Jyrki Nummenmaa and Erkki Mäkinen

2.4 Depot (a competition task)

Problem

A Finnish high technology company has a big rectangular depot. The depot has a worker and a manager. The sides of the depot, in the order around it, are called left, top, right and bottom. The depot area is divided into equal-sized squares by dividing the area into rows and columns. The rows are numbered starting from the top with integers 1, 2,... and the columns are numbered starting from the left with integers 1, 2,...

The depot has containers, which are used to store invaluable technological devices. The containers have distinct identification numbers. Each container occupies one square. The depot is so big, that the number of containers ever to arrive is smaller than the number of rows and smaller than the number of columns. The containers are not removed from the depot, but sometimes a new container arrives. The entry to the depot is at the top left corner.

The worker has arranged the containers around the top left corner of the depot in such a way that he will be able to find them by their identification numbers. He uses the following method. Suppose that the identification number of the next container to be inserted is *k* (container *k*, for short). The worker travels the first row starting from the left and looks for the first container with identification number larger than *k*. If no such container is found, then container *k* is placed immediately after the rightmost of the containers previously in the row. If such a container *l* is found, then container *l* is replaced by container *k*, and *l* is inserted to the following row using the same method. If the worker reaches a row having no containers, the container is placed in the leftmost square of that row.

Suppose that containers `3,4,9,2,5,1` have arrived to the depot in this order. Then the placement of the containers at the depot is as follows.

```
1 4 5
2 9
3
```

The manager comes to the worker and they have the following dialogue:
Manager: Did container `5` arrive before container `4`?
Worker: No, that is impossible.
Manager: Oh, so you can tell the arrival order of the containers by their placement.
Worker: Generally not. For instance, the containers now in the depot could have arrived in the order `3,2,1,4,9,5` or in the order `3,2,1,9,4,5` or in one of 14 other orders.

As the manager does not want to show that the worker seems much smarter, he goes away. You are to help the manager and write a program which, given a container placement, computes all possible orders in which they might have arrived.

Input

The input file name is `depot.in`. The first line contains one integer *R*: the number of rows with containers in them. The following *R* lines contain information about rows 1,..., *R* starting from the top as follows. First on each of those lines is an integer *M*, the number of containers in that row. Following that, there are *M* integers on the line: the identification numbers of the containers in the row starting from the left. All container identification numbers *I* satisfy $1 \leq I \leq 50$. Let *N* be the number of containers in the depot, then $1 \leq N \leq 13$.

Output

The output file name is `depot.out`. The output file contains as many lines as there are possible arrival orders. Each of these lines contains *N* integers, the identification numbers of the containers in the potential arrival order described by that line. All lines describe an arrival order not described in any other line.

Example inputs and outputs

Example 1: depot.in

```
3
3 1 4
5
2 2 9
```

depot.out

```
3 2 1 4 9 5
3 2 1 9 4 5
3 4 2 1 9 5
3 2 4 1 9 5
3 2 9 1 4 5
3 9 2 1 4 5
3 4 2 9 1 5
3 4 9 2 1 5
3 2 4 9 1 5
3 2 9 4 1 5
3 9 2 4 1 5
3 4 2 9 5 1
3 4 9 2 5 1
3 2 4 9 5 1
3 2 9 4 5 1
3 9 2 4 5 1
```

Example 2: depot.in

```
2
2 1 2
1 3
```

depot.out

```
3 1 2
1 3 2
```

Scoring

If the output file contains impossible orders or no orders at all, your score is 0 for that test case. Otherwise the score for a test case is computed as follows. If the output file contains all possible orders exactly once, your score is 4. If the output file contains at least half of the possible orders and each of them exactly once, your score is 2. If the output file contains less than half of the possible orders or some of them appear more than once, your score is 1.

Solutions

With no loss of generality, we may assume that the containers are numbered from 1 to N. The following algorithm can be used to solve the problem for small inputs.

Algorithm GenerateAndTestDepot

Generate all permutations of numbers. For each permutation, generate the placement of containers in the depot using the method used by the worker and compare the placement with the input placement. Output all permutations, for which the generated placement is the same as the input placement.

Lemma 1. GenerateAndTestDepot solves the Depot problem.

Proof. Not a very interesting one...

However, GenerateAndTestDepot only works with fairly small numbers of N. The rest of our solutions are based on the following idea. Make successive removals, recovering the state of the depot as if the removed item had been the last inserted item. Recovering the depot might involve pulling up items from rows below (as opposed to pushing them down when inserting items).

Algorithm BacktrackingDepot

For all items on the top row, call RecursiveDelete(inputdepot, item, solution) with empty solution.

Procedure RecursiveDelete(depot, item, solution)

If the item is the last item left in the depot, add it to the solution and output solution.

If the item is not the last item left in the depot, then for all items on the first row, delete the item from the depot, add it to the solution, and call RecursivePullUp(depot, item, 1,solution)

Procedure RecursivePullUp(depot, item, solution)

If we are on the last row, then we just find all the items based on their values, which can replace the item on the previous row. For each such item found, remove that item and call RecursiveDelete(depot, newitem, solution) for all newitem values on the first row. (This is equal to finishing pulling up and continuing recursion.)

If we are not on the last row, then we similarly find all the items based on their values, which can replace the item on the previous row. For each such item found, call RecursivePullUp(depot, newitem, solution) recursively.

Lemma 2. BactrackingDepot solves the Depot problem.

Proof. BacktrackingDepot tries out every possible way to delete all containers from the depot. That's it.

An unfortunate thing with BactrackingDepot is that it may be highly inefficient. In fact, it favours inputs where there are lots of short rows (optimally N rows with 1 item). However, with an input file with just one row of N items, the solution is no more efficient than GenerateAndTestDepot (as a matter of fact, due to implementational overhead, it is likely to be even slower than GenerateAndTestDepot). One obvious optimisation is readily available: As soon as there is only one row left, we can form the rest of the solution in linear time, as the items must have been inserted in the order where they are, as

otherwise some item would have pushed some other item down. Already this in practice speeds the process up considerably.

Even with this optimisation, it is clear that the solution is inefficient. However, RecursivePullUp seems to be doing much extra work to what is really needed. A little investigation reveals the following lemma.

Lemma 3. Assume d is a depot with N - 1 items in it. Consider pushing down a value y from row r to row r + 1 with the insertion of value x to row r while computing a depot with N items. Then, clearly, $x < y$ and for the value y' previous to y on row r before pushing, it holds that $y' < x$.

Proof. Obvious from the way insertions are done.

From Lemma 3 we get the following lemma.

Lemma 4. Assume d is a depot with N items in it. Consider pulling up values from row r + 1 to row r, while computing a depot with N - 1 items. Then, each value on row r + 1 can only be considered when finding the replacement for exactly one of the items on the row above.

Proof. Follows from Lemma 3.

With Lemma 4 we can slim down the search considerably. If we keep indexing for the depot showing which items can be pulled up from the row below for each item on each row, and only study those, then we speed up the search considerably. However, with smallish values of N (under 15) the optimisation does not really pay off, since the rows are so short. The same applies for using some asymptotically more efficient data structure for arranging the rows. Another algorithm is available using the following lemma.

Lemma 5. The last item to be positioned when an item is inserted is going to end up at the end of a row, and the row above it must not be shorter than the row below.

Proof. Should be quite straightforward to see.

Lemma 6. Any one of the items at the end of a row, where the row above is not equally long, may have been added last.

Proof. Follows quite easily using Lemma 5.

This suggests a similar search to the one we have proposed before, but starting from the bottom.

The advantage is that we only start from such items that they can be moved up reversing an insertion, as opposed to the other method starting from the top, where we might start with an item which can not be the last item inserted.

Background

This problem was motivated by the theory of tableaux and permutations. The depot represents a Young tableau. A reader interested in the subject is encouraged to study the area from Knuth's book [1]. However, Knuth does not treat the problem of computing the permutations, which yield a given Young tableau. In his book a classical result is given, where it is said that there exists n^2 different forms for a tableau with n items. From Lemma 6 and this it follows that there are only n^2 possible numbers of solutions. Gyula Horvath [2] has studied these numbers empirically further than the authors, and suggested a variation of the task, where we only ask for the containers that could have arrived first.

We thank Isto Aho for helpful discussions, and Tero Karras, Janne Kujala, and Samuli Laine for test solving the task, and helping the authors to see some of the properties of the problem.

References

[1] Donald E. Knuth, *The Art of Computer Programming* , Vol. 3, Addison-Wesley, 1973.

[2] Gyula Horvath, private communication, 2001.