POLISH OLYMPIAD IN INFORMATICS

POLISH MINISTRY OF NATIONAL EDUCATION AND SPORT

UNIVERSITY OF INFORMATION TECHNOLOGY
AND MANAGEMENT IN RZESZÓW

INSTITUTE OF INFORMATICS, WARSAW UNIVERSITY

The 11th Central European Olympiad in Informatics

CEOI 2004

Rzeszów, Poland

Tasks and Solutions

Edited by Marcin Kubica

WARSAW, 2004

POLISH OLYMPIAD IN INFORMATICS

POLISH MINISTRY OF NATIONAL EDUCATION AND SPORT

UNIVERSITY OF INFORMATION TECHNOLOGY
AND MANAGEMENT IN RZESZÓW

INSTITUTE OF INFORMATICS, WARSAW UNIVERSITY

The 11th Central European Olympiad in Informatics

CEOI 2004

Rzeszów, Poland

Tasks and Solutions

Edited by Marcin Kubica

WARSAW, 2004

Authors:

Michał Adamaszek
Tomasz Czajka
Łukasz Kowalik
Marcin Kubica
Marcin Michalski
Anna Niewiarowska
Paweł Parys
Jakub Pawlewicz
Jakub Radoszewski
Rafał Rusin
Wojciech Rytter
Krzysztof Sikora
Piotr Stańczyk

Proofreaders: Piotr Chrząstowski-Wachtel, Marcin Kubica

Volume editor: Marcin Kubica

ISBN 83–917700–6–0

Contents

Preface

Central European Olympiad in Informatics (CEOI) gathers the best teen-age programmers from several European countries. The 11-th CEOI was held at University of Information Technology and Management in Rzeszów, Poland, July 12–18, 2004. It was organized by Polish Olympiad in Informatics and University of Information Technology and Management in Rzeszów, together with Polish Ministry of National Education and Sport, and Institute of Informatics, Warsaw University.

Eight countries took part in the competition: Bosnia and Herzegovina, Croatia, Czech Republic, Germany, Hungary, Poland, Romania and Slovakia. All of the countries, except Poland, were represented by 4 contestants. Poland was represented by one official team and two additional teams (not ranked officially), each team consisting of 4 contestants. More information about this competition, including the results, can be found at `http://www.oi.edu.pl/ceoi2004`.

The contest consisted of three sessions: a trial session and two competition sessions. During the trial session contestants had an occasion to become familiar with software environment and to solve a preparation task 'Journey'. During each of the competition sessions they had to solve three tasks within five hours of time: 'Clouds', 'Sweets' and 'Trips' during the first session, and 'Football league', 'Puzzle' and 'Two sawmills' during the second one. The time limits given in the task descriptions refer to Pentium 4, 2.4 GHz machines.

Four contestants were awarded gold medals, five contestants were awarded silver medals and eight contestants were awarded bronze medals. The results are as follows:

- Gold medalists:

Luka Kalinovćić	Croatia
Filip Wolski	Poland
Lovro Puzar	Croatia
Bartłomiej Romański	Poland

- Silver medalists:

Jakub Łącki	Poland
Alexandru Mosoi	Romania
Erik Panzer	Germany
Thomas Fersch	Germany
Dan-Constantin Spatarel	Romania

- Bronze medalists:

Peter Perešíni	Slovakia
Sorin Stancu-Mara	Romania
Tomasz Kuras	Poland
Gergely Tassy	Hungary
Mircea Adrian Digulescu	Romania
Balázs Kormányos	Hungary
Michal Poláčik	Slovakia
Daniel Marek	Czech Republic

This booklet presents tasks from CEOI'2004 together with the discussion of their solutions. Some more materials, including test data used during the evaluation and example solutions, can be found at `http://www.oi.edu.pl/ceoi2004`. It was prepared for the contestants of various programming contests to help them in their exercises. It should also give a taste of coming 17th International Olympiad in Informatics, which will be held in Poland, Nowy Sącz, in 2005.

Marcin Kubica

Task by **Jakub Radoszewski**	Solution description by **Tomasz Czajka, Anna Niewiarowska**
Available memory: 32 MB.	**Maximum running time: 0.5 s.**

Journey

There are n cities in Byteland (numbered from 1 to n), connected by bidirectional roads. The king of Byteland is not very generous, so there are only $n-1$ roads, but they connect the cities in such a way that it is possible to travel from each city to any other city.

One day, a traveler Byterider arrived in the city number k. He was planning to make a journey starting in the city k and visiting on his way cities m_1, m_2, ..., m_j (not necessarily in this order) — the numbers m_i are all different and they are also different from k. Byterider — like every traveler — has only a limited amount of money, so he would like to visit all the cities that he has planned to visit using the shortest possible path (starting in the city k). A path is one road or a sequence of roads, where every next road begins in the city where the previous one ends. Help Byterider to determine the length of the shortest path for his journey.

Task

Write a program that:

- reads from the standard input:
  - the description of the roads connecting the cities of Byteland,
  - the number of the city where Byterider arrived,
  - a list of cities which Byterider would like to visit,
- determines the minimum length of Byterider's journey,
- writes the result to the standard output.

Input

The first line of the standard input contains two integers n and k separated by a single space ($2 \leq n \leq 50\,000$, $1 \leq k \leq n$), n is the number of cities in Byteland and k is the number of the first city on Byterider's path. Each of the following $n-1$ lines contains the description of one road in Byteland. Line $(i+1)$ (for $1 \leq i \leq n-1$) contains three integers a_i, b_i and d_i separated by single spaces ($1 \leq a_i, b_i \leq n$, $1 \leq d_i \leq 1\,000$), a_i and b_i are the cities connected by the road, and d_i is the length of the road. Line $(n+1)$ contains one integer j — the number of cities which Byterider would like to visit ($1 \leq j \leq n-1$). The next line contains j different integers m_i separated by single spaces — the numbers of the cities that Byterider would like to visit ($1 \leq m_i \leq n$, $m_i \neq k$).

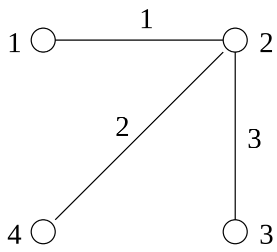
Output

The first and only line of the standard output should contain exactly one integer: the length of the shortest path for Byterider's journey.

Example

For the input data:

```
4 2
1 2 1
4 2 2
2 3 3
2
1 3
```



the correct result is:

```
5
```

Solution

It can be easily seen that the situation from the task can be considered as an undirected graph G, in which the cities are vertices and the roads are edges. The graph G has n vertices, $n-1$ edges and it is connected (there is a path between each pair of vertices). It can be easily noticed that the graph G is a tree.

Let us make the vertex k (which represents the city the traveler starts from) the root of the tree. The vertices $m_1 \ldots m_j$ represent the cities the traveler wants to visit.

Algorithm

The problem can be solved using the following algorithm:

- Build a graph (tree) data structure using adjacency lists.
- Traverse the tree depth-first, taking the starting city k as a root. For each subtree recursively determine:
  - if there are any cities to visit in the subtree at all,
  - length of the shortest path starting and ending in root and visiting the given cities,
  - length of the shortest path starting (but not necessarily ending) in the root and visiting the given cities.

These values can easily be back-propagated up the tree.

- The third value in the root is the result.

Proof of correctness

Let us call a *correct path* in a subtree $H = (V_H, E_H) \subseteq G$ — a path in H that visits all of the vertices $\{m_1 \ldots m_j\} \cap V_H$.

In the shortest correct path we will not use one road more than twice. To prove it, let us suppose that in the optimal solution we use the road x at least 3 times. The path looks like:

$$a_1 \ldots a_n, x, b_1 \ldots b_n, x, c_1 \ldots c_n, x, d_1 \ldots d_n$$

If we change the path like this:

$$a_1 \ldots a_n, c_1 \ldots c_n, x, b_1 \ldots b_n, d_1 \ldots d_n$$

we will visit the same cities using a shorter path. It follows that the previous path is not optimal. Therefore in the optimal path each road will be used no more than twice.

It also means that using the shortest correct path we will enter each subtree no more than once. If a subtree does not contain any of the elements $\{m_1 \ldots m_j\}$, we will not enter that subtree at all.

If we want to find the shortest correct path in a tree starting in the root k and ending also in k, we start with an empty path p and for each of the k's children v we have to repeat the following steps:

- Check if we have to enter the subtree with root in v.
- If we have to enter that subtree:
  - we add to the path p the edge from k to v,
  - we recursively find the shortest correct path that begins in v and ends in v and we add that path to p,
  - we add to the path p the edge from v to k.

It is easy to check, that we can choose k's children in any order and it does not change the path's length.

Let us suppose that we want to find the shortest correct path starting in the root k and ending in any vertex. Let us denote by $k_1 \ldots k_l$ — the children of k which we have to visit. We start with an empty path p and apply the following steps:

- Choose which of the vertices $k_1 \ldots k_l$ should be visited last. Let us call that vertex w.
- For each $v \in \{k_1 \ldots k_l\}$ such that $v \neq w$:

  - add to the path p the edge from k to v,
  - recursively find the shortest correct path that begins in v and ends in v and add that path to p,
  - add to the path p the edge from v to k.

- Add to the path p the edge from k to w.
- Recursively find the shortest correct path that begins in w and ends in any vertex and add that path to p.

As in the previous task, the order of the first $l-1$ vertices does not change the path's length. The only problem is to find the vertex k_i that should be visited as the last one. But this task can also be solved easily. Suppose that for each of k's children v, if v has to be entered, we know the lengths of the paths:

- $l_{v,1}$ — the length of the path that is the sum of: the edge from k to v, the shortest correct path (in a subtree with a root in v) that begins in v and ends in v, the edge from v to k,
- $l_{v,2}$ — the length of the path that is the sum of: the edge from k to v, the shortest correct path (in a subtree with a root in v) that begins in v and ends in any vertex.

We can easily find the vertex that should be entered as the last one — that is the one with the biggest value of $l_{v,1} - l_{v,2}$.

Knowing the above facts, we can modify our algorithm of finding the shortest correct path that begins in the root k and ends in any vertex in the following way:

For each v being a child of k, we recursively find:

- whether we have to visit the subtree beginning in v or not (we have to visit it, if any of the vertices $m_1 \ldots m_j$ is in that subtree),
- the length of the shortest correct path in the subtree that begins and ends in v,
- the length of the shortest correct path in the subtree that begins in v and ends in any node,
- the values $l_{v,1}$ and $l_{v,2}$.

These values can be easily back-propagated up the tree. If we know all the above information, we can easily choose:

- which subtrees should be visited,
- which one of them should be visited last,

and so for the subtree with a root in k we can find:

- whether we have to visit that subtree (if we have to visit any of the subtrees with root in v's child or if $v \in \{m_1 \dots m_j\}$),
- the length of the shortest correct path in this subtree that begins and ends in v (the sum of $l_{v_i,1}$ for all the children v_i of k that should be visited),
- the length of the shortest correct path in the subtree that begins in v and ends in any node (the previous sum decreased by $l_{w,1} - l_{w,2}$, where w is the last k's child to be visited).

Task by	Solution description by
Jakub Radoszewski	**Łukasz Kowalik, Piotr Stańczyk**
Available memory: 64 MB.	**Maximum running time: 3 s.**

Clouds

There are n clouds in the sky, all moving continuously with a wind in the same direction and with the same constant velocity $v = (v_x, v_y)$, i.e. for any real number $t \geq 0$ and any point of a cloud with initial coordinates (x, y) the position of this point at time t is $(x + t * v_x, y + t * v_y)$.

For the sake of simplicity, we assume that every cloud is a polygon (containing its periphery), whose vertices have integer coordinates. This polygon does not have to be convex, but no two of its edges cross each other (with exception of the common endpoints of consecutive edges). The clouds may intersect.

On the ground there is a satellite control center, at coordinates $(0, 0)$, and there is a satellite directly above the control center and above the clouds. Straight upwards from the control center to the satellite goes a laser beam. The laser beam is used to communicate with the satellite. However, when the beam crosses a cloud, the communication is not possible. Initially, the beam does not cross any cloud. During the observation, there may be several moments, when the laser beam crosses (one or more) clouds, interrupting the communication. Even when the laser beam crosses a single vertex of a cloud, the communication is interrupted for an instant. You are to write a program that computes how many times the communication is interrupted, until all the clouds drift away.

Task

Write a program, that:

- reads from the standard input the positions, shapes and velocity of the clouds,
- determines how many times communication is interrupted,
- writes the result to the standard output.

Input

The first line of input contains three integers n, v_x and v_y, separated by single spaces, $1 \leq n \leq 1\,000$, $-1\,000\,000\,000 \leq v_x, v_y \leq 1\,000\,000\,000$; n is the number of clouds, $v = (v_x, v_y)$ is the velocity vector of the clouds ($v \neq (0, 0)$). The x-coordinates correspond to the West-East direction and the y-coordinates correspond to the North-South direction.

The following n lines contain descriptions of the clouds, one cloud per line. Each of these lines consists of a sequence of integers, separated by single spaces. The first integer in a line is the number of cloud's vertices k, $3 \leq k \leq 1\,000$. It is followed by $2k$ integers: $x_1, y_1, x_2, y_2, \ldots x_k, y_k$, $-1\,000\,000\,000 \leq x_i, y_i \leq 1\,000\,000\,000$; $(x_1, y_1), (x_2, y_2), \ldots (x_k, y_k)$ are the coordinates of the consecutive vertices in a clockwise order.

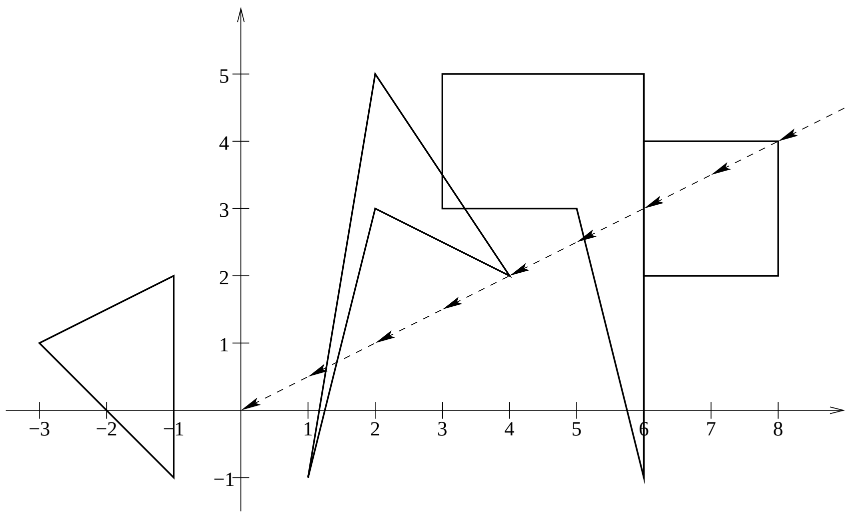
The laser beam crosses clouds peripheries at most $100\,000$ times.

Output

The first and only line of output should contain exactly one integer: the number of times the communication is interrupted.

Example

For the input data:
```
4 -2 -1
4 6 2 6 4 8 4 8 2
4 2 3 1 -1 2 5 4 2
3 -3 1 -1 2 -1 -1
5 5 3 3 3 3 5 6 5 6 -1
```
the correct result is:
```
3
```



The figure shows the clouds seen from above. The dashed line marks the points that will cross the laser beam.

Solution

The task Clouds is one of many geometric problems in which all correct solutions (abstracting from implementation) are based on a similar idea. It is easy to see that in order to calculate how many times communication is interrupted one has to focus on points in which the laser beam crosses clouds' peripheries — these points are important for the problem given, as they are the only places where communication may change its state.

In our task there is a laser beam at position $(0, 0)$ and a set of clouds moving at the velocity described by vector (v_x, v_y). However we know from physics that the motion is relative. Thus there will be no difference if we assume that the clouds stay at their initial positions and the laser beam is moving at the velocity $(-v_x, -v_y)$. The latter formulation is more convenient and it will be used in the sequel. Let us draw a semi-line with the end in point $(0,0)$ representing the track of the laser beam in time (see Fig. 1).

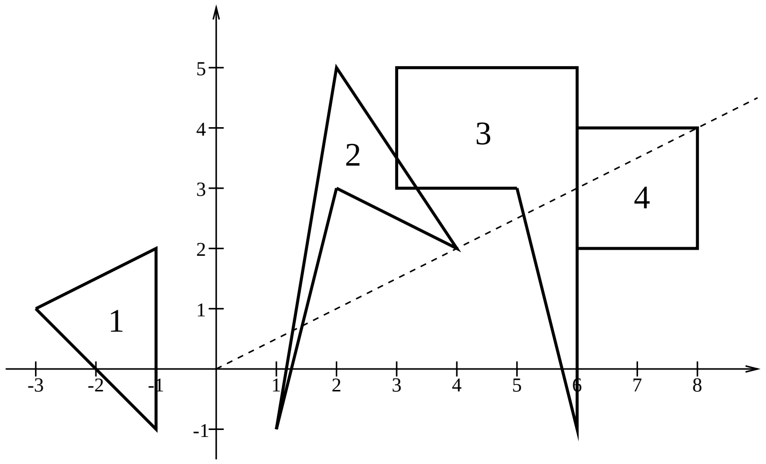


Figure 1: Semi-line for the sample input data.

The intersection of each cloud with the semi-line is a set of segments of the semi-line (there may be single points as well but we will treat them as *segments of length 0*). For example in Fig. 1 the intersection of the cloud number 2 with the semi-line consists of two segments, one of length 0. Each segment represents a time interval when the communication is broken. In the first phase of our solution we compute the segments for each cloud separately. Subsequently we consider the union of the segments computed for all the clouds. Clearly in the union some segments of different clouds may glue together creating a single segment (see clouds 3 and 4 in Fig. 1). Our goal is to compute the number of segments in the union. As an example, for the sample input data there are:

- two segments for cloud number 2, one of which is of length 0,
- one segment for cloud 3,
- one segment for cloud 4.

The answer in this case is 3, as in the union there are 3 segments. Now let us move to the implementation details for the above algorithm.

Phase One: Computing Segments for Each Cloud

Clearly one needs to find all intersection points between the semi-line and the cloud's periphery. To this end one has to examine every edge of the cloud and compute the point of intersection if it exists. This part is performed in $O(k)$ time for a k-vertex cloud.

However, even if the algorithm finds all intersection points in order of occurrence on the perimeter it is not possible to determine all segments for that cloud out of hand, as clouds are of arbitrary shape (excluding self-intersections). To cope with that problem we represent each intersection point by its distance from the point $(0,0)$. Then we sort the intersection points and subsequently group them into pairs. Each pair corresponds to one segment (recall that no single cloud covers laser beam at the very beginning).

Unfortunately, the above description is not complete. It works properly unless the semi-line contains a vertex of some cloud. We cannot exclude such a situation. Therefore we have to consider a set of nasty special cases listed in Figure 2.

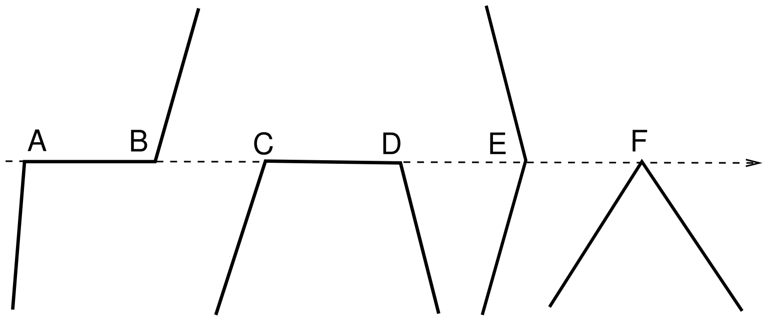


Figure 2: A collection of nasty cases to consider

Two last special cases can be solved straightforward by adding point E for the third case and point F twice for the last case. Observe that this will work no matter on which side is the interior of the cloud. First two cases are harder. To cope with them each intersection point will store additionally its *type*. All points added in the last two cases are called *standard points*. In the first two cases we will also use points marked as *segment points*, as follows:

- The first case — add point A twice: once as a standard point and then as a segment point. Then add point B as a segment point.

- The second case — add points C and D as segment points.

When the set of intersection points for a given cloud is generated, we sort the points (leaving the points with the same coordinates in the same order). In the model program we use built-in quick-sort routine. Then the segments can be computed. We examine subsequent points in the sorted sequence. Standard points in the sequence represent left and right ends of the segments, alternately. Observe that segment points appear in pairs in the sorted sequence. When we find such pair X, Y we look at the most recent standard point. If it turns out to be a left end of a segment, the pair X, Y is ignored. Otherwise (also when there was no previous standard point) we add segment $[X, Y]$.

Clearly, the total time needed to process one cloud is dominated by sorting. Thus the segments for a k-vertex cloud that intersect with the semi-line in q points are computed in $O(k + q \log q)$ time.

Phase Two: Computing the Number of Communication Interruptions

As it has been mentioned before all we need to do to obtain the result is to pick segments computed for all clouds and compute the number of segments in their union. In order to do this we sort the segments according to their left ends. Subsequently we examine successive segments. If the processed segment has its left end farther than all ends of already processed intervals we increase the answer by one (the left end of such an interval does not intersect with previous ones and represents the left end of some segment of the union).

The total time complexity is $O(m + p \log p)$, where m is the total number vertices in all clouds and p is the total number of intersections.

A Hidden Catch

This solution contains a hidden catch. As mentioned before, the algorithm sorts points, what requires comparing their coordinates. In some cases the distance between different intersections is so small, that even the `long double` type is not precise enough. To solve this problem, one has to represent point's location as a fraction of two 64-bit integers. To compare points' locations, it is needed to calculate multiplication of two 64-bit values.

Task by **Marcin Michalski, Anna Niewiarowska**	Solution description by **Łukasz Kowalik, Jakub Pawlewicz**
Available memory: 64 MB.	**Maximum running time: 0.1 s.**

Sweets

John has got n jars with candies. Each of the jars contains a different kind of candies (i.e. candies from the same jar are of the same kind, and candies from different jars are of different kinds). The i-th jar contains m_i candies. John has decided to eat some of his candies. He would like to eat at least a of them but no more than b. The problem is that John can't decide how many candies and of what kinds he would like to eat. In how many ways can he do it?

Task

Your task is to write a program that:

- reads from the standard input the amount of candies in each of the jars, and integers a and b,
- determines the number of ways John can choose the candies he will eat (satisfying the above conditions),
- writes the result to the standard output

Input

The first line of input contains three integers: n, a and b, separated by single spaces ($1 \leq n \leq 10$, $0 \leq a \leq b \leq 10\,000\,000$). Each of the following n lines contains one integer. Line $i+1$ contains integer m_i — the amount of candies in the i-th jar ($0 \leq m_i \leq 1\,000\,000$).

Output

Let k be the number of different ways John can choose the candies to be eaten. The first and only line of output should contain one integer: $k \bmod 2004$ (i.e. the remainder of k divided by 2004).

Example

For the input data:
```
2 1 3
3
5
```
the correct result is:
```
9
```

John can choose candies in the following ways:

$$(1,0),(2,0),(3,0),(0,1),(0,2),(0,3),(1,1),(1,2),(2,1)$$

Solution

First, let us restate the problem using more convenient combinatorial terms. We have to determine the number of ways to distribute at most r identical objects into n distinct containers so that the i-th container has at most m_i objects, for all $i = 1, \ldots, n$. Let us denote this value by $C(r)$. Finding solution requires computing $C(b) - C(a-1)$ or just $C(b)$ if $a = 0$.

Let us recall base combinatorial facts about distribution of identical objects into distinct containers without restrictions.

Proposition 1: *The number of distributions of r identical objects into n distinct containers equals $\binom{r+n-1}{n-1}$.*

Proof: The number of ways r objects can be distributed among n containers corresponds to the number of ways to arrange r objects and $n-1$ bars, which amounts to $\binom{r+n-1}{n-1}$. Bars divide objects into n groups. Each consecutive group is a container. ■

Proposition 2: *The number of distributions of at most r identical objects into n distinct containers equals $\binom{r+n}{n}$.*

Proof: It immediately follows from Proposition 1. Putting at most r objects into n containers corresponds to putting exactly r objects into $n+1$ containers. Additional container collects objects which are not distributed among the first n containers. ■

Let U be the set of all distributions of at most r objects to n containers without restrictions. Let A_i be a set of all distributions of at most r objects to n containers such that the i-th container contains more than m_i objects. Thus $\overline{A_i}$ is a set of all distributions of r objects to n containers for which the i-th container contains no more than m_i objects. Thus the searched number is $|\overline{A_1} \cap \ldots \cap \overline{A_n}|$.

We determine this value using the Inclusion-Exclusion Principle:

$$\begin{aligned} |\overline{A_1} \cap \ldots \cap \overline{A_n}| \quad = \quad & |U| - |A_1| - \ldots - |A_n| + |A_1 \cap A_2| + \ldots \\ & + |A_{n-1} \cap A_n| - \ldots + (-1)^n |A_1 \cap \ldots \cap A_n| \end{aligned} \tag{1}$$

We will find the value of $|A_{i_1} \cap \ldots \cap A_{i_k}|$ for given $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$. $A_{i_1} \cap \ldots \cap A_{i_k}$ is a set of distributions of r objects to n containers such that for every $j \in \{i_1, \ldots, i_k\}$, j-th container contains more than m_j objects. Let us see how we can build such distribution. First we put $m_j + 1$ objects to the j-th container for every $j \in \{i_1, \ldots, i_k\}$. Then we distribute the remaining objects to all n containers. From Proposition 2 follows that we can do it in $\binom{r-(m_{i_1}+1)-\ldots-(m_{i_k}+1)+n}{n}$ ways. Hence

$$|A_{i_1} \cap \ldots \cap A_{i_k}| = \binom{r - (m_{i_1} + 1) - \ldots - (m_{i_k} + 1) + n}{n}. \tag{2}$$

Using (1) and (2) we can write the final result in a single formula:

$$S(r) = \sum_{I \subseteq \{1,\ldots,n\}} (-1)^{|I|} \binom{r - \sum_{i \in I}(m_i + 1) + n}{n}. \tag{3}$$

Binomial Coefficients

The main difficulty we have to face during implementation is to compute the values of binomial coefficients modulo 2004. There are several methods to cope with that, varying in simplicity and time complexity.

Pascal's Triangle

We can take advantage of the widely-known recurrence leading to the Pascal's Triangle:

$$\binom{p}{q} = \begin{cases} 1 & \text{for } p = 0 \text{ or } q = 0, \\ \binom{p-1}{q} + \binom{p-1}{q-1} & \text{for } p, q > 0. \end{cases}$$

Using this recurrence we can compute all values of $\binom{p}{n} \bmod 2004$ for $p \leq b - \sum_{i \in I}(m_i + 1) + n$ using simple dynamic programming. Then the stored numbers can be used to compute $S(r)$. This preprocessing uses an array of at most $b + n$ integers which is quite much but still within the given limit. However the time complexity of the preprocessing is $O(bn)$ which dominates the time needed for computing $S(b)$ and $S(a-1)$. The total time complexity of the solution is then $O(bn + 2^n)$.

A Smarter Recurrence

Let us now present an approach similar to binary exponentiation. To get $\binom{p}{q}$ we calculate values $\binom{p}{i}$ for all $i = 0, \ldots, q$. For $p = 1$ we have $\binom{p}{0} = \binom{p}{1} = 1$ and the other values are 0. For $p > 1$ we use the following formula:

$$\binom{p}{i} = \begin{cases} \binom{p-1}{i} + \binom{p-1}{i-1} & \text{for odd } p, \\ \sum_{j=0}^{i} \binom{p/2}{j}\binom{p/2}{i-j} & \text{for even } p. \end{cases}$$

Hence we may compute the value of a binomial coefficient $\binom{p}{q}$ in time $O(q^2 \log p)$ using only addition and multiplication operations. The running time complexity is $O(2^n n^2 \log b)$ and the space complexity is $O(n)$.

One More Approach

Finally we can take advantage of the following widely-known formula $\binom{p}{q} = \frac{p^{\underline{q}}}{q!}$, where $p^{\underline{q}} = p \cdot (p-1) \cdot \ldots \cdot (p - q + 1)$. Unfortunately the value in the numerator of the above expression can be very large. One way to get rid of that obstacle is to implement big integer arithmetic (i.e. addition and subtraction of big numbers and also multiplication and division of a big number by a small number). However this method ignores the useful fact we are interested in results modulo 2004. Then one can proceed as follows.

1. Compute the prime factorization of $q!$. As $q = n \leq 10$, the contestants can even find the factorization by hand for each $q = 1, 2, \ldots 10$ and code it in their source files as a constant array.

2. Prepare an array `A[1..`q`]` containing initially numbers $p, p-1, p-2, \ldots, p-q+1$.

3. Next we take successive prime factors. For each such factor p of order k we repeat the following step k times: find the first number `A[`i`]` divisible by p, divide it by p and store the result in `A[`i`]`.

4. Compute the product `A[1]`$\cdot$`A[2]`$\cdot \ldots \cdot$`A[`q`]` mod 2004. Clearly, to stay in `int` range it suffices to compute the remainder of division by 2004 after each multiplication. The result is equal to $\binom{p}{q}$ mod 2004.

Computing $\binom{p}{q}$ using the above method takes $O(q^2 \log q)$ time. It follows that the total time complexity is then $O(2^n n^2 \log n)$.

Alternative Solution: Recursion and Dynamic Programming

Let $c_k(r)$ denote the number of distributions of exactly r objects to the first k containers. The following recurrence holds:

$$c_0(r) = [r = 0], \quad c_k(r) = \sum_{i=0}^{m_k} c_{k-1}(r - i)$$

Using the above equations and dynamic programming we get an algorithm with the running time complexity $O(bM)$ and the space complexity $O(b)$, where $M = \sum_{i=1}^{n} m_i$.

We can speed up calculations of c_k by preparing partial sums of c_{k-1}. Let $C_{k-1}(r) = \sum_{i=0}^{r} c_{k-1}(i)$, then $c_k(r) = C_{k-1}(r) - C_{k-1}(r - m_k - 1)$. Of course $C_{k-1}(r - m_k - 1) = 0$ if $r - m_k - 1 < 0$. This solution has the running time complexity $O(bn)$.

Task by	Solution description by
Krzysztof Sikora	**Marcin Michalski, Anna Niewiarowska**
Available memory: 64 MB.	**Maximum running time: 3 s.**

Trips

In the forthcoming holiday season, a lot of people would like to go for an unforgettable travel. To mostly enjoy their journey, everyone wants to go with a group of friends. A travel agency offers several trips. A travel agency offers group trips, but for each trip, the size of the group is limited: the minimum and maximum number of persons are given. Every group can choose only one trip. Moreover, each trip can be chosen by only one group. The travel agency has asked you for help. They would like to organize as many trips as possible. Your task is to match groups of people and trips in such a way, that the maximum number of trips can be organized.

Task

Write a program, that:

- reads the description of the groups and the trips from the standard input,
- matches the groups and trips in such a way, that the maximum number of arranged trips is reached,
- writes the result to standard output.

If there are several possible solutions, your program should output anyone of them.

Input

The first line of input contains two integers: n and m separated by single space, $1 \le n \le 400000$, $1 \le m \le 400000$; n is the number of groups and m is the number of trips. The groups are numbered from 1 to n, and the trips are numbered from 1 to m.

The following n lines contain group sizes, one per line. Line $i+1$ contains integer s_i — the size of the i-th group, $1 \le s_i \le 10^9$.

The following m lines contain trip descriptions, one trip per line. Line $n+j+1$ contains two integers: l_j and u_j, separated by single space. l_j is the minimum, and u_j is the maximum size of a group for which the trip can be arranged, $1 \le l_j \le u_j \le 10^9$.

Output

The first line of output should contain one integer $k \ge 0$ — the maximum number of trips that can be arranged. The following k lines should contain the description of the matching. Each of these lines should contain a pair of integers separated by single space: the number of a group and the number of a trip. There can be many answers and your program may print anyone of them.

Example

For the input data:

```
5 4
54
6
9
42
15
6 6
20 50
2 8
7 20
```

the correct result is:

```
3
2 1
3 4
4 2
```

Solution

Maximal matching in a bipartite graph

Our problem can be easily reduced to a problem called *maximal matching in a bipartite graph* in the graph theory. In a graph $G = (V, E)$ a matching M is a subset of E such that no two different edges belonging to M have a vertex in common. The problem is to find a matching that consists of the maximum number of elements.

In our problem we have a bipartite graph $G = (V, W, E)$, where V is the set of groups, W is the set of trips and E is the set of edges. Two vertices $v \in V$ and $w \in W$ are connected with an edge if the group v can be sent to the trip w. It means:

$$(i, j) \in E \Leftrightarrow i \in V \wedge j \in W \wedge l_j \leq s_i \leq u_j$$

A matching M in that graph represents an assignment of some groups with trips. If $(v, w) \in M$ then the group v will be sent to the trip w. Each matching in the graph represents a correct assignment of the trips with the groups. The definition of a matching guarantees that one trip will be assigned to at most one group of people and one group of people will not go to more than one trip.

Our task is to choose a maximal matching P in the graph — such $P \subseteq E$ that

$$\forall_{(i,j)\in P}\forall_{(a,b)\in P}((i, j) = (a, b) \vee (i \neq a \wedge j \neq b))$$

and $|P|$ is the greatest possible.

A problem of finding a maximal matching in a bipartite graph can be solved with many well known algorithms such as the Hopcroft-Karp algorithm. However, these algorithms are too slow for our task (and too complicated for such competition as CEOI). They do not use one interesting property of considered graphs: the set E is convex, i.e.: $\forall_{(i,j),(k,j)\in E}(i+1, j), \ldots, (k-1, j) \in E$. Using this property, the problem can be solved in a greedy way.

Greedy algorithm

Our problem can be solved by a greedy algorithm. Below there is a description of the algorithm:

1. Sort groups by size in the ascending order.
2. For each group i:
   (a) find such a trip j which is still not used, $u_j \geq s_i \geq l_j$ and u_j is minimal (where s_i - the number of people in the i-th group, l_j - the minimum size of the j-th group, u_j - the maximum size of the j-th group),
   (b) if such a trip exists then send the i-th group on this trip (it means that this trip is now used) and increase the number of trips.
3. Print the result and all matched pairs.

Proof of correctness

After matching a group to a trip we remove the trip from the trip list. Let us say that we have already matched previous $k-1$ groups optimally and now we are going to do so for the k-th group. Let us presume that we can send the k-th group on trips $a_1, a_2, \ldots, a_p$ (it means that those trips haven't been chosen earlier) and that $u_{a_1} \leq u_{a_2} \leq \ldots \leq u_{a_p}$.

As we matched previous $k-1$ groups optimally there is at least one optimal solution S. I am going to prove that there is an optimal solution in which the k-th group is send on trip a_1 and all previous groups are matched as before.

There are three possibilities:

1. in the solution S the k-th group is send on the trip a_1,
2. in the solution S the k-th group is send on the trip a_x and $x \neq 1$,
3. in the solution S the k-th group is not matched.

If the first condition is met then it is perfect and we have an optimal solution with the matching we wanted.

If the k-th group is not matched then the trip a_1 must be matched. (Otherwise we could simply send the k-th group on the trip a_1 and we would have a better solution). Let us presume that in the solution S the trip a_1 is matched with a group z. Of course the group z has not been processed yet because the trip a_1 would have been deleted before (as we delete trips after "using" them). So if we match the k-th group with the trip a_1 and z with nothing, we will still have an optimal solution (the number of matchings stays unchanged).

The last possibility is that the k-th group is matched with the trip a_x and $x \neq 1$. There are two cases:

- the trip a_1 is not matched,

- the trip a_1 is matched with a group z and the group z has not been processed yet.

If the trip a_1 is not matched, we simply "change" matching from the trip a_x to a_1 as that will not change the number of matchings. If a_1 is matched with z and z has not been processed yet, then we can "swap" these two matchings. It means we match the k-th group with a_1 and the group z with a_x. We can do it because if k is matched with a_1 and z with a_x then

$$l_{a_1} \leq s_k \leq u_{a_1}$$

$$l_{a_x} \leq s_z \leq u_{a_x}$$

But from the definition of a_x and a_1 we have

$$l_{a_x}, l_{a_1} \leq s_k$$

and

$$u_{a_1} \leq u_{a_x}$$

And so we have:

$$l_{a_1}, l_{a_x} \leq s_k \leq s_z \leq u_{a_1} \leq u_{a_x}$$

So we can match k with a_1 and z with a_x. And this also does not decrease the number of matchings.

We managed to prove that if our algorithm optimally matched first $k-1$ groups it will match optimally the k-th group. By the mathematical induction, presuming that the matching of zero groups is optimal, the above algorithm generates a correct answer.

Implementation: The solution uses a heap H to keep all the possible trips for the current group. If the group is changed, some trips are added and others are removed. In the root of the heap there is always a trip with the smallest u_i from all the trips that are in the heap.

The algorithm consists of the following steps:

- Sort the groups in the ascending order of s_i.
- Sort the trips in the ascending order of l_j.
- For each group i (taking the groups in the ascending order of s_i):
  - add to the heap H all the trips, that have not been added before and that have minimal demands (l_j) not bigger than the size of the current group,
  - take from the heap H all the groups that have the maximal demands (u_j) smaller that the size of the current group,
  - take from the heap the group with the minimal u_j (if the heap is not empty now) - this is the correct matching for the group i.

The time complexity is $\Theta(nlogn) + \Theta(n + mlogm)$, and the memory cost is $\Theta(n + m)$.

Task by **Jakub Pawlewicz, Paweł Parys**	Solution description by **Jakub Radoszewski**
Available memory: 32 MB.	**Maximum running time: 3 s.**

Football league

There are n teams in a football league (we assume that n is even). During a season each team plays with every other team exactly once. The season consists of $n-1$ turns. Every team plays exactly once during a turn. It is desired for a team to play consecutive matches on different stadia: one at home stadium, and one away, etc. Unfortunately it is not always possible to construct such a game schedule that no team plays twice in a row at home stadium, or twice in a row away. When constructing the schedule the number of such situations should be minimized. (For example, if a team plays once away, then four times at home stadium and then once away, it counts as three such situations.)

Your task is to minimize the number of situations in which a team plays twice in a row at home or away and to construct such game schedule for the whole season. The schedule should consist of $n-1$ turns. Each turn consists of $\frac{n}{2}$ matches — each team plays exactly one match. There are $\frac{n(n-1)}{2}$ matches in the whole season, and every two teams should play exactly one match against each other. Each match is played at one of the opponents' stadium — one team plays at home stadium and the other one plays away. The total number of situations in which a team plays two consecutive matches at home or away should be minimal.

Task

Write a program, that:

- reads the number of teams from the standard input,
- computes the minimum total number of situations in which a team plays twice in a row at home or away and constructs the game schedule,
- writes the result to standard output.

Input

The first and only line of the standard input contains one even integer n ($2 \leq n \leq 1000$) — the number of teams.

Output

The first line of the standard output should contain a single integer — the minimum total number of situations in which a team plays twice in a row at home or away. The following $n-1$ lines should contain a game schedule: the line $k+1$ should contain the description of the k-th turn.

The description of a turn consists of n different numbers $d_1, d_2, \ldots, d_n$ from $\{1, 2, \ldots, n\}$ separated by single spaces. For $i = 1, 2, \ldots, \frac{n}{2}$ the pair d_{2i-1}, d_{2i}

denotes a match between teams d_{2i-1} and d_{2i}. Team d_{2i-1} plays at home and team d_{2i} plays away.

Example

For the input data:	the correct result is:

`4`	`2`
	`1 4 2 3`
	`1 2 4 3`
	`2 4 3 1`

Solution

To describe the optimal solution we need to prove a few lemmas first.

Lower bound

Lemma 1: *The lower bound of the number of situations when a team plays twice in a row at home or away is $n-2$, where n is the number of teams.*

Proof: Let us assume (without losing the generality) that in the first turn of the game schedule teams $1,\ldots,\frac{n}{2}$ play at home (and teams $\frac{n}{2}+1,\ldots,n$ play away). When any two of teams $1,\ldots,\frac{n}{2}$ have to play together, then one of them has to play twice in a row at home or away (because all of those teams are in the same phase of playing at home and away in the beginning, so if none of the teams would change its phase, then it would not be possible for any two teams to play together without changes of phase — meaning teams playing alternately at home and away in the whole league). Let us assume that this team which plays two consecutive matches at home or away, has number $\frac{n}{2}$. (It is worth noticing, that when this team actually changes its phase, it can play with all the teams $1,\ldots,\frac{n}{2}-1$, if they have not changed their phases earlier.)

This argumentation can be repeated for teams $1,\ldots,\frac{n}{2}-1$, as they also have to play all possible matches with each other. More precisely, each of the teams $2,\ldots,\frac{n}{2}-1$ has to change its phase at least once in order to be able to play matches with all the teams with numbers lower than its. So there have to be at least $\frac{n}{2}-1$ situations, when a team plays two consecutive matches at home or away.

The whole above argumentation can be repeated for teams $\frac{n}{2}+1,\ldots,n$ (as they played away in the first turn and their situation is symmetric to the situation analyzed above), getting another $\frac{n}{2}-1$ previously mentioned situations. So, the total number of such situations, when a team plays two matches at home or away in a row is at least $n-2$. ■

Applying lemma 1, we get a lower bound of situations when a team plays two consecutive matches at home or away. In fact, the construction of a game schedule where the number of such situations is equal to $n-2$ is always possible.

At first such a construction will be described, and then we prove that it meets all the conditions of correct game schedule, and that it generates exactly $n-2$ conflicts.

The construction

For convenience we will number turns and teams from 0 (so the last team has number $n-1$, and the last turn has number $n-2$). Then we define functions $p_k : \{0, \ldots, n-2\} \mapsto \{0, \ldots, n-1\}$ for $k = 0, \ldots, n-2$. Let $i \in \{0, \ldots, n-2\}$. Let $j \in \{0, \ldots, n-2\}$ be such that $i + j \equiv k \pmod{n-1}$. Then if $i \neq j$, then we set $p_k(i) = j$, and when $i = j$, we set $p_k(i) = n-1$.

Now we can show how the k-th turn looks like. Let $l = \lfloor \frac{k+1}{2} \rfloor$. When k is even, then in k-th turn we arrange pairs: $\big(l, p_k(l)\big), \big(l+1, p_k(l+1)\big), \ldots, \big(l + \frac{n}{2} - 1, p_k(l + \frac{n}{2} - 1)\big)$. When k is odd, we arrange pairs: $\big(p_k(l), l\big), \big(p_k(l+1), l+1\big), \ldots, \big(p_k(l+\frac{n}{2}-1), l+\frac{n}{2}-1\big)$. Below it is proved, that the above construction meets all the expected conditions.

The correctness and optimality of the construction

Lemma 2: *For the above construction, every team from* 0 *to* $n-1$ *plays exactly once (so every single turn of the schedule is correct).*

Proof: For convenience, let us consider cases when k is even and k is odd separately:

1. If k is even, then there exists an integer m such that $k = 2m$. So, $l = \lfloor \frac{2m+1}{2} \rfloor = m$. In this case, the pairs that we construct are:

$$\big(m, p_k(m)\big), \big(m+1, p_k(m+1)\big), \ldots, \big(m + \frac{n}{2} - 1, p_k(m + \frac{n}{2} - 1)\big)$$

If we use the definition of p_k to count its value, we get a precise representation of these pairs:

$$\big(m, n-1\big), \big(m+1, (m-1) \ mod \ (n-1)\big), \big(m+2, (m-2) \ mod \ (n-1)\big), \ldots,$$

$$\big(m + \frac{n}{2} - 1, (m - \frac{n}{2} + 1) \ mod \ (n-1)\big)$$

where „mod" is an operation of taking the remainder in the integer division by $n-1$, which is always of the range between 0 and $n-2$. The numbers $m - \frac{n}{2} + 1, \ldots, m-1, m, \ldots, m + \frac{n}{2} - 1$ are consecutive $n-1$ integers, so every one of them gives a different remainder when divided by $n-1$, hence they represent every team between 0 and $n-2$. As team $(n-1)$-th appears exactly once in this list, every team appears exactly once in this list of pairs (it is true because, as $k \leq n-2$, $m + \frac{n}{2} - 1 = \frac{k}{2} + \frac{n}{2} - 1 \leq \frac{n-2}{2} + \frac{n}{2} - 1 = n-2$, so none of the teams on the first positions in the pairs will be $n-1$, and of course none of the teams appearing on the second position in the pairs apart from the first pair may be $n-1$).

2. If k is odd, then there exists such an integer m, that $k = 2m + 1$. Hence, $l = \lfloor \frac{2m+2}{2} \rfloor = m + 1$. In this case, the pairs that we construct are:

$$\big(p_k(m+1), m+1\big), \big(p_k(m+2), m+2\big), \ldots, \big(p_k(m+\frac{n}{2}), m+\frac{n}{2}\big)$$

If we use the definition of p_k to count its value, we similarly, like above get a precise representation of these pairs:

$$\big(m, m+1\big), \big((m-1) \ mod \ (n-1), m+2\big), \ldots,$$
$$\big((m-\frac{n}{2}) \ mod \ (n-1), m+\frac{n}{2}-1\big), \big(n-1, m+\frac{n}{2}\big)$$

(as $m - \frac{n}{2} + 1 = m + \frac{n}{2} - (n-1)$, so $m - \frac{n}{2} + 1 \equiv m + \frac{n}{2} \ mod \ (n-1)$ — and that is why we get $n-1$ in the last pair). Similarly like in the previous case, in this list of pairs all the team numbers between 0 and $n-2$ appear exactly once, and because $k = 2m+1 \leq n-2$, $m \leq \frac{n}{2} - 2$ (as n is always even), $m + \frac{n}{2} \leq n-2$, the team number $n-1$ appears also exactly once, and that completes the proof of the lemma in this case.

■

Lemma 3: *When we take the whole game schedule into consideration, every pair of teams plays exactly once (so the whole schedule is correct).*

Proof: Let $i \in \{0, \ldots, n-2\}$. The opponents of i-th team will be teams with such numbers j_k, that $i + j_k \equiv k \pmod{n-1}$ (when such j_k is equal to i, the opponent of i will be of course $n-1$) — it is true, because i plays in every round exactly once (we get it applying lemma 2). So, these numbers j_k will be: $(-i) \ mod \ (n-1), (-i+1) \ mod \ (n-1), \ldots, (-i+n-2) \ mod \ (n-1)$. The numbers $-i, -i+1, \ldots, -i+n-2$ are $n-1$ consecutive integers, so after counting the values of the appropriate remainders, we get every number between 0 and $n-2$ exactly once. Of course, the number i will appear here because of that, and so in the place of its appearance the team $n-1$ will be the opponent of i-th team, so every (other than i-th) team will appear exactly once in the list $p_0(i), \ldots, p_{n-2}(i)$.

From this part of proof we can deduce, that the team number $n-1$ will also play with all the teams between 0 and $n-2$ exactly once in the schedule. It is true, because for every $i \in \{0, \ldots, n-2\}$ there exists such a round number, that in this round i-th team plays with team $(n-1)$-th, and — applying lemma 2 — the team $(n-1)$ plays exactly once in every round, so this implies that the matches of the team $n-1$ also satisfy the thesis of lemma 3. This completes the proof of lemma 3. ■

From lemmas 2 and 3 we deduce, that the game schedule constructed is correct. In the final theorem we will prove, that it is also optimal.

Theorem 1: *The game schedule constructed above is optimal, meaning that the number of situations, when a team plays twice at home or away, is equal to the lower bound from lemma 1, that is $n-2$.*

Proof: In this proof, similarly to the proof of lemma 2, we will firstly consider cases when we move from a turn with an even number to a turn with an odd number and secondly the remaining ones:

1. If we consider turns k-th and $(k+1)$-th where k is even, using the representation of the pairs described in lemma 2's proof (with $m = \frac{k}{2}$) we get for k-th turn pairs: $\big(m, n-1\big), \big(m+1, (m-1) \ mod \ (n-1)\big), \big(m+2, (m-2) \ mod \ (n-1)\big), \ldots, \big(m + \frac{n}{2} - 1, (m - \frac{n}{2} + 1) \ mod \ (n-1)\big)$, and for the $(k+1)$-th turn pairs: $\big(m, m+1\big), \big((m-1) \ mod \ (n-1), m+2\big), \ldots, \big((m - \frac{n}{2}) \ mod \ (n-1), m + \frac{n}{2} - 1\big), \big(n-1, m+\frac{n}{2}\big)$. Now, we can easily notice, that only the team number m plays here twice in a row at home, and the team number $m + \frac{n}{2}$ plays twice in a row away (as — what was already mentioned — $m - \frac{n}{2} + 1 \equiv m + \frac{n}{2} \pmod{n-1}$), with all the other teams not changing their phases. So, every such change of turns gives two teams changing their phases, and there are exactly $\frac{n}{2} - 1$ such changes, what gives $2 \cdot (\frac{n}{2} - 1) = n - 2$ situations we consider.

2. Now let us consider turns k-th and $(k+1)$-th where k is odd, again using the representation from lemma 2's proof (with $m = \lfloor \frac{k}{2} \rfloor$); for k-th turn we have pairs: $\big(m, m+1\big), \big((m-1) \ mod \ (n-1), m+2\big), \ldots, \big((m-\frac{n}{2}) \ mod \ (n-1), m+\frac{n}{2}-1\big), \big(n-1, m+\frac{n}{2}\big)$, and for the $(k+1)$-th turn pairs: $\big(m+1, n-1\big), \big(m+2, m \ mod \ (n-1)\big), \big(m+3, (m-1) \ mod \ (n-1)\big), \ldots, \big(m+\frac{n}{2}, (m-\frac{n}{2}+2) \ mod \ (n-1)\big)$. If we consider the first teams in the second set of pairs and the second teams in the first set of pairs we easily notice, that these sets of teams are the same, what implies that all the teams maintain their phases in playing at home and away, so there are no previously mentioned conflicts.

After considering both cases we get, that there are exactly $n - 2$ situations when a team plays twice at home or away in a row, and that proves the thesis of theorem 1. ■

The correct and optimal solution of the task, has time complexity $O(n^2)$ (the size of the output is $O(n^2)$) and requires constant memory.

Task by	Solution description by
Rafał Rusin, Anna Niewiarowska	**Łukasz Kowalik, Marcin Michalski**
Available memory: 32 MB.	**Maximum running time: 0.3 s.**

Puzzle

The king of Byteland has received a gift, a jigsaw puzzle. The puzzle consists of a board of size $n \times n$. The field in the i-th row and j-th column $(1 \leq i, j \leq n)$ has coordinates (i, j) and contains a piece with the number $p(i, j)$, $1 \leq p(i, j) \leq n^2$. Each of the numbers $1 \ldots n^2$ appears on exactly one of the pieces. To solve the

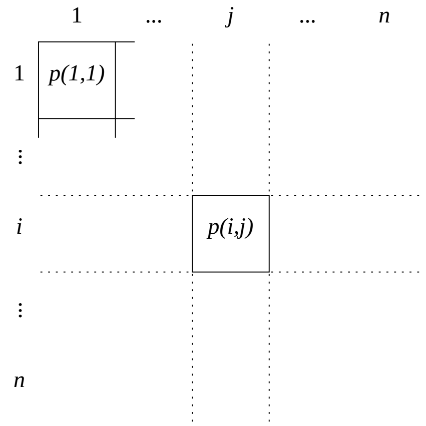


Figure 3: Coordinates of the fields

puzzle, you have to put the pieces in order, so that for each $1 \leq i, j \leq n$ the field (i, j) contains the piece with number $j + (i - 1) \cdot n$.

The following moves are allowed to solve the puzzle:

- a cyclic shift of all pieces in a row a certain number of fields to the right,
- a cyclic shift of all pieces in a column a certain number of fields down.

The king of Byteland managed to solve his puzzle, but he is not sure if he would be able to solve it starting from a different initial configuration. Help him to solve this problem.

Task

Write a program that:

- Reads from the standard input the description of the initial configuration of the puzzle.
- Finds out if the pieces on the board can be put in order using only the moves given above. If the solution is possible, the program should find the moves putting the pieces in order.
- Writes the result to the standard output.

Input

In the first line of the standard input there is one integer n — the size of the board side ($2 \le n \le 200$). The following n lines contain the description of the initial configuration. The line $i+1$ contains n integers $p(i,1), p(i,2), \ldots p(i,n)$ separated by single spaces.

Output

If there is no solution, the program should write to the standard output only one line containing only one word `NO`.

If a solution exists, the first line should contain one integer m — the number of moves leading to the solution of the puzzle. The number of moves in your solution must not exceed 400 000. The following m lines should contain the descriptions of the moves, one move per line.

Each such line should consist of a letter `R` (for shifting a row to the right) or `C` (for shifting a column down), a space, and two integers: k and l separated by a space; $1 \le k \le n$, $1 \le l \le n-1$. A line containing `R` k l describes a cyclic shift of the k-th row l fields to the right. Such a move leads to the following board configuration:

$$p'(i,j) = \begin{cases} p(i, j+n-l) & \text{if } i = k \text{ and } j \le l \\ p(i, j-l) & \text{if } i = k \text{ and } j > l \\ p(i,j) & \text{if } i \ne k \end{cases}$$

Similarly, a line containing `C` k l describes a cyclic shift of the k-th column l fields down.

If there are several possible solutions, your program should output anyone of them.

Example

For the input data:

```
4
4 6 2 3
5 10 7 8
9 14 11 12
13 1 15 16
```

the correct result is:

```
2
C 2 1
R 1 3
```

The above sequence of moves gives the following sequence of configurations:

4	6	2	3
5	10	7	8
9	14	11	12
13	1	15	16

4	1	2	3
5	6	7	8
9	10	11	12
13	14	15	16

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Solution

The puzzle described in the task seems to be similar to the Rubik's cube. Indeed, the approach we are going to present can be used also for solving the Rubik's cube. It consists of two phases:

1. put appropriate pieces in all rows but the last one,
2. rearrange the last row.

First phase

In this phase we put successive pieces $1, 2, \ldots, n^2 - n$ on proper fields. Let us assume that we already moved pieces $1, 2, \ldots, k-1$ to proper fields. Now we focus on the piece number k and we want to place it in the proper field in such a way that the preceding pieces stay at their positions. There are four possible cases:

- the piece is already on the proper field,
- the piece is in the proper row,
- the piece is in the proper column,
- the piece is somewhere else.

In the first case nothing should be done. If the piece is in the proper row we proceed as in Figure 4. Note that after the four shifts shown in the figure only three pieces change their positions: j, k and m. If the piece is in the proper column we proceed similarly — again four shifts are sufficient.

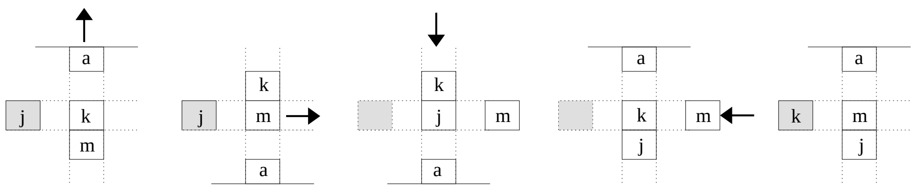


Figure 4: Four shifts to move the piece k to the proper (gray) field, provided that it was already in the proper row.

Finally, if the piece stays in improper row and column we proceed as in Figure 5. Note again that after the four shifts shown in the figure only two pieces change their positions: j and k. It ends the description of the first phase.

Second Phase

Before we show how to rearrange pieces in the last row let us think a bit on the problem raised by the king of Byteland — is there *always* a way to solve the puzzle?

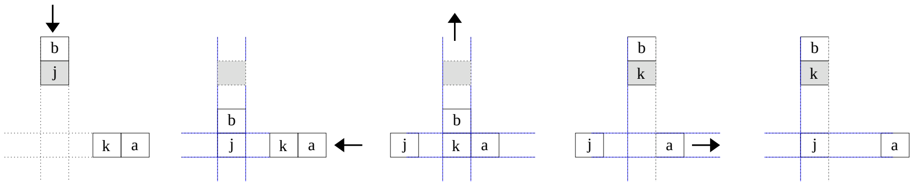


Figure 5: Four shifts to move the piece no. k to the proper (gray) field, provided that it was in improper row and column.

Permutations

Let us recall some basic facts about permutations. Let $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ be some permutation. If we start from an arbitrary element $x \in \{1, \ldots, n\}$ and iterate f obtaining $x, f(x), f(f(x)), f(f(f(x))), \ldots$ after several (at most n) iterations we will get x again. It follows that each permutation is a collection of cycles. For example, the permutation $(2, 4, 5, 1, 3)$ consists of two cycles and we write $(2, 4, 5, 1, 3) = [2, 4, 1][5, 3]$. We also say that the permutation $(2, 4, 5, 1, 3)$ is the product of $[2, 4, 1]$ and $[5, 3]$. Cycles of length 2 are of special importance and they are called *transpositions*. Each cycle can be written as a product of transpositions:

$$[1, 2, \ldots, k] = [1, k][1, k-1] \ldots [1, 3][1, 2]$$

(We apply successive transpositions from right to left). It follows that every permutation can be written as a product of transpositions. This decomposition is not unique, e.g. $(2, 4, 5, 1, 3) = [4, 1][2, 1][5, 3] = [2, 4][4, 1][5, 3]$. Even the numbers of transpositions in two decompositions need not to be the same: $[1, 3][1, 7][1, 3] = [3, 4][4, 5][5, 6][6, 7][5, 6][4, 5][3, 4]$. However, there is some invariant. The following proposition is widely-known:

Proposition 1: *If a permutation f can be written as a product of an even number of transpositions, then it cannot be written as a product of an odd number of transpositions.*

Thus we can divide all permutations into two groups: *even* permutations which have an even number of transpositions in every decomposition and *odd* permutations — the remaining ones. Consider a permutation with only one cycle. It follows that the length of the cycle is even if and only if the permutation is odd.

Permutations and the Puzzle

Clearly the initial configuration is some permutation of all n^2 pieces. Let us denote it by π_0. Each cyclic shift is also a certain permutation. Trivially it consists of one cycle. The final configuration of pieces is the identity permutation *id*. We want to apply a number of shifts to the initial configuration to get *id*. In other words, we are looking for a sequence of cyclic permutations $s_1, s_2, \ldots s_k$, corresponding to row and column shifts, such that $s_k s_{k-1} \cdots s_2 s_1 \pi_0 = id$. Assume that n is odd. Then each s_i is an even permutation. As *id* is even it follows that the puzzle is solvable only if π_0 is even. It takes $O(n^2)$-time to verify whether the initial configuration is even when n is odd.

Nevertheless, what should we do if it turns out that n is even or π_0 is even? After the first phase we obtain at most one row disarranged (namely the last row). Observe that when n is even and the configuration after the first phase is odd we can make it even by shifting the last row by one. Thus in the sequel we can assume that after completing the first phase we have got an even configuration of pieces.

Then, we can move successive pieces $n^2-n+1, n^2-n+2, \ldots, n^2$ to proper fields as follows. Consider a piece with number k. Assume that some piece number l is placed in the k-th field. There must be also a piece number m, such that $m > k$ and $m \neq l$, as otherwise our configuration consists of just one transposition and is odd. Figure 6 shows that 7 moves suffice to apply the [k,l,m] permutation. As a result, k finds its way to the proper field and every piece $j < k$ stays at its position. That ends the description of the second phase.

Observe that the number of shifts performed by the algorithm is at most $4n(n-1) + 7n < 400\,000$.

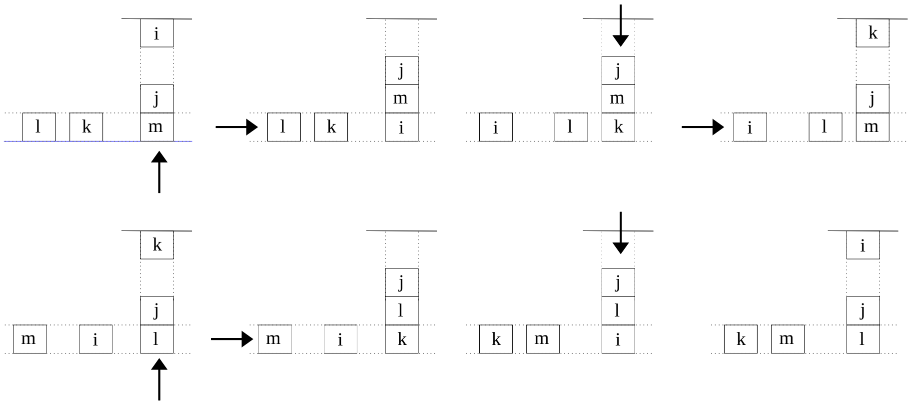


Figure 6: Rearranging the last row: k moves to l's place, l to m's place and m to k's place.

Implementation

There are many ways to code the correct solution but there are some tricks that can help to speed up the program.

If we store a board of n^2 pieces and simulate each move by moving pieces in the board the time complexity is $O(n^3)$. The time complexity can increase even to $O(n^4)$ when we do not store positions of the pieces — then we need $O(n^2)$ time to find each of $O(n^2)$ pieces.

Solution running in $O(n^2)$ time is based on a simple observation. We have described three ways of moving a piece to its proper field. Each way consists of a group of 4 or 7 shifts but as a result of these shifts only two or three pieces change their positions. Thus, positions of pieces after performing each group of shifts can be updated in constant time.

Task by
Wojciech Rytter, Paweł Parys

Solution description by
Michał Adamaszek, Łukasz Kowalik

Available memory: 32 MB. **Maximum running time: 0.1 s.**

Two sawmills

There are n old trees planted along a road that goes from the top of a hill to its bottom. Local government decided to cut them down. In order not to waste wood each tree should be transported to a sawmill.

Trees can be transported only in one direction: downwards. There is a sawmill at the lower end of the road. Two additional sawmills can be built along the road. You have to decide where to build them, as to minimize the cost of transportation. The transportation costs one cent per meter, per kilogram of wood.

Task

Write a program, that:

- reads from the standard input the number of trees, their weights and locations,
- calculates the minimum cost of transportation,
- writes the result to the standard output.

Input

The first line of the input contains one integer n — the number of trees ($2 \leq n \leq 20\,000$). The trees are numbered $1, 2, \ldots, n$, starting from the top of the hill and going downwards. Each of the following n lines contains two positive integers separated by single space. Line $i+1$ contains: w_i — weight (in kilograms) of the i-th tree and d_i — distance (in meters) between trees number i and $i+1$, $1 \leq w_i \leq 10\,000$, $0 \leq d_i \leq 10\,000$. The last of these numbers, d_n, is the distance from the tree number n to the lower end of the road. It is guaranteed that the total cost of transporting all trees to the sawmill at the end of the road is less than $2\,000\,000\,000$ cents.

Output

The first and only line of output should contain one integer: the minimum cost of transportation.

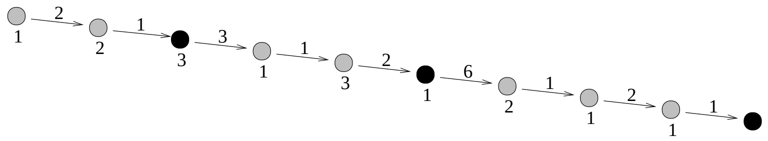
Example

For the input data:

```
9
1 2
2 1
3 3
1 1
3 2
1 6
2 1
1 2
1 1
```

the correct result is:

```
26
```



The figure shows the optimal location of sawmills for the example data. Trees are depicted as circles with weights given below. Sawmills are marked black. The result is equal to:

$$1 \cdot (2+1) + 2 \cdot 1 + 1 \cdot (1+2) + 3 \cdot 2 + 2 \cdot (1+2+1) + 1 \cdot (2+1) + 1 \cdot 1$$

Solution

First of all let us make a trivial observation: it is not reasonable to build the sawmills between trees – in such case the cost will surely be lower if we move the sawmill to the nearest tree up.

Before we describe the algorithm let us introduce a few useful definitions. Let $Cost(a, b)$ denote the total cost of transportation when the two additional sawmills are located at trees a and b, $a < b$. Moreover, assuming that the lower additional sawmill is in point i, let the optimal position of the second one be called $opt(i)$ (when there are several optimal positions then $opt(i)$ denotes the smallest of them). What we have to do is to find a pair a, b such that $Cost(a, b)$ is minimized.

Our first goal is to learn how to compute $Cost(a, b)$ quickly. Suppose we store the following values:

- $D(i)$ – the distance from tree 1 to tree i (with $D(n+1)$ being the distance from tree 1 to the sawmill in the valley),
- $W(i)$ – the sum of weights of trees $1, \ldots, i$,
- $TotalCost$ – the cost of transportation of all trees to the sawmill in the valley (when there are no additional sawmills).

All these values can be computed in $O(n)$ time without much effort – simply by moving from tree 1 downwards and adding successive weights and distances.

Let us now observe that after this preprocessing one can compute values of $Cost(a, b)$ in $O(1)$ time for given a, b. The following, easy to verify, formula holds:

$$Cost(a, b) = TotalCost - W(a) \cdot (D(b) - D(a)) - W(b) \cdot (D(n+1) - D(b)).$$

$O(n^2)$ algorithm

At this moment we are ready to give an algorithm that works in $O(n^2)$ time. It does the linear-time preprocessing and finds the optimal placement of the sawmills simply by checking all $\binom{n}{2}$ possible pairs (each pair takes $O(1)$ time). This, however, is not quick enough. We shall make an observation thanks to which we will be able to reduce the number of pairs to check.

Key observation

Our key observation is as follows:

$$\text{for any } i < j \text{ there is } opt(i) \leq opt(j).$$

In other words, if we move the lower sawmill down, then the optimal location of the upper sawmill cannot move up.

It is not difficult to prove this fact. First observe the following lemma:

Lemma 1: *For any $k < k'$, $i < j$ if $Cost(k', i) \leq Cost(k, i)$ then $Cost(k', j) \leq Cost(k, j)$.*

Proof: The inequality $Cost(k', i) \leq Cost(k, i)$ means that it is cheaper to transport trees $1, \ldots, k$ from point k to k' than to transport trees $k+1, \ldots, k'$ from point k' to i. Clearly transporting the trees $k+1, \ldots, k'$ from point k' to j would be at least that expensive. It follows that $Cost(k', j) \leq Cost(k, j)$. ■

Now, let us put $k' = opt(i)$ in the above lemma. $Cost(k', i) \leq Cost(k, i)$ holds by the definition of k'. The lemma implies that any k such that $k < k'$ is not $opt(j)$ (because it introduces higher cost than k'). This proves our key observation.

Now let us see how to use this fact.

Faster algorithm

Assume that we have calculated $a^* = opt(\lfloor n/2 \rfloor)$. We now know, that for any $i < \lfloor n/2 \rfloor$ we have $opt(i) \leq a^*$ and also for any $j > \lfloor n/2 \rfloor$ we have $opt(j) \geq a^*$. Thus we can write a recursive procedure $Solve(a_{min}, a_{max}, b_{min}, b_{max})$ which finds $a \in \{a_{min}, \ldots a_{max}\}$ and $b \in \{b_{min}, \ldots b_{max}\}$ such that $Cost(a, b)$ is minimized. It calculates the value of $a^* = opt(\lfloor (b_{min} + b_{max})/2 \rfloor)$ simply in $O(a_{max} - a_{min})$ time. Then it calls recursively $Solve(a_{min}, a^*, b_{min}, \lfloor (b_{min} + b_{max})/2 \rfloor - 1)$ and $Solve(a^*, a_{max}, \lfloor (b_{min} + b_{max})/2 \rfloor + 1, b_{max})$. Obviously, the general task is solved by calling $Solve(1, n, 1, n)$.

Time complexity

We see that the number of pairs checked is reduced. Let us estimate it. Consider the binary tree of recursive calls. Its depth is bounded by $\lceil \log n \rceil$ because the value of $b_{max} - b_{min}$ is divided by 2 in each recursive call. At each level of the recursion the possible n values of a is divided into several segments such that two successive segments have exactly 1 common value. It follows that the total number of values of a considered at each level of the recursion is $O(n)$. Thus the total time spent on computing $opt(\cdot)$ is $O(n)$ for each level. It shows that the time complexity of our algorithm is $O(n \log n)$.

Other Solutions

There are quite a few different approaches to this problem (as well as its generalizations), not all correct. Let us take a closer look at a few of them.

Naive solutions

Solutions with time complexity $O(n^3)$ or even $O(n^4)$ can easily be achieved by the contestants. What they do is test all possible locations of sawmills, but compute the cost for them with slow algorithms — in linear or even (pure brute-force) quadratic time.

An $O(kn^2)$ Solution Working for k Sawmills

The following solution computes optimal cost when k sawmills have to be located. For any $i = 1, \ldots, n$, $l = 1, \ldots, k$ let $T[i, l]$ denote the optimal cost of locating l sawmills when we consider only trees from 1 to i (we assume that there is one more sawmill at the position of $(i+1)$-th tree). We use the dynamic programming approach. The value of $T[i, l]$ can be computed in $O(i)$ time: we check all i possibilities of placing the last sawmill using stored values of $T[1, l-1], \ldots T[i-1, l-1]$. The overall time complexity is $O(kn^2)$ which gives another $O(n^2)$ algorithm for our case where $k = 2$.

Incorrect Greedy Solution

We start with finding an optimal position for one sawmill in $O(n)$ time. One of the sawmills will be located at that position. Then the other sawmill is placed in the position that minimizes the cost of transportation. The overall time complexity is $O(n)$. One can easily find a test case where this greedy approach does not work.

A linear algorithm

The problem in more general setting is widely known as the k-median problem, where one has to choose k vertices for *facilities* in a given graph with weights at vertices and edges subject to minimize the total cost, i.e. the sum for all vertices of distance to the closest facility multiplied by the weight of the vertex. The problem is NP-complete for general graphs (undirected as well as directed) and polynomial for trees. For a directed path there is an $O(nk)$ algorithm[1]. It is, however, far too complicated to consider it for a programming contest.

[1]See V. Auletta, D. Parente and G. Persiano, *Placing Resources on a Growing Line*, J. Algorithms, vol. 26, no. 1, 1998, pages 87-100.

ISBN 83–917700–6–0