It has its starting point at index 2 and its finishing point at index 5.

2 Solution

Signed permutations are important in the study of DNA sequences [Ber01].

For each framed interval the following hold:
Fact 1: The starting (F_s) and finishing (F_f) point of a framed interval F are related with the following equation:

$$\text{Value } (F_f) = \text{Value } (F_s) + \text{length}(F) - 1 \tag{1}$$

Fact 2: All values in the interior points of a framed interval are distinct and cover the entire interval $[i+1, i+k-1]$ exactly once.

2.1 Simple

A simple algorithm is the following. For each element i of the sequence one exhaustively seeks for framed intervals of length $2 \leq l \leq (|\sigma| - i + 1)$ starting at element i, where $|\sigma| = N + 2$ is the number of elements contained in the sequence σ. For each pair (i, j) the corresponding length induces, someone inspects whether **Fact 1** holds or not. If it does, then according to **Fact 2** one must verify that all elements that lie in between are proper so that this interval is in fact a framed interval. We have $O(n^2)$ pairs (i, j), and for each interval these pairs induce, we require $O(n)$ time to verify that this is a framed interval whenever **Fact 1** holds. Thus, this part of the algorithm takes $O(n^3)$ time. During this part of the algorithm one keeps track of candidate hurdles starting at index i. Of course there is no need to keep track all of O(n) framed intervals that might exist starting at index i, since all framed intervals that don't have minimum length can not be hurdles.

Moreover, according to our definition of hurdles, the algorithm requires another part so as to eliminate all those framed intervals that contain shorter framed intervals starting at different points. At this final part of the algorithm we have $O(n)$ candidate hurdles. For each one of them one can verify in $O(n)$ time if another candidate hurdle is contained and decide whether this candidate hurdle is really a hurdle or not. This step of the algorithm takes $O(n^2)$ time.

As a result, the entire algorithm takes $O(n^3)$ time.

2.2 Clever

Fact 3: The element with the minimum value in a hurdle is always the starting point of the hurdle.

Fact 4: The element with the maximum value in a hurdle is always the finishing point of the hurdle.

Crucial observation: An important observation for the proposed algorithm is stated by the following lemma:

Lemma 2.1 *The endpoint(s) of a hurdle can not be in the interior of another hurdle.*

Proof

Case 1: *Both endpoints of a hurdle lie in the interior of another hurdle. Then obviously according to definition, one of the above framed intervals can not be a hurdle, which is a contradiction.*

Case 2: *Only one endpoint of a hurdle lies in the interior of another hurdle.*

Suppose for the purpose of contradiction that a hurdle H_2 has its starting point k in the interior of another hurdle H_1, a situation like the one shown in Figure 1. Then, based

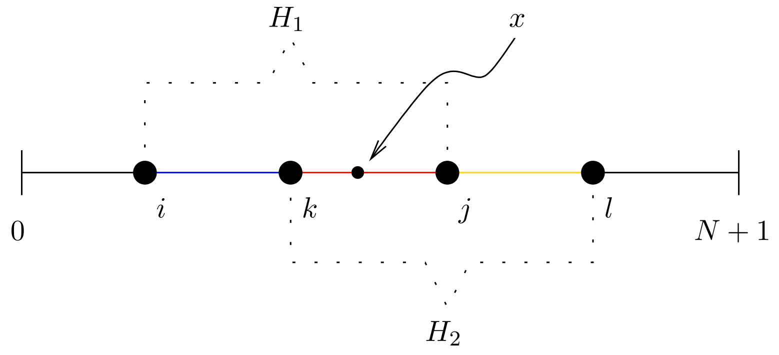

Figure 1: Proof for the second case.

on the preceding facts we have:

$$Value\ (i) < Value\ (k) < Value\ (j) < Value\ (l), \qquad (2)$$

since the k^{th} element is an interior point of hurdle H_1 and j^{th} element is an interior point of hurdle H_2. Now pick an element with index x at random such that $k < x < j$. Obviously, this element belongs to hurdle H_1, so Value $(x) <$ Value (j). Moreover, x belongs to hurdle H_2, so Value $(k) <$ Value (x). In other words, every element with index x, such that $k < x < j$, implies that:

$$Value\ (k) < Value\ (x) < Value\ (j). \qquad (3)$$

Finally, there is no other element y that implies (3) outside of the interval $[k+1, j-1]$, since if it were, then either hurdle H_1 would not be a hurdle, or H_2. As a result, all intermediate values lie in this interval (shown with red line) and as a consequence we have a framed interval. But this is a contradiction to our assumption that intervals $[i, j]$ and $[k, l]$ form two different hurdles. This completes the proof.

Note: In fact, in the above case we can prove something even stronger. Since all elements in the interval $[k+1, j-1]$ imply equation (3), all elements in the interval $[i, k]$ (blue line) imply that this region is also a framed interval since they belong to framed interval $[i, j]$. Finally, all the elements in the interval $[j, l]$ (yellow line) imply that this region is also a framed interval, since they belong to framed interval $[k, l]$.

So, a better way to handle this problem is to find a "most suitable" candidate framed interval for each element i. This can be achieved by scanning sequentially the elements of the given sequence from left to right and at the same time remembering the maximum value (max) found so far. Now, if someone finds an element x that has value Value$(x) = \max + 1$ and at the same time **Fact 1** is satisfied, it is easy to see that this implies the end of a framed interval. Since we have $O(n)$ elements to check and for each one of them we have $O(n)$ elements to scan, it is straightforward that in order to find all candidate framed intervals takes $O(n^2)$ time. On the following we present the algorithm in pseudocode assuming that the elements of the sequence are given in array SEQ. The algorithm returns array $FRAMED$, where it is stored the length of a candidate hurdle starting at position i.

```
CLEVER_CANDIDATE_ENUMERATION($SEQ[]$)
for $i \leftarrow 0$ to $N+1$
do $FRAMED[i] \leftarrow$ NIL
for $i \leftarrow 0$ to $N$
do $max \leftarrow SEQ[i]$
   for $k \leftarrow 1$ to $N+1-i$
   do if $((SEQ[i+k] = SEQ[i]+k)$ AND $(SEQ[i+k] = (max+1)))$
         then $FRAMED[i] \leftarrow k+1$
         else  if $(SEQ[i+k] < SEQ[i])$
                  then break
                  else  if $(SEQ[i+k] > max)$
                           then $max \leftarrow SEQ[i+k]$
return $FRAMED$
```

As in the previous algorithm, we can perform the elimination of false alerts in $O(n^2)$ time.

2.2.1 Elimination of false alerts in linear time

Based on the preceding lemma one can easily decide which of the candidate hurdles are in fact hurdles and not false alerts in linear time. One must keep track of the length of each candidate hurdle at its starting position in an array FRAMED. We say that a candidate hurdle is *active* if we examine elements in the interior of the hurdle.

The idea is the following. One scans sequentially the array FRAMED, and if he encounters a starting position of a candidate hurdle H_2 while a candidate hurdle H_1 is active, then case 1 of lemma 2.1 must apply because the candidate hurdle H_1 is a framed interval of minimum length starting at the starting position of candidate hurdle H_1. This process requires no more than $N+1$ queries on array FRAMED and as a result its running time is $O(n)$. The elimination algorithm is given with the following pseudocode:

```
Linear_Elimination($\mathit{FRAMED}$[])
$i \leftarrow 0$
while ($i < N+1$)
do if ($\mathit{FRAMED}[i] = $ NIL)
      then $k \leftarrow i+1$
            while ($k < \mathit{FRAMED}[i] + i - 1$)
            do if ($\mathit{FRAMED}[k] = $ NIL)
                  then $\mathit{FRAMED}[i] \leftarrow$ NIL
                        break
               $k \leftarrow k+1$
            $i \leftarrow k-1$
   $i \leftarrow i+1$
return $\mathit{FRAMED}$
```

Finally, either way we choose to eliminate false alerts, this algorithm has running time $O(n^2)$.

2.3 Advanced

This problem can be solved in superlinear $O(n\alpha(n))$ [BH96] and linear ($O(n)$) [BMY01] time if someone uses graphs. These results come from one of the most exciting areas in computer science in the last decade which was pioneered by Hannenhalli and Pevzner [HP95] and have rather simple implementations.

A detailed treatment of the field that inspired this problem can be found at [Sie01].

References

[Ber01] Anne Bergeron. A Very Elementary Presentation of the Hannenhalli-Pevzner Theory. *CPM 2001 - Lecture Notes in Computer Science*, 2089:106–117, April 19 2001.

[BH96] Piotr Berman and Sridhar Hannenhalli. Fast Sorting by Reversal. *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 168–185, 1996.

[BMY01] Bader, Moret, and Yan. A Linear-Time Algorithm for Computing Inversion Distance between Signed Permutations with an Experimental Study. In Lecture Notes in Computer Science, editor, *WADS: 7th Workshop on Algorithms and Data Structures*, volume 2125, pages 365–376, 2001.

[HP95] Sridhar Hannenhalli and Pavel Pevzner. Transforming Cabbage into Turnip. *Proceedings of the 27th Annual Symposium on Theory of Computing (STOC95)*, pages 178–189, 1995.

[Sie01] Adam C. Siepel. Exact Algorithms for the Reversal Median Problem. Master's thesis, University of New Mexico, December 2001.

[Sie02] Adam C. Siepel. An Algorithm to Enumerate All Sorting Reversals. *RECOMB*, pages 281–290, April 2002.

The Farmer

Let g_i be a field or a strip. Denote by n_i the number of cypresses in a field or a strip. If we denote by e_i the number of olive trees in a g_i, we have: $e_i = n_i$ if g_i is a field or $e_i = n_i - 1$ if g_i is a strip.

Consider now the following KNAPSACK problem: max $\sum_{i=1}^{n+m} e_i x_i$, such that $\sum_{i=1}^{n+m} n_i x_i \leq Q$ and $x_i \in \{0,1\}$, where n,m are the numbers of fields and strips respectively.

An optimum solution x_i^*, $1 \leq i \leq n+m$, of this problem consists of a subset of g_i such that the total number of their cypresses is at most Q and the total number of the included olive trees is maximized. However in general $Q' = \sum_{i=1}^{n+m} n_i x_i < Q$. If this is the case, then there is some g_i such that $x_i^* = 0$ and $n_i > Q - Q'$, for otherwise the optimum solution can be improved by the inclusion of g_i, a contradiction. Therefore, adding a chain of $Q - Q'$ cypresses of g_i and its $Q - Q' - 1$ olive trees to the optimum solution of the knapsack problem above, yields to an optimum solution. The KNAPSACK problem can be solved optimally in $O\big((n+m)Q\big)$ time by a Dynamic Programming algorithm.

Use Dynamic Programming:

Let $a(x, y)$ be the wasted area for a rectangle (x, y), $1 \leq x \leq W$, $1 \leq y \leq H$. Initially, put $a(x, y) = xy$, for all (x, y) except for the ones corresponding to needed plates, e.g. $x = w_i$ and $y = h_i$, $1 \leq i \leq N$, for which we put $a(x, y) = 0$. For a plate (x, y) consider all vertical cuts $c = 1, 2, \ldots, x - 1$ and all horizontal cuts $c = 1, 2, \ldots, y - 1$ and chose the cut producing the minimum wasted area $a(x, y) = a(c, y) + (x - c, y)$ or $a(x, c) + a(x, y - c)$ for some c.