In the second run, we introduce two new pointers r and s, initially pointing to x_{-1}. At first, we apply, k times, the assignment $s := f(s)$. Next, we move both pointers one step forward ($r := f(r)$, $s := f(s)$) until we reach a moment when $f(r) = f(s)$. Then we have exactly $r = x_a$ and $s = x_b$, since s took k steps more than r.

```
p, q := x_0
r, s := x_-1
while p ≠ q do
    p := f(p)
    q := f(f(q))
    s := f(s)
while f(r) ≠ f(s) do
    r := f(r)
    s := f(s)
return (g(r), g(s))
```

The time complexity of this solution is linear in the number of elements of (x_i) that we inspect before the first collision is found, that is, $O(\sqrt{n})$ in the expected case, with constant space usage.

In the whole analysis we assumed that computing the values of *hash*, g and f would each take constant time. This makes some sense, as all the strings for which these functions are applied are relatively short.

JAKUB ŁĄCKI

Jakub Łącki is a Ph.D. student at the Institute of Informatics, University of Warsaw. His research focuses on dynamic graph algorithms and algorithms on planar graphs. He has won the Polish Collegiate Programming Contest twice, and has been a runner-up in the ACM-ICPC Central European Regional Contest two times as well. Since 2006 he has co-organized an annual Polish Olympiad in Informatics Training Camp, and since 2009 has also been responsible for the selection of tasks for Algorithmic Engagements. He participated in the organization of the Baltic Olympiad in Informatics in 2008 and the Central European Olympiad in Informatics in 2011.

Jakub Łącki is a passionate traveler. In his free time he plays the guitar or soccer.

/ Afternoon *Tea*

Contest: Polish Collegiate Programming Contest 2011
Task author: Tomasz Idziaszek
Memory: 32 MB
http://main.edu.pl/en/archive/amppz/2011/her

During his visit to the Bytic Islands, Byteasar really enjoyed the national beverage of Byteans: that is, tea with milk. This drink is always prepared in a strictly determined manner, which is as follows. Firstly the teacup is filled with tea mixed half and half with milk. Then an n-letter ceremonial word consisting of the letters **H** and **M** is chosen. Now, for $i = 1, 2, \ldots, n$, the following action is performed: if the i-th letter of the ceremonial word is **H**, one should drink half of the teacup, add tea until the teacup is full, and stir. On the other hand, if the i-th letter of the word is **M**, one should perform a similar action, but milk should be added instead of tea. After such action is performed for each letter of the ceremonial word, the remaining liquid is disposed of.

Each time Byteasar performs the ceremony, he wonders which of the ingredients he has drunk more of: tea or milk. Help Byteasar to answer this question.

Input

The first line of input holds an integer n ($1 \le n \le 100{,}000$). The second line contains an n-letter word consisting of the letters **H** and **M**; this is the ceremonial word used by Byteasar.

Output

Your program should output a single letter **H** if Byteasar has drunk more tea than milk; a single letter **M** if he has drunk more milk than tea; or the word **HM** if he has drunk equal amounts of tea and milk.

Example

For the input data:	the correct result is:
5 HMHHM	H

Explanation of the example: Byteasar has drunk $1\frac{37}{64}$ teacups of tea and $\frac{59}{64}$ teacups of milk in total.

/ Solution

It was a few weeks before the Polish Collegiate Programming Contest that Tomasz Idziaszek told me about his idea for the task. It did not seem new to me. When I was in elementary or perhaps junior school, during a math lesson I tackled a similar problem in which a boy was drinking a mixture of two beverages and filling his glass. That version was much easier, however: the boy filled the glass only three times and drank everything at the end. It is a bit embarrassing to confess that I laboriously added up fractions that became more and more complicated after every turn, and calculated how much the boy drank every time. I did not realize that it was sufficient just to keep track of the beverage that he pours into his glass...

I did not do much better with the task about Byteasar either! This time I knew that counting what happens at each step is probably not the best idea. Also, it was obvious that simulating the process by using the floating-point types (e.g. *double*) that are embedded in the majority of programming languages would not be appropriate: their precision is far too limited for manipulating the fractions that emerge in this task.

So how can we approach this task? It gets a bit easier if we change our perspective a little. Instead of calculating the amount of tea and milk that were poured into the glass, let us consider the amount that Byteasar does not drink. In other words, we focus on calculating what remains in the cup after the ceremony. Since we know how many cups of each beverage were poured into the cup during the ceremony, this information is enough to answer the question. Moreover, in order to avoid working with fractions, we assume that the volume of the cup is 2^{n+1} milliliters and that it is initially filled with 2^n milliliters of milk and the same amount of tea.

Let us focus on the case when, in the first turn, Byteasar pours in 2^n milliliters of tea. How much of this tea will remain in the cup until the end of the ceremony? In the second turn, Byteasar drinks half of this tea, so there are 2^{n-1} milliliters remaining. After the third turn, 2^{n-2} milliliters remain, and so on. After n turns—that is, $n-1$ dilutions—in the cup there will be 2 milliliters of tea that come from the first turn.

The liquid poured in the second turn is diluted $n-2$ times, so it determines 4 milliliters of the final contents of the cup. In general, from the i-th portion, 2^i milliliters last until the end. Apart from that, there is also 1 milliliter of tea and 1 milliliter of milk, which were both poured at the very beginning.

For the sample ceremonial word **HMHHM**, at the end there are $1+2^1+2^3+2^4$ milliliters of tea and $1+2^2+2^5$ milliliters of milk in the cup.

This allows us to express the final contents of the cup as a sum of powers of 2. We then have to add all these numbers, which in some programming languages requires us to implement big-integer arithmetic. Similarly, we can count the amount of tea and milk that Byteasar pours into the cup. At the end, it is sufficient to perform two subtractions to discover how much of each beverage Byteasar has drunk.

This allows us to reach the solution. However, it turns out that we do not have to perform complicated calculations. Recall that we only have to determine which beverage Byteasar has drunk more of, so we do not have to calculate the exact numbers.

A better solution

From the description, assume that Byteasar does not perform the last pouring, so the cup is half empty after the ceremony. This does not change anything, as Byteasar does not drink what he pours at the end.

Let us denote by t_p and t_c the amount of tea poured into the cup and the amount that remained in the end. Similarly, let m_p and m_c denote equivalent values with regard to milk. We know that Byteasar has drunk $t_p - t_c$ milliliters of tea and $m_p - m_c$ milliliters of milk. At the end, the cup is half full, so $t_c + m_c = 2^n$.

We first focus on the case where there are more letters of one kind than the other in the ceremonial word. It turns out that in such a situation Byteasar drinks more of the beverage that was poured into the cup more often. Assume that he poured more tea, so t_p is larger than m_p by at least half of the cup (2^n milliliters), so $t_p \geq m_p + 2^n$. To compute the amount drunk by Byteasar, we have to subtract the final contents of the cup: that is, t_c and m_c. The cup, however, is half empty, and we know that it contains at least 1 milliliter of each beverage (coming from the first pouring). In particular, there are at most $2^n - 1$ milliliters of tea remaining in the cup ($t_c \leq 2^n - 1$). Hence, the amount of tea drunk by Byteasar is bounded as follows:

$$t_p - t_c \geq t_p - 2^n + 1 \geq m_p + 2^n - 2^n + 1 = m_p + 1.$$

This means that the amount of tea Byteasar has drunk is greater than the total amount of milk he has poured into the cup. Consequently, when the ceremonial word contains more **H**s than **M**s, Byteasar drinks more tea. Analogously, if the word contains more **M**s, Byteasar drinks more milk.

It remains to consider the case where the numbers of **H**s and **M**s are equal, so the cup is filled with the same amount of each of the beverages. In particular, this happens when $n = 1$. This is a special case, in which Byteasar drinks only half of the cup containing tea mixed half and half with milk. So he drinks an equal amount of each beverage.

Let us now assume that $n > 1$. If the amounts of milk and tea poured into the cup are equal, Byteasar drinks more of the beverage whose remaining amount in the cup is *smaller*. We know that half of the final cup contents comes from the next-to-last pouring (the last that we have not ignored). Assume that Byteasar poured milk then. In addition, in the cup there is 1 milliliter of milk poured at the very beginning. We infer that there has to be more milk than tea in the cup, so Byteasar has drunk more tea.

Therefore, the entire algorithm boils down to three simple cases. If $n = 1$, Byteasar drinks an equal amount of both beverages. If one letter is more common than the other in the ceremonial word (not considering its last letter), then Byteasar drinks more of the beverage represented by the more common letter. If the two letters appear in the same quantities, then we know that Byteasar drinks less of the beverage that he poured in the next-to-last turn.

/ Kangaroos

Contest: Algorithmic Engagements 2011
Task authors: Jakub Łącki, Jakub Radoszewski
Memory: 32 MB
http://main.edu.pl/en/archive/pa/2011/kan

Byteasar, who is interested in wildlife photography, is planning a trip to Australia. He has started preparing his photographic equipment for taking pictures of kangaroos, and has to decide what items to pack. His equipment includes an extensive collection of photographic lenses with varying characteristics for different angles of view and focusing abilities. The best quality of pictures of animals, such as kangaroos, is achieved when the distance between the camera and the animal is within the optimal range of the lens.

Byteasar's trip passes through a sequence of access points for wildlife observation and photography. The Australian guides gave Byteasar a detailed description of these observation points—in particular, the range of distances at which kangaroos can be expected.

Obviously, not all the lenses from Byteasar's large collection will fit in his luggage, so some decisions must be made. Since Byteasar wishes to keep his lens-changes to a minimum, he wants to find—for each of the lenses—the longest contiguous sequence of observation points on the trip for which that lens is suitable. A lens is suitable for a given observation point if there exists a distance from the range of expected distances that lies within the optimal range of that lens.

Input

The first line of the input contains two integers, n and m ($1 \le n \le 50{,}000$, $1 \le m \le 200{,}000$), where n is the number of observation points on the trip and m is the number of lenses in Byteasar's collection.

The next n lines describe the access points for wildlife observation and photography on Byteasar's trip. Each of these lines contains two integers, a_i and b_i ($1 \le a_i \le b_i \le 10^9$), indicating that at the i-th observation point kangaroos can appear at a distance of a_i to b_i bytean feet, inclusive.

The next m lines describe the lenses. Each of these lines contains two integers, c_j and d_j ($1 \le c_j \le d_j \le 10^9$), indicating that the j-th lens works best for kangaroos that are at a distance from c_j to d_j bytean feet from the camera, inclusive.

Output

The output consists of m lines containing exactly one integer each. The j-th output line indicates the number of observation points in the longest *contiguous* segment of the trip where the j-th lens can be used by Byteasar. The lenses are numbered according to their order in the input file.

Example

For the input data:

```
3 3
2 5
1 3
6 6
3 5
1 10
7 9
```

the correct result is:

```
2
3
0
```

/ Solution

"You are given two sequences of intervals. Find the longest subsequence of consecutive intervals in the first sequence, which..."—it seems we have another task about intervals to deal with. We need to sort something, and perhaps use an appropriate data structure, then iterate through the intervals in the right order and we are done. I believe that a simple task statement implies a simple solution, and this task seemed to match this rule perfectly. It was supposed to be used in the final round of Algorithmic Engagements in 2010; however, it was not, and the reason was quite serious: we could not solve it!

Only in the following year, when preparing the successive competition, did I manage to find an algorithm that was better than a naïve one. The task was then prepared for the contest by Adam Karczmarz, and he found the best solution known to us. Our approaches differ significantly, and I think that both deserve to be described here. The former is quite general, and uses a technique that can come in handy in other situations. The latter, on the other hand, leads to a faster algorithm and, while it is quite complex, is built from very simple blocks.

The first approach

We are given two sequences of closed intervals:

$$[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n] \text{ and } [c_1, d_1], [c_2, d_2], \ldots, [c_m, d_m].$$

As in the task statement, the first of them will be called the sequence of observation points, and the second, the sequence of lenses. For simplicity we treat each lens and observation point as an interval. Because of that, we will use some rather odd (but convenient) terminology, such as "the intersection of a lens with an observation point". What we mean in such cases is obviously the intersection of the intervals corresponding to the lens and observation point. The whole task can be summarized in a single sentence. For each lens, find the longest subsequence of consecutive observation points that intersect with the lens.

We first note that the most natural approach works in $O(mn)$ time. In this approach, for each of the m lenses $[c_i, d_i]$ we find the longest subsequence of consecutive observation points that intersect with the lens. This can of course be done in $O(n)$ time.

We now present a solution that is even slower than that, but we later show how to speed it up significantly. We use a data structure that represents a subset of integers from 1 to n. The data structure allows us to add and remove its elements. Moreover, it can report the longest sequence of consecutive integers that it contains. For example, if the structure contains numbers 3, 4, 7, 11, 12, 13, 25, 26, then the sequences of consecutive integers are (3,4), (7), (11,12,13), (25,26), and the longest among them has length 3. It's not hard to guess that we will be inserting indices of the observation points into this structure.

Finding sequences of consecutive integers. How can we implement this data structure? There are many possibilities. One of them is to use an appropriately tuned segment tree. Another is to use any data structure that represents a sorted set of objects and implements the basic set of operations. We choose the second option, as it is a bit simpler to describe*.

Let A denote the subset of integers from 1 to n, which is to be stored in the data structure. We represent A as a collection of sequences of consecutive integers; for example, set $\{1,3,4,5,7,8,12\}$ is represented by $[1,1]$, $[3,5]$, $[7,8]$, $[12,12]$. These intervals will be stored in sorted order in a data structure F_A. We can sort them by their left or right ends—this makes no difference, as the intervals are disjoint. In addition, we have a second data structure D_A that stores the set (or, rather, a multiset, since we can have many copies of the same element) of the lengths of intervals describing set A. In our example, the lengths are 1, 3, 2 and 1.

Querying for the longest subsequence of consecutive integers in A boils down to finding the maximum in D_A.

When we add an integer k to A, we add interval $[k,k]$ to F_A and find the interval that is just before and after it in the sorted order. If we find an interval $[r, k-1]$ or $[k+1, s]$, then the appropriate intervals have to be merged—it is best just to delete them and insert an interval formed by merging them together.

When a given element, say k, is deleted from A, we do a similar thing: find an interval $[r,s]$ in F_A that contains k and delete it, and then insert intervals $[r, k-1]$ and $[k+1, s]$ into F_A, provided that they are nonempty.

Observe that, in all these operations, only at most three intervals are involved, which allows us to update D_A easily. Each time we insert or delete an interval to/from F_A, we need to add or remove its length to/from D_A. Using an efficient implementation of sorted dictionaries, each operation on both structures F_A and D_A can be carried out in $O(\log n)$ time.

* And it can be implemented using the set template from STL.

The general idea. Let us go back to the task. Denote by P_i the data structure we have just described, filled with indices of the observation points that intersect with lens $[c_i, d_i]$. It takes $O(n)$ operations on the data structure to build it, so the required time is $O(n \log n)$. In order to get the solution of the entire task, we need to know P_i for all i from 1 to m, which, with the straightforward approach, takes $O(nm \log n)$ time. This does not look impressive, especially if we recall that the naïve solution is faster. However, we will show that this new approach can be speeded up considerably.

The high-level idea is as follows. Assume that after obtaining P_j, we would like to compute P_i. In addition, let us assume that the set represented by P_i differs only a little from the set represented by P_j. Thus P_i can be easily computed—we can take P_j and, after a small number of add or delete operations, get P_i.

This would work nicely if only P_i and P_j were similar to each other. How can we guarantee that? Observe that the structures P_i can be computed in any order, as long as we iterate through all of them. As we will show in a moment, for *some* order of constructing the data structures, the total number of differences between consecutive structures is not too big.

But before we show the above property, let us think how to find the observation points that should be erased from P_j or added to it, in order to obtain P_i. Denote by $[c, d]$ the lens represented by P_j; that is, assume that P_j contains the indices of observation points that intersect with $[c, d]$. On that basis we now want to compute the P_i that represents the lens $[c', d']$. For presentation, let us assume that $c < c'$ and $d < d'$.

Suppose that $c' \le d$, so the intervals intersect. We first build a data structure representing $[c, d']$. We move the right end of our interval to d' and find all observation points that we encounter on our way; that is, the ones that should be added to our data structure. It should be clear that these are exactly the observation points whose left end a satisfies $d < a \le d'$. Next, we move the left end c right to c'. We now have to get rid of all the observation points that no longer intersect with our lens, namely the ones whose right end b satisfies $c \le b < c'$. Analogous reasoning can be applied to $d < c'$; that is, when the intervals are disjoint. In this case, the intervals fully contained in (d, c') will first be added to the data structure, only to be deleted in a moment.

All this can easily be generalized to other cases of the mutual relations between c and c' and between d and d'. To sum up, it is sufficient to iterate through all intervals whose right end is contained between c and c', and whose left end is contained between d and d'.

Transforming the input data. How many intervals have the aforementioned property? It is difficult to say, as it depends on the input data. But we can cope with that. Observe that the input data can be freely modified, as long as the smaller/greater relationships between the numbers are preserved. We transform the input data so that it will be easier to bound the number of differences between data structures P_i. The outcome will be that, in order to obtain a data structure representing $[c', d']$ from a data structure representing $[c, d]$, at most $|c - c'| + |d - d'|$ insertions or deletions will have to be made. We start by subtracting $\frac{1}{2}$ from every c_i and adding $\frac{1}{2}$ to every d_i. This means that we extend the intervals a bit. It is easy to see that the operation does not influence the intervals' intersections. Second, we "compress" all coordinates: that is, we renumber all a_i and b_i, maintaining their mutual relations, so that they become distinct integers from 1 to $2n$. For example, the set of intervals $[1,5]$, $[1,7]$, $[3,10]$ can be transformed into $[1,4]$, $[2,5]$, $[3,6]$. In order to achieve this, all a_i and b_i have to be sorted (together) and then the number from the i-th position is replaced by i.

Since we have changed the ends of intervals $[a_i, b_i]$, we also have to move c_i and d_i, to restore their original relations to a_i and b_i. Note that we maintain the fractional parts of c_i and d_i, which are equal to $\frac{1}{2}$.

All these transformations can be implemented in $O((n+m) \log n)$ total time. We leave the details to the reader.

After performing all the above transformations, for every integer k from 1 to $2n$ there is exactly one interval $[a_i, b_i]$, whose end is k. Hence, if we have built a data structure that represents $[c, d]$ and we now want to compute the data structure for $[c', d']$ (c, c', d, d' now denote the transformed values), then the number of intervals $[a, b]$ that satisfy $d < a \le d'$ is at most $d' - d$. Similarly, there are at most $c' - c$ intervals satisfying $c \le b < c'$. This can be generalized to other cases of mutual relations between c and c' and between d and d'. In the end, the outcome is that building a data structure for interval $[c', d']$, given a data structure for interval $[c, d]$, requires at most $|c - c'| + |d - d'|$ insert or delete operations.

Geometric problem. The observation made in the previous paragraph gives us a simple geometric interpretation of our problem. For each interval $[c_i, d_i]$ we draw a point (c_i, d_i) in the plane. Our goal is to draw a broken line that visits all these points and is as short as possible. Additionally, since we measure the distance according to the Manhattan distance—that is, the distance between (x, y) and (x', y') is equal to $|x - x'| + |y - y'|$—the broken line has to consist solely of horizontal and vertical segments. If we find a broken line of length l,

then the consecutive points it visits will correspond to the order in which we should construct data structures P_i. We know that in this case we will need to perform at most n operations to build the initial data structure, and then at most l operations to build all structures, basing them on the previous ones. All the structures are computed in $O((n + l) \log n)$ time in this way. It remains for us to show that we can always find a sufficiently short broken line.

We will solve a slightly more general problem. Let us assume that we are given m points in a square of size $n \times n$. Our points lie in a square that is a bit bigger than $2n \times 2n$, but this does not matter from the asymptotic notation's point of view. We also do not use other properties of the points, although we know that, for example, $c_i \leq d_i$.

The first idea is to try to find the shortest broken line possible. This, however, probably cannot be done in polynomial time, as this problem is NP-complete. But this is not a big issue—our problem has a nice property that if the optimal length of the broken line is l_{opt}, and we find an algorithm that always finds broken lines of length, say, at most $10l_{opt}$, then the overall running time complexity will be the same. All in all, since the running time is $O((n + l) \log n)$, increasing l ten times does not increase the asymptotic complexity.

It turns out that we can indeed find the broken line that has this nice property. We will describe it briefly, as the approach we take in the end will be different. It is sufficient to connect all m points into a minimum spanning tree (in the plane with Manhattan distance this can be done in $O(m \log m)$ time), root the tree arbitrarily and construct the broken line by connecting the consecutive nodes visited during a preorder traversal of the tree. It can be shown that the broken line constructed in this way is at most twice as long as the optimal one.

Thus we obtain a rather unusual algorithm. We first solve an optimization problem and then, the better the solution we find, the faster the second phase will be. We have only given a high-level description of the algorithm because there is also a simpler way of finding a good broken line. While it can sometimes give much worse results, both approaches work similarly well in the pessimistic case.

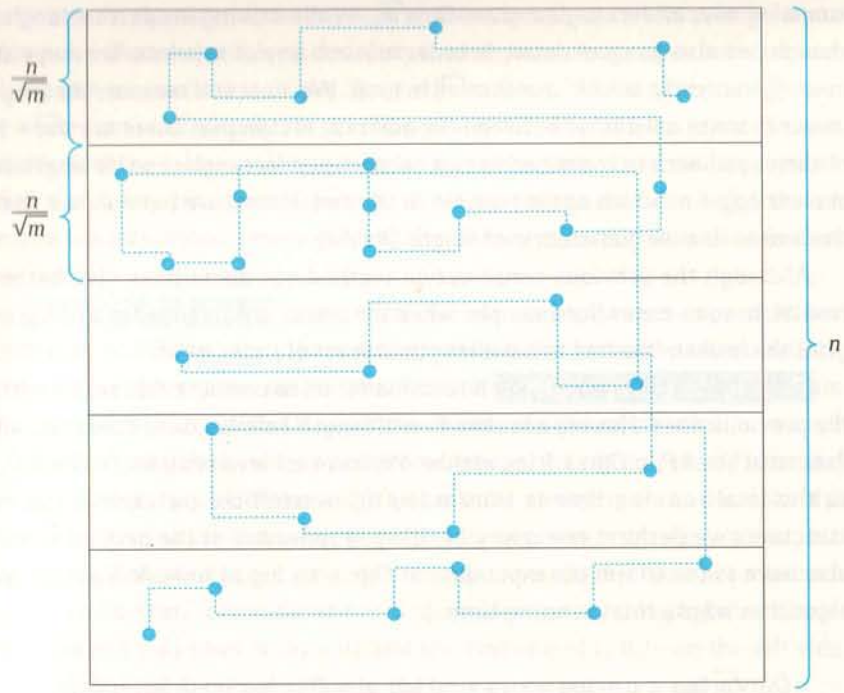


Figure 1: Finding a broken line of length $O(n\sqrt{m})$.

Let us divide our $n \times n$ square into $\sqrt{m}$ rectangles of size $\frac{n}{\sqrt{m}} \times n$, as shown in figure 1. The broken line is constructed as follows. Visit all points in the bottommost rectangle in increasing order of x coordinate. Then go to the point with the smallest x coordinate in the rectangle above, and go through all points in this rectangle in a similar way; and so on. We stop after visiting all the points.

Inside each rectangle the broken line goes only right, up or down. The total length of the segments going right inside one rectangle is at most n, which, after summing over all rectangles, gives $O(n\sqrt{m})$. While moving inside a rectangle, though, we also go up or down. In order to reach any of m points, we move at most $\frac{n}{\sqrt{m}}$ vertically, so also use $O(n\sqrt{m})$ in total. We must still measure the fragments that are used to go between the adjacent rectangles. There are $\sqrt{m}-1$ of them, and each is contained in two consecutive rectangles, so its length is at most $2\frac{n}{\sqrt{m}}+n$, which again sums up to $O(n\sqrt{m})$. Hence, we have shown that the broken line we construct is of length $O(n\sqrt{m})$.

Although the previous construction method can sometimes give better results, in some cases (for example, when the points are arranged in a uniform grid) the broken line has to have length at least of order $n\sqrt{m}$.

Putting the pieces together. It remains for us to combine this result with the previous ones. Having a broken line of length l allows us to construct all data structures P_i in $O((n+l)\log n)$ time. We have achieved that for $l = O(n\sqrt{m})$, so the total running time is $O(n\sqrt{m}\log n)$. In addition, for each of the m structures we perform one query in $O(\log n)$ time, and at the beginning we also have to transform the input data in $O((n+m)\log n)$ time. We obtain an algorithm whose total running time is

$$O(n\sqrt{m}\log n + m\log n + (n+m)\log n) = O(m\log n + n\log n\sqrt{m}).$$

The technique we have used is quite general, and can also be used in other problems, especially if we have to answer queries about intervals. One example is the task *Powerful Array* from the Codeforces contest in 2011.

Our approach is better than the $O(nm)$ solution by about $O(\frac{\sqrt{m}}{\log n})$. While this is a significant improvement from a theoretical point of view, for the sizes of input data we often deal with, the numerator is only a few times greater than the denominator, so the gain is moderate. However, the problem can be solved even more efficiently by using a completely different method.

The second, more efficient solution

This time we take a different approach. We break the problem into two subproblems that we will solve independently.

Assume that the input data have already been transformed, as described in the previous solution.

Let us color every t-th observation point blue: that is, we mark points $[a_t, b_t]$, $[a_{2t}, b_{2t}]$,...; there are $\lfloor \frac{n}{t} \rfloor$ of them in total. Now, for each lens, we first find the longest subsequence of consecutive observation points that includes some blue point (first subproblem) and then the longest subsequence without any blue point (second subproblem). As it will turn out, the first subproblem can be solved more efficiently for greater values of t, whereas the more efficient solutions of the second problem exist for smaller values of t. For an appropriate choice of t, we will obtain solutions to both problems that can be combined into an algorithm that is more efficient that the one resulting from the previous approach.

The first subproblem. In this case we use a very natural approach. For each lens $[c_i, d_i]$ and every blue point j we find the longest subsequence of consecutive observation points that contains the j-th point and only intervals that intersect $[c_i, d_i]$. We will call it the solution anchored at point j.

We first compute how many consecutive intervals to the left of j (starting from j) intersect $[c_i, d_i]$. Then we do the same for the intervals to the right of j. These two steps are analogous, so we now focus on the latter. For a given j we want to find the first index $j' > j$, such that interval $[a_{j'}, b_{j'}]$ does not intersect $[c_i, d_i]$. To do that, it is sufficient to find the first interval $[a_p, b_p]$ that lies to the right of $[c_i, d_i]$ (that is, $a_p > d_i$) and the first one—$[a_q, b_q]$—on the left side of $[c_i, d_i]$ ($b_q < c_i$). It is then the case that j' is a minimum of p and q.

In the end, for a fixed blue point in which a solution is anchored, we solve the following problem.

Problem 1. You are given an array of integers $a_1,\ldots,a_n$, where $1 \le a_i \le 2n$, and a sequence of integers $c_1,\ldots,c_m$. For each c_i find the first index j, such that $a_j > c_i$.

We can tackle this easily. We first compute an array p, where $p[i]$ is the first position of number i in array a (or ∞ if i is not in array a). This allows us to compute array r, in which $r[i]$ is the first index in array a, such that $a[r[i]]$ is greater than i. It is sufficient to set $r[i]$ equal to $\min(p[i+1], p[i+2],\ldots,p[2n])$. This can of course be done in linear time, since $r[i]$ can be computed using $r[i+1]$; namely $r[i] = \min(r[i+1], p[i+1])$. Having array r, we can immediately find the answer for each c_i. Thus, solving this problem requires $O(n+m)$ time.

We use it four times for each blue point, to find a greater/smaller number to the left/right of the given blue point. The overall running time for a single blue point can be bounded by $O(n+m)$, so the entire first subproblem can be solved in $O(\frac{n}{t}(n+m))$ time.

The second subproblem. It remains for us to solve the second subproblem. This time, for each lens we are looking for a solution that is not anchored at any blue point. If the value of t is small, there are not too many candidate intervals to consider. The blue points divide the sequence of observation points into $\lfloor \frac{n}{t} \rfloor + 1$ intervals of length smaller than t. In an interval of length t we can choose a subinterval in $O(t^2)$ ways; thus in this subproblem the solution for each lens is chosen from among $O(\frac{n}{t} \cdot t^2) = O(nt)$ candidates.

In this case we assume that nt is rather small. All candidates have lengths from 1 to $t-1$. Let us divide them into groups based on their lengths. We will process these groups in order of decreasing lengths of intervals. If we find an interval which is good for some lens, then we immediately have the answer for that lens, so we can remove the lens from further consideration. This is because we check the longer candidates first.

Let us give some more details. Again we will use...

Geometric interpretation. Consider a sequence of observation points $[a_r, b_r]$, $[a_{r+1}, b_{r+1}], \ldots, [a_s, b_s]$. This is a good sequence for lens $[c, d]$, if every observation point intersects the lens, so for each $r \le i \le s$ we have $b_i \ge c$ and $a_i \le d$. This can be equivalently rewritten as

$$\min_{r \le i \le s} b_i \ge c \quad \text{and} \quad \max_{r \le i \le s} a_i \le d.$$

This allows us to draw our problem in the plane. We assign to the aforementioned sequence of observation points the point

$$(\min_{r \le i \le s} b_i, \max_{r \le i \le s} a_i).$$

On the other hand, for each lens $[c_i, d_i]$ we draw a point (c_i, d_i). We can now see that the sequence of observation points represented with a point (B, A) is good for lens (c, d) if $B \ge c$ and $A \le d$, so (B, A) lies to the right and down of (c, d). Thanks to that, solving this part boils down to the following problem.

Problem 2. In the plane, there are g green points (b_j, a_j) and m red points (c_i, d_i). All the points have integer coordinates from 1 to $2n$. The goal is to find all red points (c_i, d_i) such that there exists some green point (b_j, a_j) lying to the right and down of (c_i, d_i); that is, $b_j \ge c_i$ and $a_j \le d_i$.

We begin by getting rid of all useless green points. If, for some green point, there exists another green point that lies to the right and down of it, then the first point can be deleted. This can be achieved with a simple algorithm. First, inside each group of points of equal x coordinate we leave only the point with the smallest y coordinate—it is clear that the other points are not useful. Then we sort all green points *decreasingly* with respect to the x coordinate and go through them one by one. We delete all the points whose y coordinate is not smaller than the smallest y coordinate of the points we have already visited. This process eliminates all useless points. As the coordinates are at most $2n$, this step can be implemented to run in $O(n+g)$ time. After that, at most $2n$ useful points remain, since each of them has to have a different x coordinate. We encourage the reader to show the correctness of this algorithm.

Although getting rid of the useless points may seem redundant, it will allow us to solve Problem 2 more effectively. We insert all red points into $2n$ stacks $S_1, \ldots, S_{2n}$. In stack S_i we have points whose x coordinate is equal to i. Inside each stack we sort the points increasingly: that is, on the top we put the point with the greatest y coordinate. It is easy to see why we choose this particular ordering: if some green point is good for some point in the stack, then it is also good for all higher points in the stack. This means that we can process points in every stack starting from the top.

The remaining part is simple. Let (b_1, a_1), $(b_2, a_2), \ldots$ denote the set of green useful points sorted decreasingly by their x coordinate. Since they are all useful, we also know that they are sorted decreasingly by their y coordinate. We now find all red points in stacks $S_{b_2+1}, \ldots, S_{b_1}$, for which point (b_1, a_1) is good. These points are removed from the stacks. Then we consider the point (b_2, a_2) and stacks $S_{b_3+1}, \ldots, S_{b_2}$ and so on. Each time we check a point on a stack, we either delete this point or move to the next stack. Thus, if the number of red points we find equals r, this phase of the algorithm runs in $O(r+n)$ time.

This gives us a solution to Problem 2. This solution has now to be run for each group of intervals of equal length. Observe that it is sufficient to sort all red points and insert them into the stacks only once, at the beginning.

The ingredients of the solution to this subproblem are as follows:

- Generate $O(nt)$ candidate intervals.
- Convert the candidates to points in the plane in $O(n \log n + nt)$ time.
- Insert m lenses into stacks and sort them within each stack, in total time of $O(n + m \log m)$.
- Delete useless points in $O(nt)$ time.

→ Run the algorithm that solves Problem 2 $t-1$ times. In each iteration we use $O(r+n)$ time, where r is the number of lenses for which we want the answer. Hence, the sum of these numbers over all $t-1$ runs of the algorithm is at most m and the total time is $O(m+nt)$.

This gives an overall running time of $O(n \log n + m \log m + nt)$.

Merging the subproblems. We are almost done. Our algorithm first modifies the input in $O((n+m) \log n)$ time, then solves the first subproblem in $O(\frac{n}{t}(n+m))$ and the second subproblem in $O(n \log n + m \log m + nt)$ time. It remains for us to find the optimal value for t.

We would like to minimize $\frac{n}{t}(n+m) + nt$, which is equivalent to minimizing $\frac{n+m}{t} + t$. It is easy to see that the smallest value is attained for $t = \sqrt{n+m}$. The algorithm works within the following time bound:

$$O((n+m) \log n + n\sqrt{n+m} + n \log n + m \log m + n\sqrt{n+m}) =$$
$$= O(n\sqrt{n+m} + m(\log m + \log n)).$$

It takes some time to realize it, but this is indeed better complexity than in the previous solution. In particular, if we assume that—as in the greatest possible input data—$m > n$, then the running time can be rewritten as $O(n\sqrt{m} + m \log m)$.

Some of the contestants obtained slightly poorer solutions to one of the subproblems. For example, if we solved the first subproblem using sorting or some data structure in $O(\frac{n}{t}(n+m) \log n)$, then t would have to be chosen to minimize $\frac{n}{t}(n+m) \log n + nt$ or, equivalently, $t + \frac{(n+m)\log n}{t}$. The best choice of t is then $\sqrt{(n+m)\log n}$ and the algorithm would then run slower by a factor of $\sqrt{\log n}$, in time

$$O(n\sqrt{(n+m) \log n} + m(\log m + \log n)).$$

I guess every computer scientist will occasionally wonder why a particular algorithm has a strange running time, like the fastest known algorithm for counting inversions in a permutation, which runs in $O(n\sqrt{\log n})$ time. The reason is often similar to that presented above. The algorithm consists of two components with different running times, and we decide how to divide the work to be performed between these pieces. We find the optimal way of splitting the work, and it turns out that the running times meet in a rather unexpected place.

KRZYSZTOF ONAK

Krzysztof Onak is currently a Simons Postdoctoral Fellow at Carnegie Mellon University. He earned his Ph.D. degree in computer science at the Massachusetts Institute of Technology in 2010, following a M.S. degree at the University of Warsaw in 2005. He was a member of the team from the University of Warsaw that won the 2003 ACM-ICPC World Finals. Krzysztof Onak's main research interests are focused on sublinear algorithms. In his spare time, he enjoys trying to convince others that he has mastered the saber at a higher level than they have.

/ Sums

Contest: 10th Polish Olympiad in Informatics
Task author: Krzysztof Onak
Memory: 32 MB
http://main.edu.pl/en/archive/oi/10/sum

We are given a set of positive integers A. Consider a set of nonnegative integers A', such that a number x belongs to A' if and only if x is a sum of some elements from A (the elements may be repeated). For example, if $A = \{2, 5, 7\}$, then sample numbers belonging to the set A' are: 0 (the sum of 0 elements), 2, 4 ($2 + 2$) and 12 ($5 + 7$ or $7 + 5$ or $2 + 2 + 2 + 2 + 2 + 2$); and the following do not belong to A': 1 and 3.

Task

Write a program that:

- reads from the input the description of the set A and the sequence of numbers b_i,
- for each number b_i determines whether it belongs to the set A',
- writes the result to the output.

Input

In the first line there is one integer n: the number of elements of the set A ($1 \le n \le 5000$). The following n lines contain the elements of the set A, one per line. In the $(i + 1)$-th line there is one positive integer a_i ($1 \le a_i \le 50{,}000$). $A = \{a_1, a_2, \ldots, a_n\}$, $a_1 < a_2 < \cdots < a_n$.

In the $(n + 2)$-th line there is one integer k ($1 \le k \le 10{,}000$). Each of the following k lines contains one integer in the range 0 to 1,000,000,000: they are respectively the numbers $b_1, b_2, \ldots, b_k$.

Output

The output should consist of k lines. The i-th line should contain the word TAK (Polish for *yes*) if b_i belongs to A', or the word NIE (Polish for *no*) otherwise.

Example

For the input data:

3
2
5
7
6
0
1
4
12
3
2

the correct result is:

TAK
NIE
TAK
TAK
NIE
TAK

Solution

The problem statement may suggest that a solution is based on number theory. However, we reduce the problem to one of the most fundamental graph problems: finding distances in a weighted graph. I believe that this is why many people enjoyed this problem and why I was asked to describe it for this volume. Nonetheless, if you know a number theory solution or any other solution that is more efficient than the one described here, I would love to hear about it.

Reduction to a graph problem

Let us start with a simple observation. Fix an arbitrary number $a \in A$. If an integer y belongs to A', then so do $y+a$, $y+2a$, $y+3a$, and so on. Our goal is to answer queries of the form "Does b belong to A'?" If we knew the smallest number $t \in A'$ such that $b \equiv t \pmod{a}$, then we could determine whether b belongs to A' just by comparing it to t. It holds that $b \in A'$ if and only if $b \geq t$. Note that if such a number t does not exist, we may assume that $t = \infty$ and check the membership of b in A' in the same way.

Therefore, to be able to reply to all queries, it suffices that for each $i \in \{0, 1, \ldots, a-1\}$, we compute the smallest number $t_i \in A'$ such that $t_i \equiv i \pmod{a}$. If a specific t_i does not exist, we set $t_i = \infty$. Note that $t_0 = 0$.

We now construct a weighted directed graph G on a vertices v_i $(0 \leq i \leq a-1)$. Each vertex corresponds to a different remainder modulo a. For each vertex v_i and each $x \in A$, we add a directed edge from v_i to $v_{(i+x) \bmod a}$ of length x. The edge expresses the fact that if we add x to a number with i as a remainder modulo a, then we obtain a number with the remainder $(i+x) \bmod a$. See figure 1 for an example of such a graph G. We show that the numbers t_i can be determined using specific distances in G. The claim follows due to an equivalence between paths starting at v_0 and sequences of numbers in A.

Claim. Let d_i $(0 \leq i \leq a-1)$ be the distance from v_0 to v_i in G (or ∞ if there is no path from v_0 to v_i). For each $i \in \{0, 1, \ldots, a-1\}$, $d_i = t_i$.

Proof. Consider an arbitrary path starting at v_0. Let $a'_1, \ldots, a'_k$ be the lengths of consecutive edges on the path and let $l = a'_1 + \cdots + a'_k$ be the total length of the path. It easily follows by induction that the j-th vertex on the path $(1 \leq j \leq k+1)$ is v_{r_j}, where $r_j = (\sum_{q=1}^{j-1} a'_q) \bmod a$. Therefore, the final vertex of the path is $v_{l \bmod a}$. We use this fact below.

Fix an $i \in \{0, \ldots, a-1\}$. By definition, we can express each $t_i < \infty$ as a sum $a'_1 + \cdots + a'_k$, where each a_j $(1 \le j \le k)$ belongs to A. Consider now a path starting at v_0 that takes k edges, where the j-th edge on the path $(1 \le j \le k)$ has length a'_j. It follows from the definition of G that this path exists. The final vertex of the path is v_i. This proves that $d_i \le t_i$, because the length of this path is t_i.

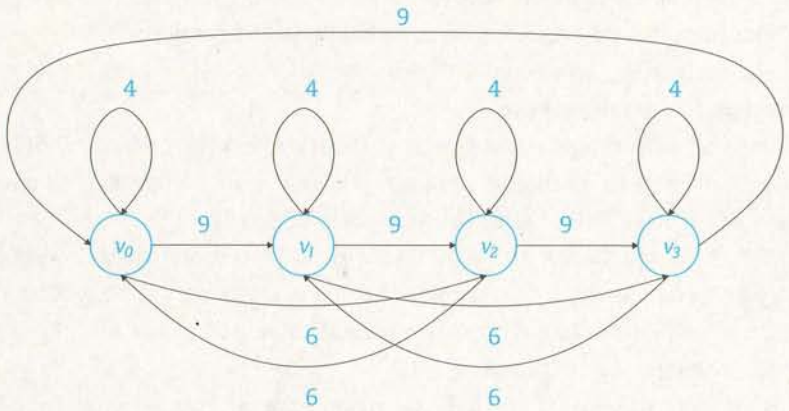


Figure 1: *A sample graph for $A = \{4, 6, 9\}$ and $a = 4$.*

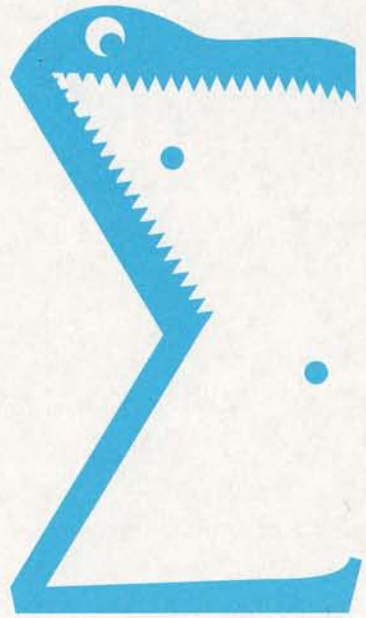
On the other hand, consider the shortest path from v_0 to v_i, assuming that it exists. Recall that d_i is the length of this path. Let $a'_1, \ldots, a'_k$ be the lengths of consecutive edges taken by the path. By the construction of G, all values of a'_j belong to A, so $d_i \in A'$. Moreover, since v_i is the final vertex, $d_i \equiv i \pmod{a}$, so $d_i \ge t_i$.

If one of d_i and t_i is finite, the above reasoning shows that the other number is also finite and equal. Otherwise, $d_i = t_i = \infty$, i.e. the equality holds as well. □

Shortest paths and time complexity

We now know that it is sufficient to compute distances from v_0 to all values of v_i in order to answer all queries b_i efficiently. This can be done using the Dijkstra algorithm. The Dijkstra algorithm is described in almost every textbook on algorithms. Depending on the selection of data structures in the implementation of the Dijkstra algorithm, we may obtain different running times. The most naïve implementation gives a running time of $O(a_1^2 + a_1 \cdot |A|)$, where a_1 is the smallest number in A. If $|A|$ is much smaller than a_1 then it may make sense to use the binary heap as a priority queue in the Dijkstra algorithm, in which case we obtain a running time of $O(a_1 \cdot |A| \cdot \log a_1)$. Note also that we do not have to construct G explicitly, which would require $\Omega(a_1 \cdot |A|)$ space. Instead, it is sufficient to remember A, as all edges incident to a specific vertex are determined by A. Therefore, computing all distances d_i via the Dijkstra algorithm requires only $O(a_1 + |A|)$ space.

Finally, recall that once we have computed all the values $t_i = d_i$, we can answer each query b in constant time. Namely, b belongs to A' if and only if $b \ge t_{b \bmod a}$.



/ Superknight

Contest: 9th Polish Olympiad in Informatics
Task author: Krzysztof Onak
Memory: 32 MB
http://main.edu.pl/en/archive/oi/9/sup

On an infinite chequered board there is a superknight which can make different moves. Each kind of move is described by two integers: the first says how many columns (to the right if the number is positive or to the left if the number is negative) and the second says how many rows (forward if the number is positive or backward if the number is negative) the knight traverses when making such a move.

Task

Write a program that:

- reads from the input sets of data describing different superknights,
- determines for each superknight whether it is able to reach any square of the board by making allowed moves,
- writes the result to the output.

Input

In the first line of the input there is one integer k specifying the number of sets of data ($1 \le k \le 100$). It is followed by k sets of data. In the first line of each set there is an integer n representing the number of kinds of move that the superknight can make ($1 \le n \le 100$). Each of the following n lines of the set contains two integers, p and q ($-100 \le p, q \le 100$), separated by a single space. These numbers describe one kind of move.

Output

The output should consist of k lines. The i-th line should contain one word, TAK (Polish for *yes*), if the superknight described by the i-th set of data is able to reach any square on the board, or the word NIE (Polish for *no*) otherwise.

Example

For the input data:

```
2
3
1  0
0  1
-2  -1
5
3  4
-3  -6
2  -2
5  6
-1  4
```

the correct result is:

```
TAK
NIE
```

/ Solution

I created this problem when I was a freshman at the University of Warsaw. I did not know at that point that there is a whole body of work related to bases of vectors with integer coordinates. Any reader interested in this kind of topic should search online, for instance, for the famous Lenstra-Lenstra-Lovász algorithm.

How does one come up with problems like this? I remember that the idea popped into my head while I was simultaneously watching a ski jumping competition and solving some linear algebra homework. A ski jumper and a chess knight are described in Polish using the same word, "skoczek", which literally means a "jumper".

That competition was won by Adam Małysz, one of the greatest ski jumpers ever. Two years later, before attending the ACM-ICPC World Finals with Tomek Czajka and Andrzej Gąsienica-Samek, I watched the Nordic World Ski Championships, at which Adam Małysz won gold medals in both individual ski jumping contests. Watching the grace and ease with which he won both contests, I started wondering whether we could perform just as well in the ICPC finals. And we did!

Overview

We treat allowed moves as two-dimensional vectors (v_x, v_y). Let S be their set. We say that S is *complete* if each pair of integers (x, y) can be expressed as a sum of a finite sequence $v_1, \ldots, v_k$ of vectors in S, where each vector in S can appear multiple times in the sequence. The goal of the problem is to determine whether S is complete.

Our algorithm consists of two stages. First, we determine whether the set of moves suffices to reach both the positive and negative half-axes of the Cartesian coordinate system. More precisely, we say that S is *axis-complete* if there exist positive integers a, b, c, and d such that vectors $(a, 0)$, $(-b, 0)$, $(0, c)$, and $(0, -d)$ can be obtained by adding vectors in S. Obviously, if S is not axis-complete, then it is not complete either.

Note that, in general, if the superknight can make a move $v = (v_x, v_y)$, it does not imply that there is a sequence of moves that results in moving the superknight by the vector $-v = (-v_x, -v_y)$. The simplest example is $S = \{(0, 1)\}$, where it is not possible to move the superknight by $(0, -1)$. Formally, we say that S is *reversible* if, for each vector $v \in S$, $-v$ can be expressed as a sum of a finite sequence of vectors from S. We show later that if S is axis-complete, then it is reversible as well.

For any set S of vectors, we write S^* to denote the set of all vectors that can be obtained by adding and subtracting vectors in S. More formally, for a finite $S = \{v_1, \ldots, v_n\}$, we have

$$S^* = \left\{ \sum_{i=1}^{n} a_i v_i : a_1, \ldots, a_n \in \mathbb{Z} \right\},$$

where $\mathbb{Z}$ is the set of all integers. Note that if S is reversible, then S^* is exactly the set of vectors that can be obtained by adding vectors in S, because every subtraction can be simulated by a sequence of moves reversing a specific move. Therefore, a reversible S is complete if and only if $S^* = \mathbb{Z}^2$.

In the second stage, our algorithm tests whether $S^* = \mathbb{Z}^2$. It follows from our reasoning that this is sufficient to test completeness of axis-complete sets S, i.e. those that have passed the first stage. We show that for any set S, it is possible to compute two vectors v_1 and v_2 such that $\{v_1, v_2\}^* = S^*$. In particular, we give a procedure that takes three vectors v_1, v_2, and w and outputs two vectors v_1' and v_2' such that $\{v_1, v_2, w\}^* = \{v_1', v_2'\}^*$. Moreover, the final vectors v_1 and v_2 are such that it is easy to check whether $\{v_1, v_2\}^* = \mathbb{Z}^2$.

This finishes the sketch of our algorithm. We will fill in all missing details in the next three sections.

```
Algorithm IsComplete(S)
    if S is not axis-complete then
        return false
    else if S* = Z^2 then
        return true
    else
        return false
```

Testing axis-completeness

In this section we give an algorithm that determines whether a set S of vectors is axis-complete. Without loss of generality, we only show how to determine whether it is possible to obtain a vector $(a, 0)$ as a sum of a sequence of vectors in S for some $a > 0$. The other three cases can be reduced to this by rotating the set of vectors.

We assign each vector of S to one of the following three sets, based on the sign of the vector's second coordinate:

$$S_0 = S \cap \mathbb{Z} \times \{0\},$$
$$S_+ = S \cap \mathbb{Z} \times \{y : y > 0\},$$
$$S_- = S \cap \mathbb{Z} \times \{y : y < 0\}.$$

In other words, $v = (v_x, v_y) \in S$ belongs to S_0, S_+ or S_- if v_y is zero, positive or negative, respectively.

Note that if S_0 contains a vector $(a,0)$ for some $a > 0$, we are trivially done. Otherwise, observe that if a vector $(a,0)$ for some $a > 0$ can be obtained as a sum of vectors in S, then we can also obtain a vector of this form as a sum of vectors only in $S_- \cup S_+$. In this case, if at least one of S_- and S_+ is empty, we are also trivially done, since by adding a nonzero number of vectors in S_- or S_+, we can only obtain a vector with a nonzero second coordinate. Summarizing, this means that for the rest of this section we can focus on showing an algorithm for determining if a vector of the form $(a,0)$ for some $a > 0$ can be obtained as a sum of vectors in $S_- \cup S_+$, where both S_- and S_+ are nonempty.

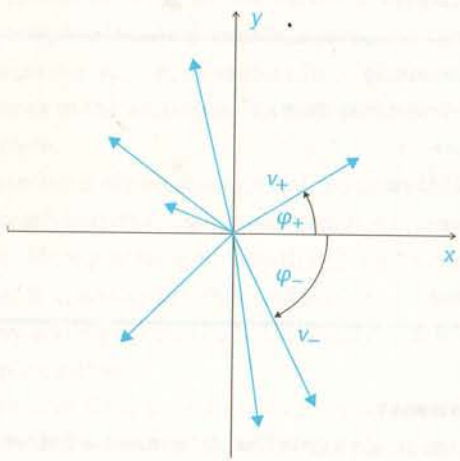


Figure 1: *Angles φ_+ and φ_-*

Let φ_- be the minimum angle that a vector in S_- makes with the positive half of the x-axis. Let $v_- = (x_-, y_-)$ be any of the vectors in S_- with this property. We analogously define φ_+ and $v_+ = (x_+, y_+)$ for S_+. See figure 1 for an example. We prove a lemma that relates the question that we want to answer to the sum of φ_- and φ_+.

Lemma 1. The following three conditions are equivalent:

- $\varphi_- + \varphi_+ < 180°$,
- there is a positive integer a such that the vector $(a,0)$ can be obtained as a sum of vectors in $S_- \cup S_+$,
- $-x_+ \cdot y_- + x_- \cdot y_+ > 0$.

Proof. We say that a vector v is a *nonnegative combination* of vectors $v_1, \ldots, v_k$ if $v = \sum_{i=1}^{k} a_i v_i$, for some nonnegative real numbers a_i. Let

$$w = (-y_-) \cdot v_+ + y_+ \cdot v_- = (-x_+ \cdot y_- + x_- \cdot y_+, 0).$$

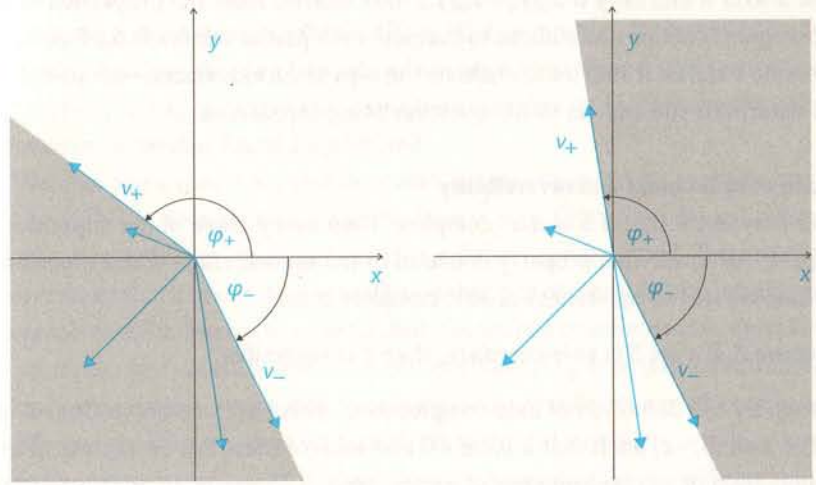


Figure 2: *The case that $\varphi_- + \varphi_+ \geq 180°$ (left) and $\varphi_- + \varphi_+ < 180°$ (right). The gray area depicts nonnegative combinations of vectors in S.*

Consider the case that $\varphi_- + \varphi_+ \geq 180°$ (see figure 2). In this case, no point on the positive half of the x-axis is a nonnegative combination of vectors in $S_- \cup S_+$. All nonnegative combinations lie within the cone limited by v_- and v_+, spanning clockwise from v_- to v_+. It is therefore not possible to reach the positive half of the x-axis by adding vectors in $S_- \cup S_+$. Also, since w is a nonnegative combination of v_- and v_+, it must be the case that $-x_+ \cdot y_- + x_- \cdot y_+ \leq 0$.

Consider now the case that $\varphi_- + \varphi_+ < 180°$. Note that w is a nonnegative combination of v_- and v_+ with positive integer coefficients, which means that w can be expressed as a sum of a finite sequence consisting of v_- and v_+. Since the angle between v_- and v_+ is not 180°, these vectors are not collinear, and $w \neq (0,0)$. Furthermore, all nonnegative combinations of v_- and v_+ lie within the cone limited by v_- and v_+, spanning clockwise from v_+ to v_-. This means that no point on the negative half of the x-axis is a nonnegative combination of v_- and v_+. This proves that $-x_+ \cdot y_- + x_- \cdot y_+$, the first coordinate of w, is positive, and finishes the proof. □

Both v_+ and v_- can easily be determined in linear time. Without loss of generality, since the cases are symmetric, let us focus on determining v_+. Consider two vectors $u = (x_u, y_u)$ and $w = (x_w, y_w)$ in S_+. The angle between u and the positive half of the x-axis is smaller than the angle between w and the positive half of the x-axis if and only if $x_u/y_u > x_w/y_w$. This follows from the properties of the cotangent function and suffices to compare each pair of vectors in S_+. Finally, as Lemma 1 states, it suffices to evaluate the sign of the expression $-x_+ \cdot y_- + x_- \cdot y_+$ to determine the answer to the question being considered.

Axis-completeness and reversibility

We now show that if S is axis-complete, then every move of the superknight can be reversed. This property is useful in the second stage of our algorithm, where we test completeness of axis-complete sets.

Lemma 2. If a set S is axis-complete, then it is reversible.

Proof. By the definition of axis-completeness, there are vectors $(a,0)$, $(-b,0)$, $(0,c)$, and $(0,-d)$ such that $a,b,c,d > 0$ and each of them can be expressed as a summation of a finite sequence of vectors from S. Consider an arbitrary vector $v = (v_x, v_y)$ in S. It is sufficient to show that $-v$ can be expressed as a summation of a finite number of vectors from S. If $v = (0,0)$, the claim is trivially true. We therefore assume for the rest of the proof that $v \neq (0,0)$.

Consider a vector $(w_x, 0)$, where w_x equals a if $v_x < 0$, and $-b$ otherwise. Similarly, consider $(0, w_y)$, where w_y equals c if $v_y < 0$, and $-d$ otherwise. Note that $(w_x, 0)$ and $(0, w_y)$ are two of the four vectors that we obtained from the definition of axis-completeness. Observe that

$$\begin{aligned} &|w_x \cdot w_y| \cdot v + |w_y \cdot v_x| \cdot (w_x, 0) + |w_x \cdot v_y| \cdot (0, w_y) = \\ &\quad = \big(|w_x \cdot w_y| \cdot v_x + |w_y \cdot v_x| \cdot w_x,\ |w_x \cdot w_y| \cdot v_y + |w_x \cdot v_y| \cdot w_y\big) = (0,0), \end{aligned}$$

where the second inequality follows from the fact that v_x and w_x do not have the same sign, and nor do v_y and w_y. Note that since both w_x and w_y are nonzero, $|w_x \cdot w_y| \geq 1$ and therefore

$$\big(|w_x \cdot w_y| - 1\big) \cdot v + |w_y \cdot v_x| \cdot (w_x, 0) + |w_x \cdot v_y| \cdot (0, w_y) = -v$$

can be obtained by adding a finite number of vectors v, $(w_x, 0)$, and $(0, w_y)$. □

Determining whether $S^* = \mathbb{Z}^2$

We write $\mathrm{GCD}(x,y)$ to denote the greatest common divisor of two integers x and y. If at least one of x and y is nonzero, then $\mathrm{GCD}(x,y)$ is the largest positive integer that divides both x and y. We assume that $\mathrm{GCD}(0,0) = 0$. In this section, we use Euclid's well-known algorithm for computing the greatest common divisor. The running time of the algorithm is $O(\log M)$ if $|x|$ and $|y|$ are bounded by M. Moreover, the algorithm can be modified to compute integers t_x and t_y such that $\mathrm{GCD}(x,y) = t_x \cdot x + t_y \cdot y$, where $|t_x|$ and $|t_y|$ are bounded by M. (This modification is known as *extended Euclid's algorithm*.)

We now show that it is possible to give at most two vectors v_1 and v_2 such that $\{v_1, v_2\}^* = S^*$. Our vectors are of the form $v_1 = (x_1, 0)$ and $v_2 = (x_2, y_2)$, where x_1 and y_2 are nonnegative. Moreover, if $y_2 = 0$ then $x_2 = 0$ as well. This requirement ensures that if the vectors are collinear then one of them is $(0,0)$. Intuitively, v_1 represents the minimum horizontal shift, the second coordinate of v_2 describes the minimum vertical shift, and the first coordinate of v_2 describes an additional horizontal deviation, which may be impossible to completely remove using v_1.

We now show how for two vectors of this form, and any vector $w = (x_3, y_3)$, we compute two vectors $v_1' = (x_1', 0)$ and $v_2' = (x_2', y_2')$ such that $\{v_1, v_2, w\}^* = \{v_1', v_2'\}^*$. Furthermore, v_1' and v_2' are of the same form as v_1 and v_2, and can be thought of as updated v_1 and v_2. Without loss of generality, we assume that $y_3 \geq 0$. If $y_3 < 0$ then w could be replaced by $-w$.

We set x_1 as follows:

$$x_1' = \begin{cases} \mathrm{GCD}(x_1, x_3) & \text{if } y_3 = 0, \\ x_1 & \text{if } y_2 = 0 \text{ and } y_3 \neq 0, \\ \mathrm{GCD}(x_1, \frac{y_3}{\mathrm{GCD}(y_2,y_3)} \cdot x_2 - \frac{y_2}{\mathrm{GCD}(y_2,y_3)} \cdot x_3) & \text{if } y_2 \cdot y_3 \neq 0. \end{cases}$$

Note that $v_1' \in \{v_1, v_2, w\}^*$. We prove the following claim which states that v_1' indeed represents the minimum horizontal shift.

Lemma 3. It holds that $\{v_1'\}^* = \{v_1, v_2, w\}^* \cap \mathbb{Z} \times \{0\}$.

Proof. If $y_3 = 0$ then any vector of the form $(a, 0) \in \{v_1, v_2, w\}^*$ can be expressed as $\alpha \cdot v_1 + \beta \cdot w$, where $\alpha, \beta \in \mathbb{Z}$, because v_2 is either $(0,0)$ or is the only vector with a nonzero second coordinate. Therefore, $\{v_1, v_2, w\}^* \cap \mathbb{Z} \times \{0\}$ is the set of vectors of the form $\alpha \cdot (\mathrm{GCD}(x_1, x_3), 0) = \alpha \cdot v_1'$, for $\alpha \in \mathbb{Z}$.

Consider now the case that $y_3 \neq 0$ and $y_2 = 0$. It must also be the case that $v_2 = (0,0)$. Therefore, all vectors of the form $(a, 0) \in \{v_1, v_2, w\}^*$ can be expressed as $\alpha \cdot v_1$, where $\alpha \in \mathbb{Z}$.

The last remaining case is $y_2 \cdot y_3 \neq 0$. Due to the earlier assumptions, it in fact holds that $y_2, y_3 > 0$. A vector $\alpha \cdot v_1 + \beta \cdot v_2 + \gamma \cdot v_3$ is of the form $(a, 0)$ if and only if $\beta \cdot y_2 + \gamma \cdot y_3 = 0$. For a given integer β, a corresponding integer γ exists if and only if y_3 divides $\beta \cdot y_2$. This is equivalent to $\beta \cdot y_2$ being a multiple of the least common multiple of y_2 and y_3. This means that the vectors of the form $(a, 0)$ in $\{v_1, v_2, w\}^*$ are exactly the set of vectors of the form

$$\alpha \cdot v_1 + \beta \cdot \left(\frac{y_3}{\mathrm{GCD}(y_2,y_3)} \cdot v_2 - \frac{y_2}{\mathrm{GCD}(y_2,y_3)} \cdot v_3\right) \quad \text{for } \alpha, \beta \in \mathbb{Z},$$

because the least common multiple of y_2 and y_3 equals $\frac{y_2 \cdot y_3}{\mathrm{GCD}(y_2,y_3)}$. Equivalently, these vectors can be expressed as

$$\alpha \cdot (x_1, 0) + \beta \cdot \left(\frac{y_3}{\mathrm{GCD}(y_2,y_3)} \cdot x_2 - \frac{y_2}{\mathrm{GCD}(y_2,y_3)} \cdot x_3, 0\right) \quad \text{for } \alpha, \beta \in \mathbb{Z},$$

which proves that they are of the form

$$\alpha \cdot \left(\mathrm{GCD}\left(x_1, \frac{y_3}{\mathrm{GCD}(y_2,y_3)} \cdot x_2 - \frac{y_2}{\mathrm{GCD}(y_2,y_3)} \cdot x_3\right), 0\right) \quad \text{for } \alpha \in \mathbb{Z}.$$

This finishes the proof of the lemma. □

We now set $v_2' = (x_2', y_2')$. Let t_2 and t_3 be integers computed by extended Euclid's algorithm such that $t_2 \cdot y_2 + t_3 \cdot y_3 = \mathrm{GCD}(y_2, y_3)$. Let $v_2'' = t_2 \cdot v_2 + t_3 \cdot w$. Let x_2'' and y_2'' be the first and second coordinate of v_2'', respectively. We set $y_2' = y_2''$ and

$$x_2' = \begin{cases} x_2'' & \text{if } x_1' = 0, \\ \min\{x_2'' + \alpha \cdot x_1' : \alpha \in \mathbb{Z} \text{ i } x_2'' + \alpha \cdot x_1' \geq 0\} & \text{otherwise.} \end{cases}$$

Note that computing the minimum in the above definition can be implemented in constant time, using the modulo operator. Note also that $v_2' = v_2'' + \alpha \cdot v_1'$ for some $\alpha \in \mathbb{Z}$, which implies that $v_2' \in \{v_1, v_2, w\}^*$. It is not hard to verify that v_1' and v_2' have the desired form.

Lemma 4. It holds that $\{v_1', v_2'\}^* = \{v_1, v_2, w\}^*$.

Proof. It holds that $\{v_1', v_2'\}^* \subseteq \{v_1, v_2, w\}^*$, because v_1' and v_2' were constructed in such a way that it is clear that v_1' and v_2' belong to $\{v_1, v_2, w\}^*$. It remains to prove the other inclusion. Consider any vector $u = (x_u, y_u)$ in $\{v_1, v_2, w\}^*$. Clearly, y_u has to be a multiple of the greatest common divisor of the second coordinates of v_1, v_2, and w. More precisely, it has to be a multiple of $y_2' = \mathrm{GCD}(y_2, y_3)$. Let $\alpha \in \mathbb{Z}$ be such that $y_u = \alpha \cdot y_2'$. For some integer q, it holds that $u = (q, 0) + \alpha \cdot v_2'$. Since both u and $\alpha \cdot v_2'$ belong to $\{v_1, v_2, w\}^*$, so does $(q, 0)$. It therefore follows from Lemma 3 that $(q, 0) \in \{v_1'\}^* \subseteq \{v_1', v_2'\}^*$. Finally, since both $(q, 0)$ and $\alpha \cdot v_2'$ belong to $\{v_1', v_2'\}^*$, so does u. □

Our algorithm for determining whether $S^* = \mathbb{Z}^2$ starts by initiating two variables v_1 and v_2 to $(0,0)$. Then, for each vector $w \in S$, it updates the vectors v_1 and v_2 as described above. We know that $\{v_1, v_2\}^*$ for the updated v_1 and v_2 equals $\{v_1, v_2, w\}^*$ for the old v_1 and v_2. The final vectors v_1 and v_2 are such that $\{v_1, v_2\}^* = S^*$, which trivially follows by induction. Due to how our procedure updates the vectors, $\{v_1, v_2\}^* = \mathbb{Z}^2$ if and only if $v_1 = (1,0)$ and $v_2 = (0,1)$.

Time complexity

The first part of the algorithm, determining whether the set is axis-complete, runs in $O(n)$ time. Let M be an upper bound on the absolute value of all coordinates in S. It can easily be proved by induction that in the second part of the algorithm, both $|x_1|$ and $|x_2|$ are bounded by M^2, and $|y_2|$ is bounded by M. Therefore, processing each vector from the input takes $O(\log M)$ time, since we run Euclid's algorithm at most a constant number of times and use only a constant number of additional arithmetic operations. Hence the total running time of the algorithm is $O(n \log M)$.

JAKUB PACHOCKI

Jakub Pachocki is a student at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. In high school he was a silver medalist at the International Olympiad in Informatics and a winner of the Baltic Olympiad in Informatics. As a university student he won both the Polish Collegiate Programming Contest and the ACM-ICPC Central European Regional Contest twice, and represented the University of Warsaw at the ACM-ICPC World Finals in 2011 and 2012. In 2011 he ranked third in the world finals of Google Code Jam and fourth at the Facebook Hacker Cup. In the same year Jakub Pachocki won two Polish programming competitions: Algorithmic Engagements and Wielka Przesmycka (University of Wroclaw Programming Contest). Coming up with challenging algorithmic tasks is one of his hobbies, along with reading interesting books (fantasy, fiction, science and popular science books), writing and swimming.

/ Ice *Skates*

Contest: 16th Polish Olympiad in Informatics
Task author: Jakub Radoszewski
Memory: 32 MB
http://main.edu.pl/en/archive/oi/16/lyz

Byteasar runs a skating club. Its members meet on a regular basis and train together, and they always use the club's ice skates. The skate sizes are (by convention) numbered from 1 to n. Naturally, each club member has a particular foot size, but there's more to it than that! Skaters have skate size tolerance factor d: a skater with foot size r can wear skates with sizes from r up to $r + d$. It should be noted, though, that *no skater ever wears two skates of different sizes simultaneously.*

To supply the club, Byteasar bought k pairs of ice skates of each size, i.e. from 1 to n. As time passes, new people join the club, just as some established members leave it. Byteasar worries whether he will have enough skates of appropriate sizes for every member at each training session.

We assume that initially the club has no members at all. Byteasar will give you a sequence of m events of the following form: x members with foot size r join or leave the club. Right after each such event, Byteasar wants to know whether he has enough skates of appropriate size for every club member. He asks you to write a program that will check this for him.

Input

The first line of the input contains four integers, n, m, k and d ($1 \le n \le 200{,}000$, $1 \le m \le 500{,}000$, $1 \le k \le 10^9$, $0 \le d < n$), separated by single spaces, that specify, respectively: maximum skate size, number of events, number of skate pairs of each size that Byteasar initially bought, and the skate size tolerance factor. The following m lines contain the sequence of m events, one per line. The $(i + 1)$-th line (for $1 \le i \le m$) holds two integers: r_i and x_i ($1 \le r_i \le n - d$, $-10^9 \le x_i \le 10^9$), separated by a single space. If $x_i \ge 0$, it means that x_i new members, each with foot size r_i, have just joined the club. And if $x_i < 0$, it means that x_i members, each with foot size r_i, have just left the club. You may assume the sequence is sensible; i.e. someone who never joined the club cannot leave it.

Output

Your program should write m lines to the output. The i-th line (for $1 \le i \le m$) should contain either the word TAK (Polish for *yes*) or the word NIE (Polish for *no*), depending on whether Byteasar has skates of appropriate size for every club member right after the i-th event.

Example

For the input data:

```
4 4 2 1
1 3
2 3
3 3
2 −1
```

the correct result is:

```
TAK
TAK
NIE
TAK
```

Explanation of the example: After all events from the input sequence have taken place, there are three club members who can wear skates of size 1 or 2, two members who can wear size 2 or 3, and three members who can wear size 3 or 4. With such a list of members, two pairs of ice skates in each of the sizes 1, 2, 3, and 4 suffice:

- two members get skates of size 1,
- the skates of size 2 are given to the member who can wear size 1 or 2, and another one who can wear size 2 or 3,
- the skates of size 3 are given to the member who can wear size 2 or 3, and another one who can wear size 3 or 4,
- the remaining two members receive skates of size 4.

Solution

I had the pleasure of battling with the task about ice skating during the second stage of the 16th Polish Olympiad in Informatics. The duel did not last overly long, since—thanks to positive currents—I managed to solve the task in under forty minutes from the start of the competition. Nevertheless, *Ice Skates* sank into my memory as an exceptionally elegant fusion of two apparently entirely independent algorithmic problems.

My first (and successful, at that!) approach to the task was very direct. Namely, let us consider when ice skates can be assigned to all members of the club. This problem has a natural graph interpretation: we may construct a bipartite graph, with the club members on its left side and the ice skates on its right side, and draw edges between every skater and the ice skates that he is allowed to use. For now, let us try *not* to think about the various special properties of the graph (originating from the simple rules that describe the tolerance of the skaters). In our task, we ask whether there exists a one-sided perfect matching in this bipartite graph. It is natural to suspect that this problem is well known. Indeed, the criterion that determines the existence of such a matching is known as Hall's theorem.

Hall's theorem. Assume a bipartite graph is given, with its edges connecting the vertices from a set U on the left side and a set V on the right side. There is a way to assign to each vertex from U a different, neighboring vertex from V if and only if for every $X \subseteq U$ the vertices from the set X are connected to at least $|X|$ vertices from V.

If we denote by $L(X)$ the set of vertices from V connected to any vertex from X, the above theorem dictates that for every X the inequality $|X| \ge |L(X)|$ must hold. In other words, we can assign ice skates to all members of Byteasar's club if and only if there exists no subset of skaters such that there are fewer ice skates that fit *any* of those skaters than the cardinality of the set. This gives us the first solution to the task: each time we want to answer one of the queries, we can go through all subsets of the skaters and check whether any of them fail to fulfill the condition from Hall's theorem. The complexity of such a solution is, however, far from satisfactory, since the time needed to answer a single query depends exponentially on n.

Do we actually have to go through all subsets of the skaters? Assume that at some point the condition from Hall's theorem is not fulfilled by the set $X \subseteq U$.

If X could be represented as the sum of two nonempty disjoint sets X_1, X_2 such that additionally $L(X_1) \cap L(X_2) = \emptyset$, then it is easy to verify that one of the sets X_1, X_2 would fail to fulfill the condition of Hall's theorem. Therefore, we only need to consider sets X that cannot be divided in this way (which we shall refer to as *indivisible* sets from now on). Suppose, in turn, that such an indivisible set X is strictly contained in some set $Y \subseteq U$, such that $L(X) = L(Y)$. Then Y is also indivisible and, naturally, $L(Y) = L(X) < |X| < |Y|$; therefore Y also fails to fulfill the condition of Hall's theorem.

The right time has now come to use the special properties of our graph. Let U be the set of club members and V be the set of ice skates. It follows from the above observations that if there exists a subset of the skaters that does not fulfill the condition of Hall's theorem, there also exist such numbers l, r ($1 \le l \le r \le n - d$) for which the set of *all* skaters of foot size in the interval $[l, r]$ is indivisible and also fails to fulfill this condition! Moreover, we may assume that there exist skaters with foot size l and r (otherwise we could simply pick a smaller interval).

Determining whether Hall's condition holds for such simply defined sets turns out to be much easier than in the general case. Let us observe that the total number of ice skates that fit skaters whose foot size lies within the interval $[l, r]$, picked as above, is equal to $k(r - l + 1 + d)$ (here we use the fact that the set of these skaters is indivisible). If we denote by a_i the number of skaters with foot size i, the total number of club members with foot size in the given interval will be equal to $a_l + a_{l+1} + \cdots + a_r$. Hall's condition is therefore not fulfilled if $(a_l - k) + (a_{l+1} - k) + \cdots + (a_r - k) > kd$. If we denote $b_i := a_i - k$, then there exists a subset of club members that does not fulfill Hall's condition if and only if the greatest possible sum of a contiguous subsequence of the sequence $b_1, \ldots, b_{n-d}$ exceeds kd. Therefore, if we can find a data structure to hold the sequence b, that will allow for fast modification of one element of the sequence and answer a query about the maximum sum of a contiguous subsequence, we will obtain a complete solution of the original task.

This structure turns out to be the segment tree—something of a classic in the Polish Olympiad in Informatics. The idea is relatively simple: to construct a tree that operates on a sequence of numbers, we create a root that aggregates information about the whole sequence, and in its children we create analogous trees corresponding to the left and right halves of our sequence. Note that such a structure contains only logarithmically many vertices that aggregate information concerning any single element of the sequence; therefore, if we can compute the information for a given node of the tree in constant time (assuming

the information for its children has already been computed), we will obtain a structure that updates itself after the modification of any element in logarithmic time.

Is the information about the maximum sum of a contiguous subsequence of the sequence corresponding to the given node something we can compute in constant time, based on the data for the children of that node? Unfortunately... not by itself. It turns out, however, that it is sufficient to keep—apart from the function $s(u)$ (the maximum sum of a contiguous subsequence for a node u of our tree)—three auxiliary functions:

- $pref(u)$, denoting the maximum sum of a prefix of the sequence corresponding to the node u,
- $suf(u)$, denoting the maximum sum of a suffix of the sequence corresponding to the node u,
- $sum(u)$, denoting the sum of all elements of the sequence corresponding to the node u.

Then we obtain a set that's "closed" with respect to the possibility of being quickly computed based on information kept in the children. Indeed, if t is a node of the tree, and u and v are its children, it holds that:

$$\begin{aligned} s(t) &= \max\big(s(u), s(v), suf(u) + pref(v)\big), \\ pref(t) &= \max\big(pref(u), sum(u) + pref(v)\big), \\ suf(t) &= \max\big(suf(v), suf(u) + sum(v)\big), \\ sum(t) &= sum(u) + sum(v). \end{aligned}$$

Our solution will therefore, on every change to the set of members of the club, update the computed values of the functions s, $pref$, suf, sum in $O(\log n)$ vertices of the tree and print TAK ("yes") if and only if the value of s in the root of the tree does not exceed kd. Interestingly, the described algorithm has a surprisingly concise implementation!

If the reader has found pleasure in searching for a way to quickly compute information regarding matchings in a specific graph, I also recommend reading the following tasks: *Balloons* from the final round of Algorithmic Engagements 2008, *Hotel* from the 18th Central European Olympiad in Informatics, and *Permutation* from Algorithmic Engagements 2009. The statements of all these problems are available on the MAIN website.

/ Termites 2

Contest: Algorithmic Engagements 2010
Task author: Jakub Pachocki
Memory: 128 MB
http://main.edu.pl/en/archive/pa/2010/ted

Do you remember the two friendly plank-eating termites that never grow tired of mental exercise? (See the task *Termites* on page 123.) After eating away most of the fences in Byteland, they have acquired an appetite for trees.

A *tree* is a connected, undirected graph with n vertices and $n-1$ edges. Our two termites quickly got bored with monotonously eating the vertices—to make their meal more interesting, they invented another game. They agreed on an ordering $(e_1, e_2, \ldots, e_{n-1})$ of the edges of the tree they are going to eat. The game will last for at most $n-1$ rounds, with exactly one termite making a move in each round. The players move alternately (the starting player plays in round 1, the second in round 2, the first again in round 3, and so on). In the k-th round the playing termite must choose an uneaten end of edge e_k and eat it. If both ends of e_k are eaten before the termite makes its move, the game ends with its loss. If the game does not end in $n-1$ rounds, it is tied.

We assume that the termites (who are, after all, experienced in this kind of game) never make an error, and that the termite with a winning strategy wants to win in the earliest possible round, while its opponent tries to delay its own defeat for as long as it can. Your task is to determine, for a given tree and the ordering of its edges set by the termites, the round in which the game will end.

Input

In the first line of the input there is an integer n ($2 \le n \le 500{,}000$), specifying the number of vertices in the tree. In the next $n-1$ lines are given the tree's edges in the order set by the termites. In the i-th of these lines there are two integers u_i and v_i ($1 \le u_i, v_i \le n$), specifying the indices of the ends of edge e_i.

Output

In the first and only line of the output exactly one integer should be written— the number of the round in which the game will end, or -1 if the game will end in a draw.

Example

For the input data:

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

the correct result is:

```
4
```

Explanation of the example: If, in the first round, the starting termite eats vertex 3, and in the third round it eats vertex 4, its opponent will not have any valid moves in the fourth round, regardless of its play in the second round.

/ Solution

Termites 2 is one of my favorite, and simultaneously one of the hardest, of my tasks. It is different from most of my other tasks in that it was *not* created in a sudden flash of inspiration, but rather as the result of a few hours of strenuously attempting to create an interesting problem.

Reading the brilliant task *Termites* by Tomasz Idziaszek awoke in me a desire to become a problem setter myself for the great Algorithmic Engagements contest. I entertained this idea for a few days until finally, on a sunny May morning, I got up at dawn and started searching for good topics.

My initial idea seemed really interesting at first. The problem was as follows: we have a board with m rows and n columns (many great algorithmic problems take place on a board!); in each of its cells a real number is inscribed. Our goal is to place rooks on this board in such a way that no two rooks attack each other, and the sum of values of the fields on which the rooks stand is maximized. My intuition suggested that this task must have a solution that would probably be interesting and nontrivial. How great was my disappointment when, a while later, I realized that the solution—which does indeed exist—is already commonly known, as the Hungarian algorithm.

What was there to do? I decided to stick with the idea of placing rooks on a board; however, to make the task a little more difficult and move away from the class of problems solved by classical algorithms, I decided to add the element of a game. What if we had two players placing rooks alternately, both of them trying to maximize the sum of values of the cells on which they placed rooks, with that total being decreased by the sum of values of the cells containing rooks placed by their opponent? This sounded really interesting. However, as often happens with interesting-sounding problems, I hit a wall of a different kind—namely, I was unable to solve the task as stated! I therefore tried to make it easier by getting rid of the optimization element—let us forget about the values inscribed in the cells; let some of them be blocked and the others free, and have the players simply place rooks in free cells alternately, until one of them cannot make a valid move (is unable to place a rook where no other rook will attack it). Unfortunately, despite the inviting, natural appearance of the problem, I was still unable to take even a single step towards solving it.

All the indications were that I had made the task too hard when introducing the game element. I therefore tried to introduce a further simplification

which, I believe, was the most important of my ideas during my work on this task; the path from the stated task to *Termites 2* was actually quite straightforward. The idea is as follows: let the set of moves that the players can make depend on the current round. In the i-th round, let the player be able to place a rook only in the i-th row. This leads to a number of interesting consequences:

- The first player will only place rooks in odd rows and the second player only in even rows.
- It is no longer necessary to worry about the rooks being able to attack to the left and to the right (since we already know that no two will be placed in the same row anyway).
- For the state of the game at any given moment, the exact placement of rooks on the board does not matter; the only important concerns are the round number and the set of columns that are already occupied. This leads in a simple way to a solution based on dynamic programming, with an $O(2^n mn)$ running time. It is not particularly interesting, but it is a visible sign of progress: finally, we have obtained a task that we can solve with a nontrivial algorithm!

Unfortunately there was still no polynomial algorithm to solve the problem in sight. I therefore focused on the aforementioned solution that works in time exponentially dependent on the number of columns. I wondered what would happen if we bounded the number of free cells in each row—let us say that their count would not exceed r. I devoted a lot of time to this idea, but I was unable to find a solution. In spite of this, I decided to follow the punch and bound the value of r.

Values of r such as 4 or 5 did not bode well. Things seemed to get interesting at $r = 3$. If there were at most three free cells in each row, a player would have seemingly very few options in each move—if more than one of the cells in which he could place a rook was taken, he would have at most one possible move. Despite long deliberation, I was still unable to reach an interesting solution.

I was therefore left with the last value of $r = 2$ (for $r = 1$ the players never have any choice; the game would therefore never be any more interesting than the War card game!). In the presented problem, the difference between $r = 3$ and $r = 2$ is diametrical, quite like the difference between the problems 3-SAT and 2-SAT. Let us assume that in every row there are precisely two free cells—rows in which there are fewer free cells do not seem to bring anything interesting to the game. Then we can model our board as an undirected graph

in which the columns are vertices and the rows are edges between them. For example, the board:

```
.**..
**...
...**
..**.
```

corresponds to a graph with five vertices and four edges, of which the first edge connects vertices 2 and 3, the second 1 and 2, the third 4 and 5, and the fourth 3 and 4. The players alternately take the next edge in turn and cross out one of its undeleted ends; the first player to be unable to make a valid move loses.

As can be seen, setting $r = 2$ has nearly brought me to the final formulation of the task—at this point, the only important difference was that, in the graphs on which the game was played, I admitted the presence of cycles. Interested in the graph model of the board and the lack of an obvious solution, I set out to search for a polynomial algorithm to solve the problem. I started with a rather obvious observation—if, after adding the first k edges, in the graph there appears a connected component in which there are more edges than vertices, the game has to end with one player winning during the first k moves (this follows from the fact that we had to cross out at least one vertex for every edge of the selected component). We may therefore only consider graphs in which, in every connected component, there are at most as many edges as vertices.

Let us look at the possible states of the game after k moves. The edges considered thus far determine some number of connected components in the graph. It is apparent that the moves made on separate connected components were independent; let us therefore think about the possible state of a single connected component. If it contains n vertices, it must contain at least $n - 1$ edges (otherwise it would not be connected). Since for every edge some vertex had to be crossed out, in the connected component there is at most one undeleted vertex remaining—there are therefore at most n possible states!

Let us look at the possible endings to the game. We can separate them into two kinds:

1. A player makes a move on an edge connecting vertices u and v belonging to two previously separate connected components; the opponent has, however, previously forced the deletion of both the vertex u in the first of those components and the vertex v in the second.
2. A player makes a move on an edge connecting vertices u and v belonging to the same connected component (which must be a tree in the graph formed by previously considered edges). The opponent has, however, previously forced the deletion of both vertices u and v in that component.

To determine whether a player X can avoid losing in the given round, we therefore have to be able to effectively answer the corresponding questions:

1. Can player X ensure that the vertex u is not crossed out? Can player X ensure that the vertex v is not crossed out? (Let us note that these questions are completely independent of each other if u and v have been in different connected components so far!)
2. Can player X ensure that at least one of the vertices u, v is not crossed out (without stating which one)?

For each connected component we may therefore keep all the information of the form "player X can ensure that the only undeleted vertex is the vertex v" and "player X can ensure that the only undeleted vertex is one of the vertices u, v; however it may be the opponent who chooses one of them specifically". Following this lead, I actually obtained a polynomial solution to the problem. I did, however, ask myself the question: does the presence of cycles in the graph, introducing the clearly more conceptually complex second case in the above considerations, really add enough value to the task? I decided that it does not— it seemed to me that this is another example of a task in which the solution for a graph with a single cycle (a connected graph with an equal number of edges and vertices) is just a more complicated version of the solution for a tree. Thus I reached the final version of the problem—adding the story about termites was now natural, considering my initial inspiration for creating the task!

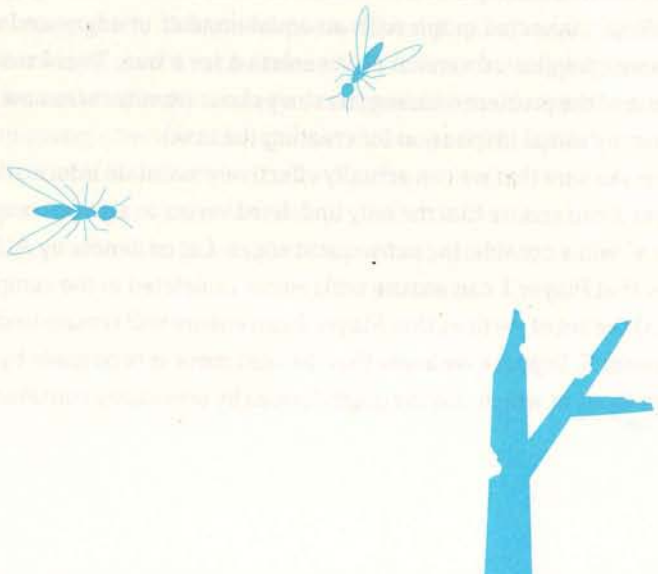
Let us make sure that we can actually effectively maintain information of the kind "player X can ensure that the only undeleted vertex in a given component is the vertex v" while considering subsequent edges. Let us denote by $A(S)$ the set of vertices that Player 1 can ensure will remain undeleted in the component S, and by $B(S)$ the set of vertices that Player 2 can ensure will remain undeleted in the component S. Suppose we know that the next move is to be made by Player 1 on the edge (u, v), at which u in the graph formed by previously considered edges

belongs to the component T_1, while v belongs to T_2 ($T_1 \neq T_2$). Let us denote by T the component formed after adding the edge (u, v) (so $T = T_1 \cup T_2$). After a moment of thought it is not hard to come to the conclusion that:

$$\begin{cases} A(T) = A(T_1) \cup A(T_2),\ B(T) = \varnothing, & \text{if } u \in A(T_1) \text{ and } v \in A(T_2), \\ A(T) = A(T_2),\ B(T) = B(T_2) \setminus \{v\}, & \text{if } u \in A(T_1) \text{ and } v \notin A(T_2), \\ A(T) = A(T_1),\ B(T) = B(T_1) \setminus \{u\}, & \text{if } u \notin A(T_1) \text{ and } v \in A(T_2). \end{cases}$$

Of course, if $u \notin A(T_1)$ and $v \notin A(T_2)$, then Player 2 can force the defeat of Player 1 in the current move.

A simple implementation of the above observations, storing the sets A and B in balanced binary search trees, allows us to obtain a solution that works in $O(n \log^2 n)$ time (where n is the number of vertices of the tree). After a more careful look at the nature of these sets, it is possible to obtain solutions that work in complexities $O(n \log n)$, $O(n \log^* n)$ and, finally, a very simple and elegant solution in $O(n)$ time—the finding of which I strongly encourage the reader to attempt!



PAWEŁ PARYS

Paweł Parys is a research assistant at the Institute of Informatics, University of Warsaw, where he has recently earned his Ph.D. degree. His miscellaneous scientific interests are connected with automata theory, including mainly algorithms for searching within XML documents and higher-order pushdown automata.

He started taking part in computer science competitions in high school. He is a two-time winner of the Polish Olympiad in Informatics and a runner-up of the International Olympiad in Informatics. During his university years, Paweł Parys and his team enjoyed three victories at the Polish Collegiate Programming Contest and two victories at the ACM-ICPC Central European Regional Contest. At the ACM-ICPC World Finals his team ranked tenth in 2004 and seventh in 2006. As an individual contestant, he was a multiple finalist of Google Code Jam, TopCoder Open and TopCoder Collegiate Challenge. He has also participated successfully in mathematical competitions.

Paweł Parys enjoys traveling, especially when it involves biking and mountain hiking.

/ Cave

Contest: 11th Polish Olympiad in Informatics
Task author: Paweł Parys
Memory: 16 MB
http://main.edu.pl/en/archive/oi/11/jas

There is a cave in Byteland. It consists of n chambers and corridors connecting them. The corridors are arranged in such a way that there is a unique path between each pair of chambers. In one of these chambers Hansel has hidden a treasure, but he won't say which chamber it is. Gretel wants to know where the treasure is, so she asks Hansel about various chambers. When she guesses right, Hansel tells her she is right, and when she guesses wrong, he tells her which way from this chamber leads to the treasure.

Task

Write a program that:

- reads from the input the description of the cave,
- finds the minimum number of questions Gretel has to ask, in the worst case, in order to learn in which chamber the treasure is hidden,
- writes the result to the output.

Input

In the first line of the input there is one positive integer n ($1 \le n \le 50{,}000$), specifying the number of chambers in the cave. The chambers are numbered from 1 to n. In the following $n - 1$ lines the corridors linking the chambers are described, one per line. Each of these lines contains a pair of distinct positive integers, a and b ($1 \le a, b \le n$), separated by a single space, indicating that there is a corridor between chambers a and b.

Output

Your program should write one integer to the output, stating the minimum number of questions Gretel has to ask in the worst case (i.e. we assume Gretel asks the questions in the best possible way, but the treasure is hidden in the chamber that requires the largest number of questions).

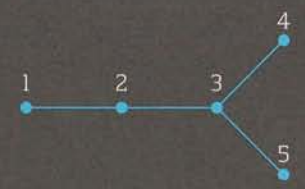
Example

For the input data:

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

the correct result is:

```
2
```



/ Solution

This problem turned out to be one of the most difficult in the history of the Polish Olympiad in Informatics. The best student's solution received just 52 points (out of 100), and only a few contestants achieved more than 10 points. This was a great surprise to me (as the author of the problem). My feeling was that it would be a relatively standard problem; the line of reasoning described in the following text appeared to me completely natural—in fact, the only approach possible—and every one of its steps not too difficult. But the contestants, in the limited time available during the competition, proved unable to deal with this problem.

After the contest, I wrote a paper together with Krzysztof Onak based on this problem. The problem is very natural: a generalization of binary search into trees. We have added another variant, where the questions are asked in terms of edges instead of nodes, and a few other small extensions. Our paper was accepted by an American conference: IEEE Symposium on Foundations of Computer Science (FOCS).

Strategy functions

So, how should this problem be solved? Of course, the plan of the cave is a tree. But how can we describe a strategy for asking questions? We have to determine only the minimum number of questions to ask, but without considering a strategy for asking questions, we will probably not manage to do that. So let us think about how we might ask questions in our problem. When asking about some node of the cave, we split the cave into some number of parts (connected components), which arise when this node is removed. As our answer we get one of these parts, and in it we have to find the treasure. It makes no sense to ask any of the subsequent questions about anything outside this part, because the answer to such a question would not give us any new information; afterwards we will be asking questions only about nodes inside this part. Nevertheless, we have to consider all of these parts, because any answer is possible. So we can think about this as follows. In the tree, we remove one selected node. In the next step we again choose one node, in each of the resulting parts (connected components), and we remove these nodes. We repeat this until the parts contain only one node each. In the problem, our goal is to minimize the number of such steps.

Based on this observation, we will be describing a strategy using a function f which assigns a nonnegative integer to each tree node. This number will tell us in which step, counting from the end, the node will be removed. Such a function has to satisfy the following condition:

for every two nodes u, v such that $f(u) = f(v)$,
on the path between u and v there is a node w such that $f(w) > f(u)$. (*)

This condition corresponds to the requirement that if we remove two nodes in the same step, some of the nodes between them would have had to be removed earlier. Such a function will be called a *strategy function*. We see that every reasonable strategy of asking questions (i.e. such that we do not ask questions that would not provide us with any new information) can be described using a strategy function, where the number of questions asked (for the worst sequence of answers) is equal to the maximum value of the function. For example, in figure 1 we present an optimal strategy function for the tree from the problem statement, and the corresponding decision tree. But of course this correspondence also goes in the other direction: having a strategy function f, it is easy to give a strategy that requires no more than $\max_v f(v)$ questions. We just ask about the node v in which $f(v)$ is maximal. As an answer, we get some part of the tree. Again we ask about the node v in which $f(v)$ is maximal in this part of the tree, and so on. In the current part of the tree, we always have exactly one node with maximal $f(v)$; this follows from the condition (*) above. Either we will hit the node with the treasure at some point or, at the end, we will get a tree consisting of one node and we will know that the treasure is in that node. To conclude, it is sufficient to find a strategy function f for which $\max_v f(v)$ is as small as possible.

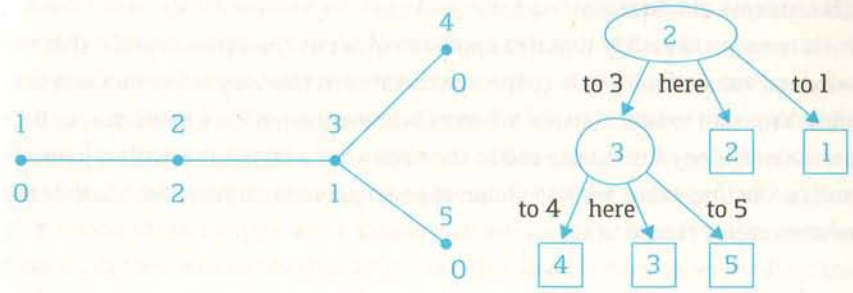


Figure 1: *Left: an optimal strategy function for the tree from the problem statement. Node numbers appear above the nodes; below them appear the values of the strategy function. Right: the corresponding decision tree. Inside the nodes appear Gretel's questions; on the edges, Hansel's answers.*

Idea of the solution

How should we find the best strategy function? A naïve approach is to begin by placing the greatest value of the function (the first question), and then consecutively smaller and smaller values. We see that the greatest value has to be assigned approximately in the middle of the cave. But it is not clear in which sense it should be the middle. We see also that it is better to choose a node that has more neighbors, since then we will divide the tree into a greater number of smaller parts. We can start in a node from which the distance to all other nodes is minimal; or in such a way that the emerging parts have the smallest number of nodes. But it is easy to ascertain that any such easy approach will not succeed. The problem is that our "size" of a tree should not be equal to the number of nodes or a diameter, but to the required number of questions; the first question should be asked in such a way that the "sizes" of the resulting parts, defined in this way, are minimal. We could start from smaller parts of the tree and, based on their results, analyze the larger ones; but the total number of parts of the tree might be very big, so we cannot process all of them in a reasonable time.

Another possible approach is to process the tree starting from the leaves and going into its center. In order to simplify the argumentation, let us choose any node of the tree as the root. This is done simply to fix the order in which we will traverse the nodes of the tree while calculating the solution: we will be considering the tree starting from the leaves and going towards the root; of course the choice of the root will not have any impact on the result. It turns out that the optimal strategy function can be obtained through a greedy approach: to every node, going from leaves to root, we assign the smallest possible strategy function value that does not conflict with the values already assigned in the descendants of that node.

It remains to justify that this approach gives us the correct result: that the maximal value of the strategy function created in this way is the smallest possible. We want to say: if, in the subtrees below a given node v, there are the best possible strategy functions, and to the node v we assign the smallest possible non-conflicting value, we will obtain the best possible strategy function for the whole subtree rooted in v.

Visibility sequences

It is easy to see that, in such an approach, a best possible strategy function is not only one whose maximal value is the smallest possible. A situation where the maximal value is at the root of a subtree is much better than a situation where this value is somewhere deeper. In the first case, above the subtree we can assign arbitrary values different from this maximal one, which is not true in the second case. We ascertain that all values "visible" from the root of a subtree are important. The value $f(v)$ of a given node v is called *visible* in a given subtree if all nodes w on the path from v to the root (excluding v, but including the root) satisfy $f(w) < f(v)$. In other words, any greater value does not cover the value $f(v)$ when we are looking from the subtree's root. The *visibility sequence* of a given subtree is the sequence of all values visible in it, in descending order. Notice that every value visible in a subtree can be visible in only one node (otherwise the condition (∗) would not be satisfied for the two nodes in which this value is visible). We see also that the arrangement of the other values is unimportant: if the visibility sequence remains the same, the values in the subtree can be rearranged arbitrarily, and it will not cause any trouble from the point of view of the rest of the tree.

We want to prove that, when we greedily assign the smallest possible values of the strategy function in order from leaves to the root, in each subtree we will obtain the best possible visibility sequence. So which visibility sequence is the best possible (i.e. allowing the greatest capabilities in the rest of the tree)? First of all, the first (largest) value of the sequence counts, as this is the maximum of the strategy function in the subtree. But, as we have already said, the remaining values of the sequence also count. It turns out that visibility sequences have to be compared lexicographically: a lexicographically smaller sequence is a better one. Indeed, let us consider a tree T with v as its root, and its subtree T_w with w as its root (where w is a child of v). Assume that for some strategy function f the visibility sequence in T_w is F_w, and in T it is F_v. Say that in the subtree T_w someone managed to define a strategy function g giving a visibility sequence G_w, lexicographically smaller than F_w. How can we extend g to the whole tree, so that we will obtain a function whose visibility sequence is lexicographically smaller that F_v or equal to F_v? It is very easy. Let a be the largest number that appears in F_w but does not appear in G_w (all greater numbers appear in none of them, or in both). If $f(v) > a$, on the rest of the tree we can leave the values of f. All values of g visible in the subtree were visible also when there were f (values greater than a), or they will be covered by $f(v)$ (values smaller than a), so the function defined in this way will be a strategy function. Its visibility sequence in T is F_v, as for f. On the other hand, if $f(v) < a$, we assign value a to the root, and on the rest of the tree we leave the values of f. It is easy to check that the function defined in this way is a strategy function. Its visibility sequence in T begins in

the same way as F_v, but numbers smaller than a stop being visible. So in this case we also obtain a visibility sequence which is lexicographically smaller than F_v, or equal to F_v, whereas the case $f(v) = a$ is impossible, as condition $(*)$ would not be satisfied for v and the node of T_w in which a is visible.

As a consequence of the above we see that, if we want to obtain the lexicographically smallest visibility sequence for the whole tree, it is enough to choose for each subtree (among the subtrees rooted in the children of the root) a strategy function that gives the lexicographically smallest visibility sequence for this subtree, and then to choose some value in the root. It is not difficult to see that the smaller value will be assigned to the root, the lexicographically smaller will be the visibility sequence for the whole tree. Of course, the same reasoning can be applied by induction for all smaller subtrees. This proves that our greedy approach is correct.

Algorithm

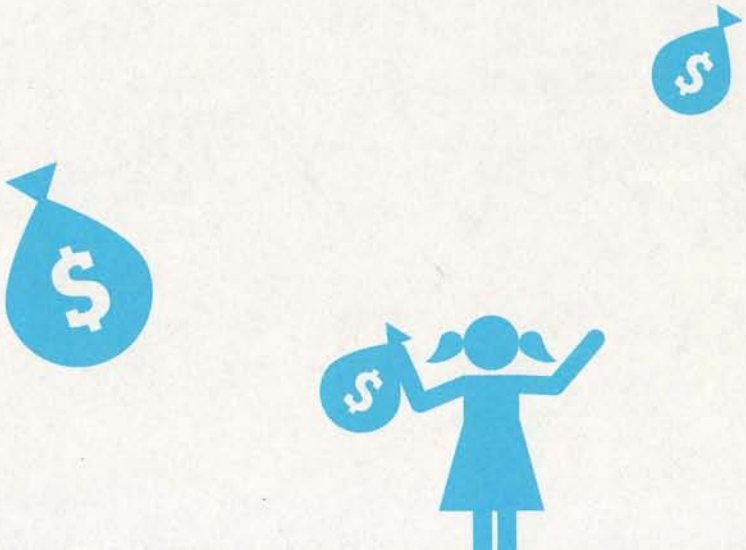
We have not yet said how to write a procedure that assigns to a node v the smallest possible value of strategy function, which does not conflict with the values already assigned in its descendants. Here again the visibility sequences are useful. As we have already said, for the whole tree rooted in a child of v, it is enough to know its visibility sequence. Let $F_1, F_2, \ldots, F_k$ be the visibility sequences for the subtrees rooted in the children of v. Condition $(*)$ for pairs containing nodes from one subtree is satisfied by assumption. We have to ensure that condition $(*)$ is satisfied for pairs of nodes from different subtrees. If some number a appears in two sequences F_i, F_j, we have to put in v something greater than a. Additionally, we have to ensure condition $(*)$ for pairs consisting of v and a node from one of the subtrees. Thus we cannot place in v a value which is already present in some sequence F_i. In other words: we look for the maximal number a appearing in two sequences F_i, F_j (assuming $a = -1$ if such a number does not exist); to node v we assign the smallest number greater than a which does not appear in any F_i. Moreover, in order to perform the same operation in the parent of v, we want to compute the visibility sequence for the whole subtree rooted in v: it contains $f(v)$ and those values from each F_i which are greater than $f(v)$.

The whole algorithm can be written using the pseudocode below.

```
Function Compute(v : node)
    {let w_1, w_2, ..., w_k—children of v}
    for i := 1 to k do
        F_i := Compute(w_i)
    a := (the greatest number appearing in two F_i
          or −1 if such a number does not exist)
    b := (the smallest number greater than a not appearing in any F_i)
    return {b} ∪ {c ∈ F_i : c > b, 1 ≤ i ≤ k}

Algorithm HowManyQuestions()
    choose arbitrary node v as the root
    output the greatest element of Compute(v)
```

Notice that in every tree of size n it is sufficient to ask $\lfloor \log n \rfloor$ questions. It is true because in every tree of size n there exists a node dividing it into parts from which each has at most $\frac{n}{2}$ nodes; in such a node we ask the first question, and we continue the same way. Thus the visibility sequences processed by the algorithm will have length at most $\lfloor \log n \rfloor$. It is easy to implement the above algorithm so that it works in time $O(n \log n)$. As a curiosity, let us add that the solution can also be implemented in linear time, representing the visibility sequences as integers.



/ Shuffle

Contest: 10th Polish Olympiad in Informatics
Task author: Paweł Parys
Memory: 32 MB
http://main.edu.pl/en/archive/oi/10/tas

Byteasar has a pack of n cards that he likes to shuffle. The positions of the cards are numbered from 1 to n. Byteasar has acquired such skill in shuffling that he obtains the same arrangement every time—i.e. a card from the k-th position ($1 \le k \le n$) always goes to the same a_k-th position. We denote by b_k the position of the k-th card after Byteasar has repeated the shuffling l times.

Task

Write a program that:

- reads from the input the numbers n and l and the sequence of numbers (b_k),
- determines the sequence of numbers (a_k),
- writes that sequence to the output.

Input

In the first line of the input there are two positive integers, n and l ($1 \le n, l \le 1{,}000{,}000$). In the consecutive n lines there are successive elements of the sequence (b_k), one per line. In the $(k+1)$-th line there is a positive integer b_k ($1 \le b_k \le n$) specifying the final position of the card from the k-th position.

Output

Your program should write to the output n integers: successive elements of the sequence (a_k), one per line. In the k-th line there should be one number a_k: the position of the card from the k-th position after a single shuffle. You may assume that, for the test data, the desired sequence (a_k) always exists. If there are multiple such sequences, your program should write any of them.

Example

For the input data:	the correct result is:	or:
5 2	1	2
1	2	1
2	4	4
5	5	5
3	3	3
4		

/ Solution

At first glance, the problem appears to be easy: some transformation of permutations is involved, and that's all. However, the truth is that one has to analyze the situation carefully. During the contest, only two people (in fact, the winners in the first two places) succeeded in doing that; all the other contestants received no more than 30 points out of 100. Thus, the problem turned out to be quite difficult.

How to solve this problem? Before we come to that, let us introduce some concepts that will be helpful in understanding the solution. Instead of shuffling, we will be talking about permutations. A permutation is a function from set $\{1,2,\ldots,n\}$ to itself, in which every value is taken exactly once. A permutation can be composed which corresponds to repeating a shuffle. Instead of composing l times, we can refer to taking the l-th power. In the problem, we are looking for a permutation whose l-th power gives the permutation from the input. Notice that taking powers of a permutation is very similar to taking powers of a number; in particular, f^{a+b} is equal to f^a composed with f^b, and f^{ab} is equal to $(f^a)^b$.

Powers of a permutation

A cycle in a permutation is a sequence of elements $a_1, a_2, \ldots, a_k$ such that

$$f(a_1) = a_2,\ f(a_2) = a_3,\ \ldots,\ f(a_{k-1}) = f(a_k),\ f(a_k) = a_1.$$

As one can easily deduce, in order to solve this problem it is essential to consider cycles of our permutations. One can see that a cycle of the initial permutation in its l-th power is transformed either to a single cycle or to a few cycles, all of the same length. More precisely, a cycle of length k in the l-th power is transformed to $\mathrm{GCD}(k,l)$ cycles of length $\frac{k}{\mathrm{GCD}(k,l)}$ (where GCD denotes the greatest common divisor).

In order to see what can happen, let us consider an example. Fix $l = 6$. A cycle of length 4 disintegrates in the l-th power into two cycles of length 2. Also, a cycle of length 12 disintegrates into cycles of length 2; this time we have six of them. But a cycle of length 2 disintegrates into two cycles of length 1. Thus, first: it can happen that a cycle of length k' can be obtained in the l-th power (for fixed l) from cycles of greater length, and not from cycles of length k'. Second: the same final effect can be obtained as a power of different permutations.

We want to reverse the power operation. Thus, we want to deduce, given k' and l, the possible values of k such that cycles of length k can give cycles of

length k' in the l-th power. In other words: for which k the equation $k' = \frac{k}{\mathrm{GCD}(k,l)}$ is satisfied. Let us decompose k' and l into prime factors:

$$k' = p_1^{\alpha_1} \cdots p_a^{\alpha_a} r_1^{\gamma_1} \cdots r_c^{\gamma_c}, \quad l = q_1^{\beta_1} \cdots q_b^{\beta_b} r_1^{\delta_1} \cdots r_c^{\delta_c},$$

where all $\alpha_i, \beta_i, \gamma_i, \delta_i$ are positive, and all p_i, q_i, r_i are distinct. No prime number p_i can appear in $\mathrm{GCD}(k,l)$; thus it has to appear in k in the same power as in k'. On the other hand, each q_i and r_i appears in $\mathrm{GCD}(k,l)$ provided that it appears in k; its exponent in $\mathrm{GCD}(k,l)$ will be the smaller of its exponents in k and l. Thus, q_i can appear in k any number of times between 0 and β_i, and r_i has to appear exactly $\gamma_i + \delta_i$ times. To conclude,

$$k = p_1^{\alpha_1} \cdots p_a^{\alpha_a} q_1^{\varepsilon_1} \cdots q_b^{\varepsilon_b} r_1^{\gamma_1+\delta_1} \cdots r_c^{\gamma_c+\delta_c}, \qquad \text{where } 0 \le \varepsilon_i \le \beta_i \text{ for } 1 \le i \le b.$$

Notice that all such numbers k are multiplicities of

$$k_{\min}(k') = p_1^{\alpha_1} \cdots p_a^{\alpha_a} r_1^{\gamma_1+\delta_1} \cdots r_c^{\gamma_c+\delta_c}.$$

Consequently, if there exists a permutation whose l-th power gives the permutation on input, there exists also such permutation that all cycles of length k' in the input permutation arise from cycles of length $k_{\min}(k')$ in this permutation (instead of one cycle of some larger length k we can take multiple cycles of length $k_{\min}(k')$).

Algorithm

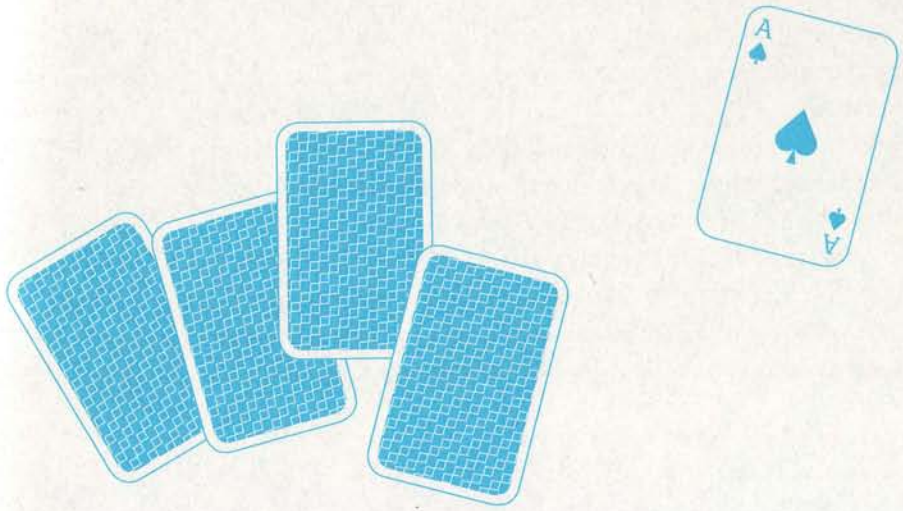
After all these observations, we come to the question of how to find the original permutation. Concentrate first on a special case where the length k' of a cycle (in the permutation on input) is coprime with l. Then $k_{\min}(k') = k$, so every cycle of this length originates from one cycle in the original permutation. From the basic facts of number theory we know that in such a situation there exists a natural number p such that $p \cdot l \equiv 1 \pmod{k'}$ (such a number is called the *inverse*). It is easy to see that for $a \equiv b \pmod{k'}$ the a-th and the b-th power of a cycle of length k' are equal. Thus, taking the p-th power of our cycle, we obtain what we want: a cycle whose l-th power gives the input cycle.

Next consider the general case. Take $\frac{k_{\min}(k')}{k'}$ cycles of length k'. We want to find a cycle of length $k_{\min}(k')$ whose l-th power disintegrates into these cycles. We see that the $\frac{k_{\min}(k')}{k'}$-th power already gives us cycles of length k' (obtained in such a way that we read every $\frac{k_{\min}(k')}{k'}$-th element of the original cycle). Then for $m = \frac{l \cdot k'}{k_{\min}(k')}$ the m-th power of these cycles is taken. But we know that k' and m are coprime (this can be easily seen from the decomposition into prime numbers). Thus we first find cycles whose m-th power gives our cycles of length k' from the input. Then we "interlace" cycles obtained in this way. This means that we first take one element from each cycle, then we take the second element from each cycle, and so on.

To conclude, we proceed as follows:

1. We find all cycles of the permutation, and calculate how many cycles of each length we have.
2. Until we have processed all cycles, we do the following:
   (a) We take any group of $\frac{k_{\min}(k')}{k'}$ not yet processed cycles of some length k'.
   (b) We compute p such that $m \cdot p \equiv 1 \pmod{k'}$, where $m = \frac{l \cdot k'}{k_{\min}(k')}$.
   (c) We take the p-th power of each of these cycles.
   (d) We interlace the resulting cycles.

Some details remain. The computation of the inverse of m modulo k' can be effectively implemented using the Euclidean algorithm (and in other situations it should be done this way). But here it is completely sufficient to check all numbers between 1 and $k'-1$; the complexity of the whole program remains the same. And how to compute $k_{\min}(k')$? We will be repeatedly dividing l by $\mathrm{GCD}(k',l)$ (in a loop) as long as $\mathrm{GCD}(k',l) > 1$. When we get $\mathrm{GCD}(k',l) = 1$, in l we will have $q_1^{\beta_1} \cdots q_b^{\beta_b}$ (for the decomposition of l and k' as above); having this, it is then easy to deduce the value of $k_{\min}(k')$. Operations such as computing the length of a cycle, taking a power of a cycle, and "interlacing" cycles can easily be implemented in such a way that the whole program will work in time linear in n.



JAKUB PAWLEWICZ

Jakub Pawlewicz works at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, where in 2009 he earned his Ph.D. in algorithmic methods for solving games. He is the author of an optimal program for playing Yahtzee. His scientific research is focused on methods of artificial intelligence in games (his main area of interest) and algorithmic number theory.

From 1994 to 1996 Jakub Pawlewicz won three medals at the International Olympiad in Informatics: bronze, silver and gold. In 1998, together with his team, he was ranked ninth at the ACM-ICPC World Finals in Atlanta. This was the first time in history that a team representing the University of Warsaw made it to the top ten of the best teams in the world.

/ Game of Tokens

Contest: Polish Olympiad in Informatics Training Camp 2010
Task author: Jakub Pawlewicz
Memory: 32 MB
http://main.edu.pl/en/archive/ontak/2010/zet

The *Game of Tokens* is a two-player game that is played with n tokens, labeled with the integers from 0 to $n-1$, on an $n \times n$ square grid filled with numbers, one per cell. The tokens and the grid are visible to both players.

The players take turns picking up tokens from the table, one per move, until all of the tokens are removed. Then the score for each player is computed as the sum of numbers in the cells captured by the player, where the cell in the i-th row and j-th column is captured if and only if the player picked up the tokens with labels i and j (or, if $i = j$, the single token i). The objective of the game is to maximize one's score advantage over the other player (and, if that is not possible, to minimize one's disadvantage). Your task is to design and implement a program that will play *Game of Tokens*.

Programming interface

Your solution should implement the following functions.

- void startgame(int *n*)—this function will be called exactly once, before either player's first move. The argument n indicates the number of tokens and the size of the grid ($1 \le n \le 40$).
- void opponentmove(int *token*)—this function is called when the opponent picks up a token, and should update your idea of the state of the game; the argument token is the label of the token ($0 \le token \le n-1$) that the opponent chose.
- int play()—this function should return your move, i.e. the label of the token that your implementation decides to take in your current move.

The functions play and opponentmove will be called in turns, with the first call indicating who makes the first move. In your implementation you may assume that the opponent's moves are correct and, in particular, that no token is picked up more than once.

The function valueat will be provided; it allows the state of any grid cell to be discovered, and may be called at any point in the program as many times as needed.

- int valueat(int *row*, int *col*)—this function returns the number placed in row *row* and column *col* of the grid ($0 \le row, col \le n-1$). Numbers in the grid are nonnegative integers not larger than 10^6.

Your solution (functions) must neither produce any output nor terminate the execution of the program. The program will terminate automatically after the last move.

Scoring

Your submission will gain a point for each match played in an optimal way: that is, maximizing the value of your score minus the opponent's score. This means that points may be gained even if the match is lost, provided that your program made optimal moves against the opponent and thereby minimized the loss.

Example

For $n = 3$ and the grid

2 8 1
3 5 5
3 2 7

a possible sequence of function calls when running a program that plays the *Game of Tokens* is:

```
-> startgame(3)
    <- valueat(0,0) = 2
    <- valueat(0,1) = 8
    <- valueat(0,2) = 1
    <- valueat(1,0) = 3
    <- valueat(1,1) = 5
    <- valueat(1,2) = 5
    <- valueat(2,0) = 3
    <- valueat(2,1) = 2
    <- valueat(2,2) = 7
-> opponentmove(1)
-> play() = 2
-> opponentmove(0)
```

Explanation of the example: As required, startgame is called first. In this example implementation, it calls valueat for each cell to return the numbers in the grid. The first move is made by the opponent, who picks up token 1 (call to opponentmove(1)). The player responded by picking up token 2. In the last move, the opponent takes token 0 (the only remaining token) and the game terminates. Both players made optimal moves, and the game ended with the opponent scoring 11 more points than the player.

/ Solution

The task's constraint ($n \le 40$) suggests that one can try some brute-force methods, such as a game-tree search or some variant of the meet-in-the-middle technique. However, it turns out that the token game can be reduced to another very simple game with an obvious strategy. To obtain that reduction, we take a closer look into the game result.

Some notations have to be introduced. We represent a rectangular board with numbers by a matrix:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

Let $T = \{0,1,\ldots,n-1\}$ denote the set of all tokens. Let T_i, for $i = 1,2$, denote tokens collected by the i-th player. The pair of sets T_1 and T_2 is a partition of the set T: $T_1 \cup T_2 = T$ and $T_1 \cap T_2 = \emptyset$. The advantage of the first player over the second player is determined by the formula

$$Res = \sum_{i,j \in T_1} a_{i,j} - \sum_{i,j \in T_2} a_{i,j}.$$

Let us try to express the value of *Res*—the game result—by other means. Let us investigate the influence of a player taking token number i. Firstly, the opponent certainly gets no points for numbers from the i-th row and i-th column. Moreover, the benefit from the field $a_{i,i}$ is doubled—besides denying the opponent the field's points, the player is also able to claim them.

Let us denote the sum of numbers of the i-th row of the matrix A as r_i and the sum of numbers of the i-th column as c_i:

$$r_i = \sum_{j \in T} a_{i,j}, \quad c_i = \sum_{j \in T} a_{j,i}.$$

We can try to use these symbols to express the game result. Let us count points according to the above reasoning. Assume that, for taking the i-th token,

the player gets $r_i + c_i$ points, because this action strips from the opponent the opportunity to get points for numbers from the i-th row and i-th column, and at the same time acquires the points for field $a_{i,i}$, which is counted twice in the sum $r_i + c_i$. This counting method gives the following total advantage for the first player.

$$Res' = \sum_{i \in T_1} (r_i + c_i) - \sum_{i \in T_2} (r_i + c_i).$$

How does this method compare to the previous one? Consider the element $a_{i,j}$. If both tokens i and j belong to the first player then $a_{i,j}$ is counted twice in the first sum. But if both tokes belong to the opponent then $a_{i,j}$ is subtracted twice because it is in the second sum. In the third case, if tokens i and j were split between players then $a_{i,j}$ is added in the first sum and subtracted in the second sum.

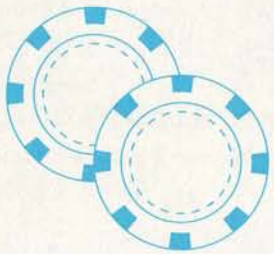
Therefore, we see that the advantage counted using the new method is twice the advantage computed with the regular rules. Thus, if we assign to token i the value

$$v_i = r_i + c_i,$$

then the doubled game result will be

$$2Res = \sum_{i \in T_1} v_i - \sum_{i \in T_2} v_i.$$

In this way, the token game reduces to a game in which each token has a fixed value v_i and the total number of points obtained by a player is the sum of possessed tokens. In such a game, a greedy strategy is optimal: always take the token with the largest value.



/ Straight *Lines*

Contest: Algorithmic Engagements 2006
Task author: Jakub Pawlewicz
Memory: 32 MB
http://main.edu.pl/en/archive/pa/2006/pro

We are given six integers, A_1, B_1, C_1, A_2, B_2, C_2, such that $A_1B_2 \neq A_2B_1$. These numbers are coefficients in the equations of two intersecting lines:

$$l_1 : A_1x + B_1y + C_1 = 0,$$
$$l_2 : A_2x + B_2y + C_2 = 0.$$

The lines divide the plane into four parts. We represent each part by any point with integer coordinates belonging to this part (but not belonging to any of the lines l_1, l_2). You are given a point (a, b) with integer coordinates representing one part. Find a point with integer coordinates (c, d) representing the same part, such that its distance from the point of intersection of lines l_1 and l_2 is the smallest possible.

Task

Write a program that:

- reads the equations of lines l_1 and l_2 and a point representing one part of the plane,
- finds the point with integer coordinates representing the given part which is closest to the point of intersection of lines l_1 and l_2,
- writes the answer to the output.

Input

The first line of the input contains three numbers, A_1, B_1, C_1, specifying the coefficients of l_1's equation. The second line contains three numbers, A_2, B_2, C_2, specifying the coefficients of l_2's equation. It is guaranteed that $A_1B_2 \neq A_2B_1$. The third and final line contains two integers, a, b, specifying the coordinates of a point representing one part of the plane. Point (a, b) does not belong to any of lines l_1, l_2. Every number x from the input falls within the range $-2{,}100{,}000{,}000 < x < 2{,}100{,}000{,}000$.

Output

Your program should write to the output two numbers, c, d, separated by a single space: the coordinates of a point (c, d) representing the given part, closest to the point of intersection of lines l_1 and l_2. If there are multiple such points, your program should output only one of them.

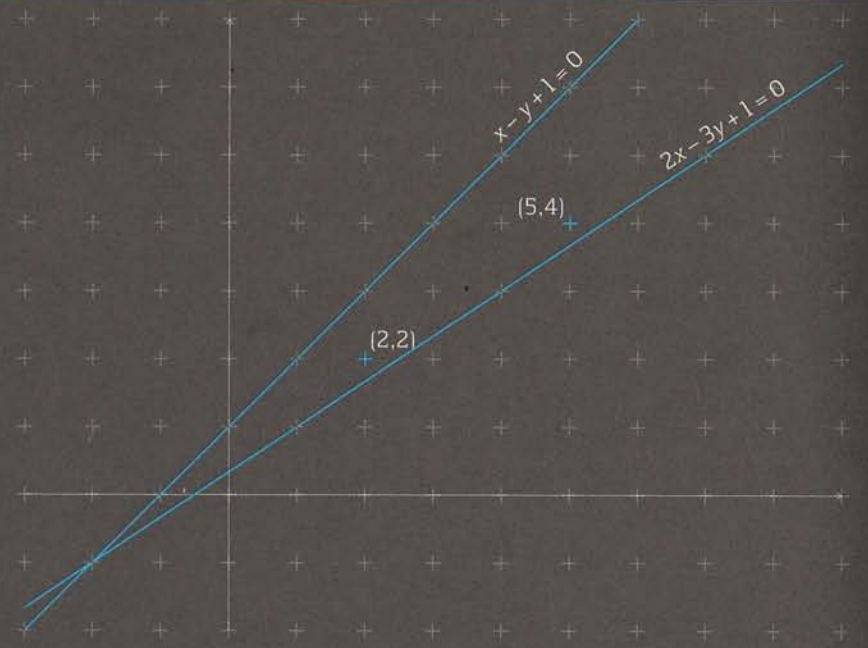
Example

For the input data:

1 −1 1
2 −3 1
5 4

the correct result is:

2 2



/ Solution

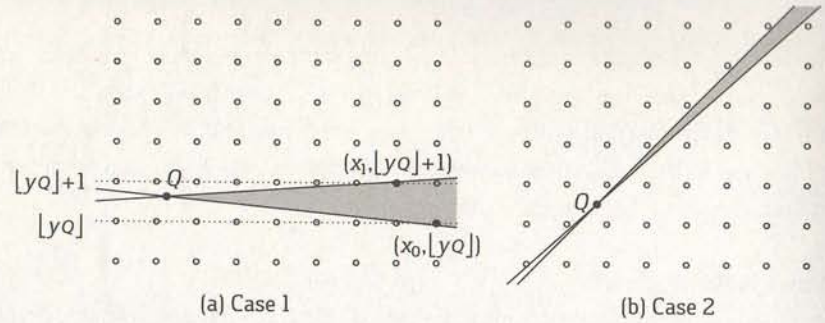
In order to simplify the task analysis we introduce some simple notations. Let $Q = (x_Q, y_Q)$ be the intersection of the lines l_1 and l_2. These lines divide the plane into four parts. We call each such part a *quarterplane* and we refer to the part pointed out in the task as the *examined quarterplane*. We say that a point with integer coordinates is a *lattice point*.

Where is the problem?

At the first glance the task seems to be easy. It is sufficient to scan lattice points inside the examined quarterplane and output the coordinates of the one closest to Q. How should such points be scanned? The simplest way is to focus only on points surrounding Q, hoping that there will be a point belonging to the examined quarterplane. We can take into account only those points at which the difference of each coordinate from the corresponding coordinate of Q is bounded by a constant. For example, we can set this constant to 5; then we need to check only about 100 lattice points. We describe this as a *brute force* method.

Unfortunately, there are instances in which the closest lattice point to Q, belonging to the examined quarterplane, is far, far away. Nevertheless, a brute force method reduces our task to a case in which the examined quarterplane is very "narrow". With respect to 90° rotation, two cases are possible (see figure 1).

In the first case (figure 1a) we can find a solution: it is either $(x_0, \lfloor y_Q \rfloor)$ or $(x_1, \lfloor y_Q \rfloor + 1)$, where x_0 and x_1 are the smallest integers whose points belong to the examined quarterplane. The second case (figure 1b) is still problematic, and we need more sophisticated tools to solve it.



(a) Case 1 (b) Case 2

Figure 1: *Narrow quarterplanes immune to a brute force method.*

Narrowing the search space of lattice points

We present a method that enables us to iterate efficiently over lattice points inside many typical geometric objects such as our quarterplane. We start by introducing some notations and definitions.

Let $\mathbb{R}$ denote the set of real numbers and $\mathbb{Z}$ the set of integers. For a set $A \subseteq \mathbb{R}$ we denote as A_+ the set $\{a \in A \mid a > 0\}$. We have already introduced an informal definition of a quarterplane; now we provide a more formal one.

Definition 1. For a point $O \in \mathbb{R}^2$ and vectors $\vec{u}, \vec{v} \in \mathbb{R}^2$ such that $\vec{u}$ is not parallel to $\vec{v}$ we define a *quarterplane* $Q[O, \vec{u}, \vec{v}]$ as the set of points $\{O + \alpha\vec{u} + \beta\vec{v} \mid \alpha, \beta \in \mathbb{R}_+\}$ (see figure 2).

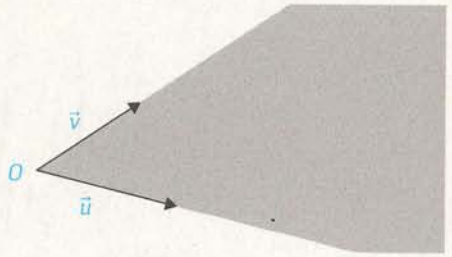


Figure 2: *Quarterplane $Q[O, \vec{u}, \vec{v}]$.*

We are interested in lattice points, though, and for those we need quarter-lattices.

Definition 2. For a point $O \in \mathbb{Z}^2$ and vectors $\vec{u}, \vec{v} \in \mathbb{Z}^2$ such that $\vec{u}$ is not parallel to $\vec{v}$ we define a *quarterlattice* $\mathcal{L}[O, \vec{u}, \vec{v}]$ as the set of points $\{O + \alpha\vec{u} + \beta\vec{v} \mid \alpha, \beta \in \mathbb{Z}_+\}$ (see figure 3).

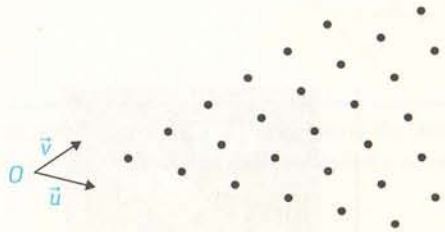


Figure 3: *Quarterlattice $\mathcal{L}[O, \vec{u}, \vec{v}]$.*

Quarterlattice $\mathcal{L}[O, \vec{u}, \vec{v}]$ represents a set of lattice points inside a quarterplane $Q[O, \vec{u}, \vec{v}]$ thus:

$$\mathcal{L}[O, \vec{u}, \vec{v}] \subseteq Q[O, \vec{u}, \vec{v}] \cap \mathbb{Z}^2.$$

If one wants to iterate over all lattice points inside a quarterplane, then in the above relation we would rather have an equation sign.

Definition 3. We say that a quarterlattice is *full* if the following equation holds:

$$\mathcal{L}[O, \vec{u}, \vec{v}] = Q[O, \vec{u}, \vec{v}] \cap \mathbb{Z}^2.$$

A full quarterlattice is a convenient way of representing all lattice points inside some quarterplane. An ability to construct full quarterlattices is key to a solution.

Constructing full quarterlattices. Assume that O is some fixed lattice point. The most basic full quarterlattice is $\mathcal{L}[O, (1,0), (0,1)]$. In order to create other quarterlattices the following fact is sufficient:

Fact 4. If quarterlattice $\mathcal{L}[O,\vec{u},\vec{v}]$ is full then quarterlattices $\mathcal{L}[O,\vec{u},\vec{u}+\vec{v}]$ and $\mathcal{L}[O,\vec{u}+\vec{v},\vec{v}]$ are also full.

Proof. Any lattice point $P = O + \alpha\vec{u} + \beta\vec{v}$, $\alpha, \beta \in \mathbb{Z}_+$, belonging to quarterplane $Q[O,\vec{u},\vec{v}]$, falls into one of the following sets (see figure 4):
(i) $Q[O,\vec{u},\vec{u}+\vec{v}]$ if $\alpha > \beta$,
(ii) $Q[O,\vec{u}+\vec{v},\vec{v}]$ if $\alpha < \beta$,
(iii) a ray starting at O going off in direction $\vec{u}+\vec{v}$ if $\alpha = \beta$.

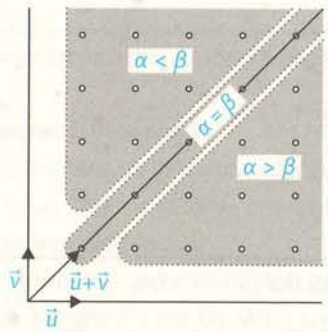


Figure 4: *Three parts of a quarterlattice.*

In case (i) point P belongs to quarterlattice $\mathcal{L}[O,\vec{u},\vec{u}+\vec{v}]$ since $P = O + (\alpha-\beta)\,\vec{u} + \beta(\vec{u}+\vec{v})$, whereas in case (ii) point P belongs to quarterlattice $\mathcal{L}[O,\vec{u},\vec{u}+\vec{v}]$ since $P = O + \alpha(\vec{u}+\vec{v}) + (\beta-\alpha)\vec{v}$. □

Applying Fact 4 many times, we reach the following conclusion:

Corollary 5. If quarterlattice $\mathcal{L}[O,\vec{u},\vec{v}]$ is full then quarterlattices $\mathcal{L}[O,\vec{u},k\vec{u}+\vec{v}]$ and $\mathcal{L}[O,\vec{u}+k\vec{v},\vec{v}]$ are also full for every $k \in \mathbb{Z}_+$.

Goal. Our goal is to build a quarterlattice such that there will be a clear way to output a small set of candidates for a solution. We build more and more narrow quarterlattices by Corollary 5. Quarterlattice $\mathcal{L}[O,\vec{u},\vec{v}]$ represents some new coordinate system with origin at O and axes set by $\vec{u}$ and $\vec{v}$. It means that any point (x,y) is represented by a point (α,β) such that $(x,y) = O + \alpha\vec{u} + \beta\vec{v}$. As we construct increasingly narrow quarterlattices, the examined quarterplane becomes increasingly wide from the point of view of the new coordinate system (see figure 5).

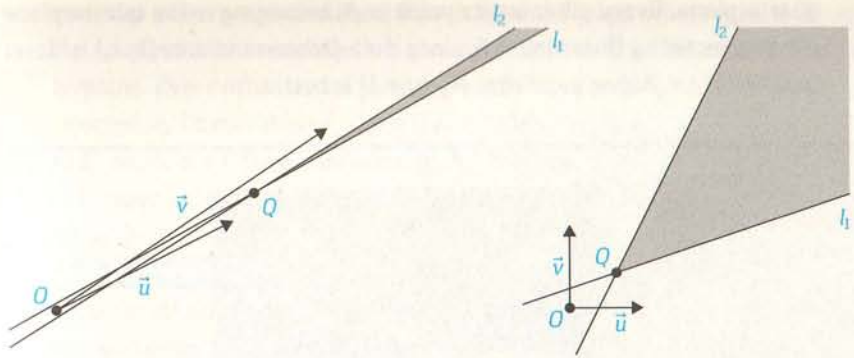


Figure 5: *A narrow quarterlattice $\mathcal{L}[O,\vec{u},\vec{v}]$ comprising the examined quarterplane (on the left) and a look at the examined quarterplane from the point of view of $\mathcal{L}[O,\vec{u},\vec{v}]$ (on the right).*

It is possible to find a quarterlattice $\mathcal{L}[O,\vec{u},\vec{v}]$ such that, from the point of view of this quarterlattice, the following conditions hold:
(i) The entire examined quarterplane between lines l_1 and l_2 is inside quarterplane $Q[O,\vec{u},\vec{v}]$.
(ii) Coordinates (α_Q,β_Q) of point Q are close to the origin; that is, $0 \le \alpha_Q,\beta_Q < 1$.
(iii) The angle between the X axis and line l_1 is less than 45°.
(iv) The angle between the Y axis and line l_2 is less than 45°.
In such a situation—thanks to the observation below—it is easy to give candidates for the required point.

Observation 6. Let $\alpha_1,\beta_1,\alpha_2,\beta_2 \in \mathbb{Z}_+$ be such that $\alpha_1 \le \alpha_2$ and $\beta_1 \le \beta_2$. Then the distance of point $P_1 = O + \alpha_1\vec{u} + \beta_1\vec{v}$ from point Q is not greater than the distance of point $P_2 = O + \alpha_2\vec{u} + \beta_2\vec{v}$ from point Q.

Proof. In vectors $\vec{u}$ and $\vec{v}$ both coordinates are nonnegative, thus each coordinate of vector $\overrightarrow{QP_1}$ is not greater than the corresponding coordinate of vector $\overrightarrow{QP_2}$; hence the requested inequality between Euclidean distances follows. □

We examine candidates from the point of view of the coordinate system of quarterlattice $\mathcal{L}[O,\vec{u},\vec{v}]$. If point $(1,1)$ belongs to the examined quarterplane then by Observation 6 it is the only candidate; thus this is the required point. Otherwise there are two possible cases:
1. Point $(1,1)$ lies on line l_1 or underneath (see figure 6). Then it is sufficient to consider only two candidates: (k_1,k_1) and (k_2,k_2+1), where $k_1,k_2 \in \mathbb{Z}_+$

are the smallest numbers such that given points belong to the examined quarterplane. Every other lattice point (α,β) belonging to the quarterplane can be rejected by Observation 6, since if $\alpha \geq \beta$ then candidate (k_1,k_1) is closer to Q and if $\alpha < \beta$ then candidate (k_2,k_2+1) is better.

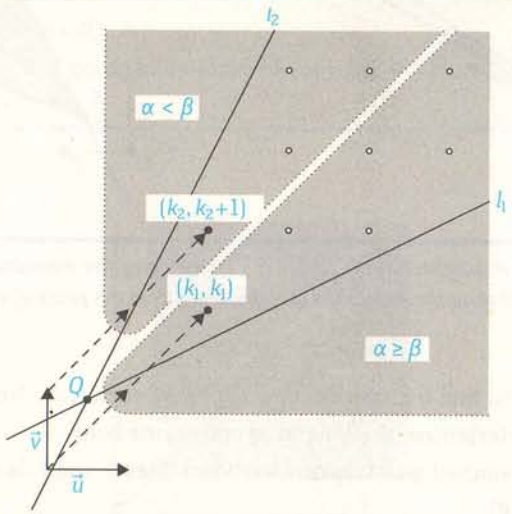


Figure 6: *Point* $(1,1)$ *is below line* l_1.

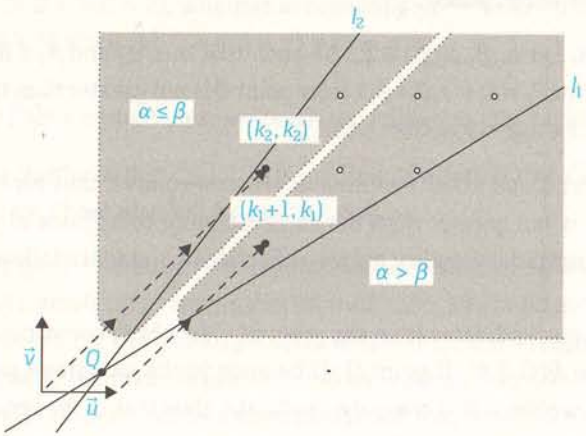


Figure 7: *Point* $(1,1)$ *is above line* l_2.

2. Point $(1,1)$ lies on line l_2 or above it (see figure 7). Then it is sufficient to consider only two candidates: (k_1+1,k_1) and (k_2,k_2), where $k_1,k_2 \in \mathbb{Z}_+$ are the smallest numbers such that given points belong to the examined quarterplane. Every other lattice point (α,β) belonging to the quarterplane can be rejected by Observation 6, since if $\alpha > \beta$ then candidate (k_1+1,k_1) is closer to Q, and if $\alpha \leq \beta$ then candidate (k_2,k_2) is better.

In both cases we get two candidates, and it is sufficient to check which one is closer to Q.

Reaching the goal. Let us use another, more convenient equation $bx - ay = c$ to represent straight lines. Vector $\overrightarrow{(a,b)}$ is parallel to such a line, and we call it a *direction vector*. Of course, it is easy to express the lines from the task using the introduced representation:

$$l_1 : b_1x - a_1y = c_1,$$
$$l_2 : b_2x - a_2y = c_2,$$

Moreover, because we are solving the case from figure 1b, we can further assume that coefficients a_1,b_1,a_2,b_2 are nonnegative.

To check whether the angle between a line $bx - ay = c$ and the X or Y axis is smaller than 45°, it is enough to test direction vector $\overrightarrow{(a,b)}$. Thus we can restate conditions (iii) and (iv) from the *Goal* section:

(iii) $a_1 > b_1$,

(iv) $a_2 < b_2$.

Let us find how the equation of a line changes after altering a coordinate system. Assume a line l can be expressed by equation $bx - ay = c$ in the coordinate system for quarterlattice $\mathcal{L}[O,\vec{u},\vec{v}]$. What does the line look like in the coordinate system for quarterlattice $\mathcal{L}[O,\vec{u},\vec{u}+\vec{v}]$? Let $P = (\alpha,\beta)$ be an arbitrary point in the system of the new quarterlattice. The coordinates of P in the previous system are $(\alpha+\beta, \beta)$, because $O + \alpha\vec{u} + \beta(\vec{u}+\vec{v}) = O + (\alpha+\beta)\vec{u} + \beta\vec{v}$. Therefore, if point P belongs to line l, then $b(\alpha+\beta) - a\beta = c$ must hold, and we conclude that the equation of the line after altering the coordinates is $bx - (a-b)y = c$. Similarly, the equation of the line in the coordinate system for quarterlattice $\mathcal{L}[O,\vec{u}+\vec{v},\vec{v}]$ is $(b-a)x - ay = c$. Repeating the above reasoning k times leads to the following property:

Fact 7. If the equation of a line is $bx - ay = c$ in the coordinate system for quarterlattice $\mathcal{L}[O,\vec{u},\vec{v}]$, then the equation of this line is

$$bx - (a - kb)y = c$$

in the coordinate system for quarterlattice $\mathcal{L}[O, \vec{u}, k\vec{u} + \vec{v}]$, and

$$(b - ka)x - ay = c$$

in the coordinate system for quarterlattice $\mathcal{L}[O, \vec{u} + k\vec{v}, \vec{v}]$, where $k \in \mathbb{Z}_+$.

This fact suggests how to refine a quarterlattice if conditions (iii) or (iv) are not fulfilled. Namely, in such case we have

1. $a_2 \geq b_2$ and $a_1 \geq b_1$ or
2. $a_1 \leq b_1$ and $a_2 \leq b_2$.

There are no other cases, since vector $\overrightarrow{(a_1, b_1)}$ is below $\overrightarrow{(a_2, b_2)}$. From this mutual relation of the vectors we also conclude that $a_1 > 0$ and $b_2 > 0$.

Suppose that the first case is true. Let $k_2 = \lfloor a_2/b_2 \rfloor$ and assume that we move to quarterlattice $\mathcal{L}[O, \vec{u}, k_2\vec{u} + \vec{v}]$. Then, coefficients a_1 and a_2 of both lines are altered. Let a'_1 and a'_2 be the altered coefficients:

$$a'_1 = a_1 - k_2 b_1,$$
$$a'_2 = a_2 - k_2 b_2.$$

Then $a'_2 = a_2 \bmod b_2$ and so $a'_2 < b_2$ holds. It means that moving to the new quarterlattice makes condition (iv) true. If $b_1 = 0$ then, of course, $a'_1 > b_1$, which makes (iii) also true. Therefore, let us assume that $b_1 > 0$ and test what the relation is between a'_1 and b_1. Let $k_1 = \lfloor a_1/b_1 \rfloor$. Since $\overrightarrow{(a_1, b_1)}$ is below $\overrightarrow{(a_2, b_2)}$, we have $k_2 \leq k_1$. If $k_2 = k_1$ then $a'_1 = a_1 \bmod b_1$ and $a'_1 > b_1$. For $k_2 < k_1$ there is $a'_1 \leq b_1$ where the equality $a'_1 = b_1$ holds only if $k_2 + 1 = k_1$ and $a_1 = k_1 b_1$.

So, after altering direction coefficients there is a hope that $a'_1 > b_1$, and this means that (iii) is fulfilled and the goal is reached. Otherwise we have $a'_1 \leq b_1$, but then we are in the second case in which we deduct analogously. In general, it looks like the Euclidean algorithm applied simultaneously to pairs (a_1, b_1) and (a_2, b_2). As soon as the algorithm "diverges", i.e. $k_1 \neq k_2$, we reach our conditions.

The above procedure allows us to find vectors $\vec{u}$ and $\vec{v}$ of the searched quarterlattice in logarithmic time. We still need to find point O. For this it is enough to find out the coordinates of the intersection of the lines in the new coordinate system and round them down to the nearest integer. The entire discussion is summarized in the pseudocode below.


```
ū := (1,0); v̄ := (0,1)
while not (a1 > b1 and a2 < b2) do
    if a2 ≥ b2 then
        {The angle between X axis and each line is not greater than 45°}
        k := ⌊a2/b2⌋
        v̄ := kū + v̄
        a1 := a1 − kb1; a2 := a2 − kb2
    else {a1 ≤ b1}
        {The angle between Y axis and each line is not greater than 45°}
        k := ⌊b1/a1⌋
        ū := ū + kv̄
        b1 := b1 − ka1; b2 := b2 − ka2
(αQ, βQ) := intersection of b1x − a1y = c1 and b2x − a2y = c2
O := ⌊αQ⌋ū + ⌊βQ⌋v̄
```

Summary of the solution

The entire solution appears as follows.

1. Use a brute force method to find a solution or to reduce the problem to the case of a narrow quarterplane.
2. Apply the method of narrowing the search space of lattice points to obtain quarterlattice $\mathcal{L}[O, \vec{u}, \vec{v}]$ in which the examined quarterplane would be "wide enough".
3. Determine candidates (one or two) in quarterlattice $\mathcal{L}[O, \vec{u}, \vec{v}]$.
4. Convert back coordinates of candidates to the initial coordinate system and choose that which is closest to the point Q.

The key point in the presented solution is the ability to deal with quarterlattices. The method of narrowing the search space of lattice points delivers a convenient and efficient way to iterate over lattice points inside a quarterplane. However, it has many more applications. For example—after some modifications—one can use it to iterate over all points inside some geometric objects such as triangles or even arbitrary polygons with rational coordinates. This method is especially useful with narrow objects. We leave it to the reader to find other problems where it can be applied.

MARCIN PILIPCZUK

Marcin Pilipczuk's interests used to focus mostly on mathematics, occasionally also touching on computer science and physics. In high school he won gold medals in two international olympiads: one in mathematics and one in physics. Computer science became his main area of interest during his M.S. studies, and this change led to a championship in the ACM-ICPC World Finals in 2007, which he won for the University of Warsaw in a team with Marek Cygan and Filip Wolski. For many years he has taught programming courses for high school students, and has occasionally authored programming tasks for Polish programming competitions. Currently, as a Ph.D. student at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Marcin Pilipczuk conducts research in parameterized complexity and moderately-exponential algorithms. At times he also engages in graph theory and some aspects of calculus.

As the father of two young children, he does not have any free time. Before, he used to play board games, hike in the mountains and read a lot of books, mostly fantasy.

/ Guilds

Contest: 17th Polish Olympiad in Informatics
Task author: Marcin Pilipczuk
Memory: 64 MB
http://main.edu.pl/en/archive/oi/17/gil

King Byteasar faces a serious matter. Two competing trade organizations, The Tailors Guild and The Sewers Guild, asked, at the same time, for permission to open offices in each town of the kingdom.

There are n towns in Byteland. Some of them are connected with bidirectional roads. Each of the guilds postulates that every town should:

- have an office of the guild, or
- be directly connected to another town that does.

The king, however, suspects foul play. He fears that if there is just a single town holding the offices of both guilds, it could lead to a clothing cartel. For this reason he seeks your help.

Input

Two integers, n and m ($1 \le n \le 200{,}000$, $0 \le m \le 500{,}000$), are given in the first line of the input. These specify, respectively, the number of towns and roads in Byteland. The towns are numbered from 1 to n. Then the roads are given as follows: the $(i + 1)$-th line of the input describes the i-th road; it holds the numbers a_i and b_i ($1 \le a_i, b_i \le n$, $a_i \ne b_i$), specifying that the i-th road connects the towns a_i and b_i. Every pair of towns is (directly) connected by at most one road. The roads do not cross—meeting only in towns—but may lead through tunnels and overpasses.

Output

Your program should write one word in the first line of the output: TAK (Polish for *yes*) if the offices can be placed in towns according to these rules, or NIE (Polish for *no*) otherwise. If the answer is TAK, then the following n lines should give an exemplary placement of the offices. Thus the $(i + 1)$-th line should hold:

- the letter K if there should be an office of The Tailors Guild in town i, or
- the letter S if there should be an office of The Sewers Guild in town i, or
- the letter N if there should be no office in town i.

Example

For the input data:

7 8
1 2
3 4
5 4
6 4
7 4
5 6
5 7
6 7

the correct result is:

TAK
K
S
K
S
K
S
N

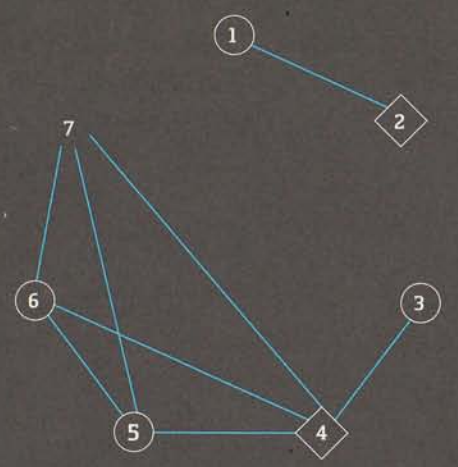


Figure. *The towns in which an office of The Tailors Guild should open are marked with circles, whereas those in which an office of The Sewers Guild should open are marked with rhombi.*

/ Solution

I would like to show you my favorite type of contest problems—those in which there are myriads of solutions.

In the great majority of problems you encounter at a programming contest, you are supposed to find an *optimal* solution—the single optimal one, or one from a very narrow choice of optimal solutions. In such cases, all algorithms are usually extremely similar to the jury solution—sometimes they are a bit slower, sometimes they are smarter in some way, but they rarely use techniques unknown to the author of the problem.

However, there are other problems that do not follow this scheme. In such problems we are given some conditions relating to the required solution that make finding a solution fairly nontrivial, but that nevertheless leave 'lots of freedom' in its construction. Usually there is no requirement for the solution to be optimal. When solving such problems, the contestants—always very creative—often invent weird methods and techniques, surprising the author and the jury with their originality.

Let us proceed to a few examples. I would like to present two interesting problems with "lots of freedom". It is worth mentioning that in both problems—after a correct way to construct a solution has been discovered—the implementation is extremely easy, making these problems even more enjoyable.

Guilds

Guilds was one of the simplest problems in the first stage of the Polish Olympiad in Informatics some time ago. Let us translate the problem statement into the language of graph theory: given a graph $G = (V, E)$ with n vertices and m edges, we are to find two *disjoint* sets of vertices $K, S \subseteq V$ (towns where the offices of the tailors and sewers guilds will be open), so that both sets are in some sense *dense* in the vertex set V. Such sets K and S that satisfy the conditions of the problem statement are called, in the literature, *dominating sets*: formally, a set $X \subseteq V$ is a *dominating set* in a graph $G = (V, E)$ if each vertex of G either belongs to X or has at least one neighbor in the set X. Thus, in the problem *Guilds*, we are to find two disjoint dominating sets in a given graph G.

Can we always succeed? It is quite easy to see that the answer is no: if there is a town in Byteland without any road going out of it (i.e. an isolated vertex in the graph G), there is no solution—in this vertex an office of only one guild can be placed, so the second guild will not service (dominate) this vertex. If we

spend a few more minutes drawing examples on a sheet of paper, we should convince ourselves that the isolated vertices are probably the only obstacle for king Byteasar. Let us try to find an algorithm that places the offices of the guilds in Byteland, provided that there are no isolated vertices.

Let us start with a few simplifications. First, clearly the connected components of G can be considered independently. Second, any superfluous edge in the graph only *helps* us in the constructions: if we suspect that we can find a solution in any graph without isolated vertices, then the hardest case would be to find it in a tree (connected acyclic graph). Third, we do not gain anything by leaving a town without any office. Thus, we can assume that we will build an office in each town; in other words, we are to *partition* the vertex set V into two dominating sets K and S.

Let us assume for a moment that G is a tree, and let v be its arbitrary leaf (a vertex of degree one). Without loss of generality, we can assume that $v \in K$; that is, we construct an office of the tailors guild in the vertex v. Then, in the unique neighbor of v—denote it by u—we need to place an office of the sewers guild (i.e. take $u \in S$), as otherwise S would not dominate the vertex v. By the same argument, if u has any other neighbors that are leaves, all these neighbors must belong to K, as otherwise they would not be dominated by the set K. So maybe let us assign all neighbors of u to the set K—then they would be dominated by both S (due to the vertex u) and K. And assign neighbors of neighbors of u again to set S, and so on.

Let us summarize the aforementioned algorithm: for each connected component of G we choose its arbitrary spanning tree, we root it in an arbitrary vertex, and we assign the vertices of the tree to the sets K and S in an alternating fashion—say, we assign vertices at an even distance from the root to K and at an odd distance to S. It is easy to see that this construction is correct: if we look only at the edges of the chosen spanning tree, each vertex has only neighbors with offices of the other guild than the vertex itself. Thus, if no connected component consists only of one vertex, both K and S are dominating sets in G.

Observe that if we choose the spanning tree in a smart way—e.g. we use the tree of a depth-first search (DFS)—our algorithm is very easy to implement, because we do not need to store the tree explicitly in memory. Below we present pseudocode for such a solution.


```
Algorithm Guilds(V, E)
    for all v ∈ V do
        guild[v] := 0
        {0—not yet assigned, 1—tailors guild, 2—sewers guild}
    for all v ∈ V do
        if guild[v] = 0 then
            DFS(v, 1)  {in the root we place an office of the tailors guild}

Function DFS(v, g)
    guild[v] := g
    for all u neighbor of v do
        if guild[u] = 0 then
            DFS(u, 3 − g)
            {in the children of v we place offices of the second guild}
```

The above solution was the one I had in mind when proposing the problem. As expected, the contestants turned out to be very creative and, among many inventive and obscure solutions, I found the following two both to be simpler than mine.

First solution by creative contestants. Initially, each vertex of the graph has no office assigned to it. We go over all edges of the graph in an arbitrary order. When considering an edge uv, if we did not assign any office to at least one of the vertices u or v, we assign offices to both of these vertices in such a way that the offices in u and v are of different guilds (note that this can always be done, regardless of whether one of these vertices already has an office assigned to it, or both are unassigned). If both u and v have their offices already assigned, we do nothing.

To see that this solution is correct, take an arbitrary vertex v and consider the moment when we assign an office to the vertex v. It happens when we consider some edge coincident with v, say uv. At this moment, we assign the offices in such a way that u and v receive offices of different guilds—and therefore v is dominated by both guilds.

This solution is extremely simple to implement, as shown below.

```
Algorithm Guilds2(V,E)
  for all v ∈ V do
    guild[v] := 0
    {0—not yet assigned, 1—tailors guild, 2—sewers guild}
  for all uv ∈ E do
    if guild[u] = 0 and guild[v] = 0 then
      guild[u] := 1; guild[v] := 2
    else if guild[u] = 0 then
      guild[u] := 3 − guild[v]
    else if guild[v] = 0 then
      guild[v] := 3 − guild[u]
```

Second solution by creative contestants. Again, initially each vertex has no office assigned to it. We go over the vertices of the graph in an arbitrary order. If we find a vertex that has no office assigned to it as yet, we place an office of the tailors guild in it, and, moreover, we place offices of the sewers guild in all neighboring vertices that have no office assigned to them.

To show the correctness of this solution, first notice that no two vertices with an office of the tailors guild are adjacent, since whenever we place an office of the tailors guild somewhere, we immediately pass all the unassigned neighbors to the sewers guild. In graph theory, a set of vertices with no two vertices connected by an edge is called an *independent set*. Note that the set of vertices of the graph where the offices of the tailors guild were placed (i.e. the set K) is not only an independent set, but an *inclusion-wise maximal* independent set: we cannot extend it by any vertex without closing already-opened offices of the tailors guild. Summing up, in the language of graph theory, the above solution constructs an inclusion-wise maximal independent set, places offices of the tailors guild in the vertices of the independent set, and places offices of the sewers guild in the remaining vertices.

Let us now show that, in a graph without isolated vertices, any inclusion-wise maximal independent set and its complement are both dominating sets. For the first part, it is sufficient to observe that any vertex of the graph is either in the chosen inclusion-wise maximal independent set or has a neighbor in it, since otherwise we could extend the independent set with the considered vertex. For the second part, we need to use the assumption that the graph does not contain isolated vertices: any vertex of the graph that is chosen to be in the independent set has at least one neighbor; this neighbor cannot be in the independent set, and thus dominates the considered vertex.

Again, the pseudocode below shows that this solution is very easy to implement.

```
Algorithm Guilds3(V,E)
  for all v ∈ V do
    guild[v] := 0
    {0—not yet assigned, 1—tailors guild, 2—sewers guild}
  for all v ∈ V do
    if guild[v] = 0 then
      guild[v] := 1
      for all u neighbor of v do
        if guild[u] = 0 then
          guild[u] := 2
```

The presented algorithms for the problem *Guilds* are obviously not the only ones, but in my opinion they are the simplest. It is worth noting that *Guilds* was the simplest problem in the first stage of the 17th Polish Olympiad in Informatics, and almost half of the contestants received perfect scores for their solutions.

A short note on the theoretical background

Let us now consider the problem *Guilds* from a broader perspective. We will see that behind a very simple problem we can find a nice piece of theoretical computer science, lying somewhere between algorithmics and complexity theory.

We have already established that our problem, in the language of graph theory, boils down to one of finding two disjoint dominating sets in a given graph. What happens if we demand more pairwise disjoint sets? In other words, what happens if, in Byteland, there are not only tailors and sewers guilds, but also silkworm breeders or Gore-Tex manufacturers?

It turns out that the problem with three guilds is already NP-complete—which means that we should not expect an algorithm that solves the problem in polynomial time in the size of the graph. For a given graph G, the largest natural number k, such that in G we can construct offices of k guilds that satisfy the conditions from the problem statement, is called the *domatic number* of the graph G. As deciding if the domatic number of a graph is at least 3 is already hard, computing the exact value of the domatic number is even harder.

The situation here is thus almost the same as for the case of the *chromatic number*—the minimum number of colors needed to color the vertices of the graph such that no two adjacent vertices receive the same color (here we partition the set of vertices of the graph into independent sets). It is easy to see that two colors suffice if and only if the graph G is bipartite (which is easy to check). It was proven that deciding whether three colors suffice is NP-complete even if we know that the input graph is planar and each vertex has degree exactly 4. Both the chromatic and domatic number can be computed in time $O(2^n p(n))$ (where $p(n)$ is some polynomial) by a non-trivial application of the inclusion-exclusion principle. The question of whether the constant 2 in the base of the exponent can be improved is now a major open problem.

Let us go back for a moment to the concept of a dominating set. It turns out that finding a minimum dominating set is a very hard task, with many applications: it models the problem of finding the cheapest way to place sensors (or resources) in a network such that each node is controlled by a sensor (or has quick access to a resource, respectively). Let us consider a decision version of this problem, where, beside the graph G, we are given an integer k and we are to decide whether the graph G admits a dominating set of size at most k. This problem is also NP-complete, and similar in terms of computing domatic and chromatic numbers.

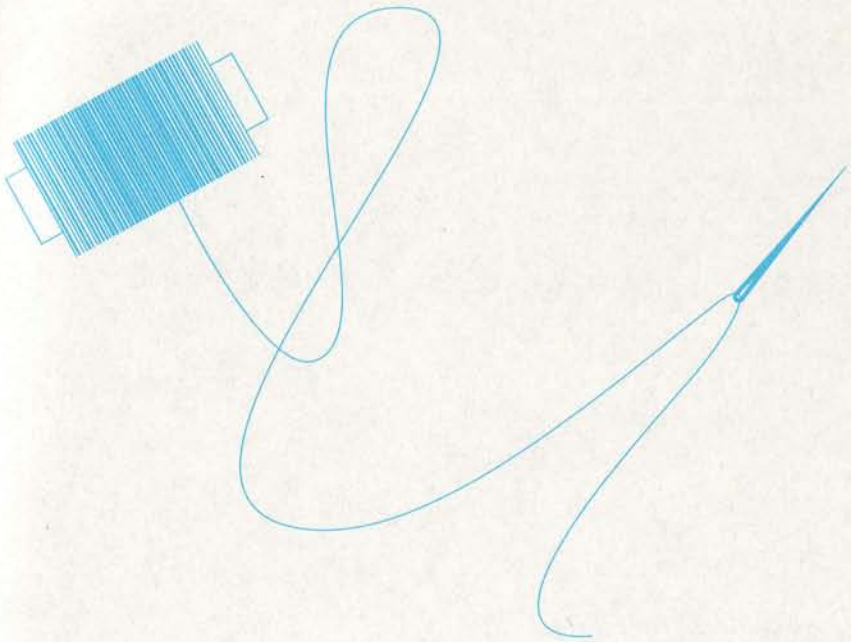
In order to consider the difficulty of the problem of finding a minimum dominating set from another perspective, let us recall a similar problem of finding a minimum vertex cover of a graph. A set of vertices $X \subseteq V$ is called a *vertex cover* of the graph $G = (V, E)$ if each edge of G has at least one endpoint in the set X. Intuitively, a vertex cover of a graph is a set of vertices that dominates each *edge*, whereas a dominating set has to dominate each *vertex*.

Note that we can decide whether a given graph G with n vertices has a vertex cover or dominating set of size at most k by checking all possible answers. Such a solution works in time roughly $\binom{n}{k}$, which, for k being much smaller than n, is of order roughly n^k.

However, in the vertex cover problem we can use a much smarter approach. Consider the following branching algorithm: as long as G contains at least one edge, take an arbitrary edge, say uv, and branch into two sub-cases: in one case we take u to the constructed vertex cover, and in the second case we take v. Whenever we take a vertex to the constructed vertex cover, we delete it and all its incident edges, and decrease k by one. This algorithm finds a solution if, and only if, in one of the sub-cases we find a solution before k becomes negative. A careful implementation of this algorithm runs in $O(2^k(n+m))$ time and, thus,

is very efficient for small values of k; say $k \sim 15$. By contrast, for $k \sim 15$ the brute-force algorithm with time complexity n^k is very slow.

Can we do a similar trick for the dominating set problem? It turns out that the answer is probably no: it was shown that the problem of finding a minimum dominating set is hard for a class of problems called $W[2]$. Without getting into the technical details of the definitions, it means that the existence of an algorithm that solves this problem in time similar to the algorithm for the vertex cover problem described in the previous paragraph would imply the existence of such algorithms for a large class of other problems—and this is considered unlikely.



/ Reconstruction *of Byteland*

Contest: Algorithmic Engagements 2008
Task author: Marcin Pilipczuk
Memory: 32 MB
http://main.edu.pl/en/archive/pa/2008/odb

Everyone get to work! Byteland needs to be reconstructed after a devastating war!

You are responsible for assigning postal codes to Bytean cities. Each city should receive one postal code—a positive integer not greater than 10^9. Different cities should be assigned different postal codes.

The Bytean Postal Service is organized in quite a strange way: a letter can be sent from city A to city B only if the postal codes of the two cities have a common divisor greater than 1. Obviously, one of your objectives is that it should be possible to send letters directly from every city to every other city after the postal codes are assigned.

Additionally, the new anti-corruption law requires that, for each set of cities containing more than half of the Bytean cities, postal codes assigned to cities within this set must not have any common divisor greater than one.

Task

Write a program that:

- reads the number of cities of Byteland from the input,
- assigns a postal code to each Bytean city,
- writes the assignment to the output.

Input

The first and only line of input contains one integer n ($4 \le n \le 100$) specifying the number of cities of Byteland.

Output

The output should consist of exactly n lines. The i-th line should contain one positive integer not greater than 10^9—the postal code of the i-th Bytean city. You can assume that for each possible input there exists a valid assignment of postal codes to the cities. If there are multiple correct solutions, your program should output any one of them.

Example

For the input data:

5

the correct result is:

714
2090
4485
29029
215441

/ Solution

The second problem from the line of 'lots of freedom' problems that I would like to present here is *Reconstruction of Byteland*. Although this problem does not have such a deep and interesting theoretical background as was shown for the *Guilds* problem, it has a handful of possible solutions and you can show your creativity while solving it.

After reading the problem statement, it is quite natural to look at the postal codes—the numbers assigned to the cities—from the point of view of their factorization into prime numbers. Each pair of postal codes needs to have a common prime factor; however, no set of more than half of the cities may have such a factor. Moreover, the cities need to receive pairwise different postal codes.

Let us try the following solution: take $\binom{n}{2}$ prime numbers and assign a different prime number to each pair of cities; the postal code of a city is the product of all numbers assigned to pairs that contain this city. This way, the postal codes of each pair of cities have a common divisor, and this does not hold for any trio of cities—let alone half of all the cities. Moreover, it is easy to see that the postal codes generated in this way are pairwise different.

Unfortunately, in Byteland there could be 100 cities, so we need 4950 prime numbers. The 4950th smallest prime number is 48049, and our postal codes are products of 99 such numbers. Thus, the postal codes generated in this way are very large: much larger than the required upper bound of 10^9. Precise calculations show that this technique works for up to six cities without violating the upper bound.

What can be improved? Note that we have fulfilled the anti-corruption law very strictly—too strictly: not only can any set of more than half of all the cities not have a common divisor, but no trio of cities can have such a divisor either. Let us now try to be less strict: let us partition the cities into five roughly equal parts, and assign to each pair of parts one of the $\binom{5}{2} = 10$ smallest prime numbers. The postal code of a city is a product of four prime numbers assigned to pairs of parts in which the city in question is contained.

Is this a correct solution? Unfortunately, it requires two corrections. The first problem is easy to notice: the postal codes assigned to the cities of each part are equal. Luckily, this is easy to fix: take the next 100 prime numbers and multiply the code of each city by a different prime number. Let us check whether we will fit into the 10^9 upper bound: the tenth prime number is 29, the hundred-and-tenth is 601, and we have $29^4 \cdot 601 < 10^9$.

The second error is much more subtle: this solution does not work for $n = 7$. In this case, the parts into which we divide the set of all cities have the sizes 2, 2, 1, 1 and 1. The common divisor of the two parts with two cities each will be a common divisor of the postal codes of four cities, which is more than half of all the cities. This unfortunate case can be solved in a few ways. One is to partition the cities into six groups. One can also solve this case manually and hard-code the solution.

This ends the description of my solution of the problem *Reconstruction of Byteland*. Naturally, this is only one of many possible algorithms for this problem. I conclude with a fun fact: the bounds in the problem statement—100 cites and postal codes not larger than 10^9—did not originate from the technical difficulties of solutions—the algorithm described above generalizes to a larger number of cities easily with only a slight increase in the upper bound on the numbers—but are required by the program that verifies the solutions. Note that this program needs to factorize all postal codes, check whether each pair of codes has a common divisor and verify that no divisor appears in more than half of the cities. This is much more work than is expected from a solution to this problem!

MICHAŁ PILIPCZUK

Michał Pilipczuk completed a double degree program in Computer Science and Mathematics at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. Currently he is a Ph.D. student at the University of Bergen, Norway. His scientific research focuses primarily on parameterized complexity and kernelization. However, his broader interests include combinatorics and especially its application in computer science.

During his university years, Michał Pilipczuk participated in several programming competitions, including the ACM-ICPC, where he won a silver medal in 2010 at the World Finals in Harbin (together with Karol Kurach and Krzysztof Pawłowski).

For the past few years, Michał Pilipczuk has been involved in the Polish Mathematical Olympiad, where he was mostly responsible for the selection of problems. He was the deputy leader of the Polish team at the International Mathematical Olympiad three times. During the holiday season you can usually find him in the mountains or some other wilderness area.

/ Army *Training*

Contest: Algorithmic Engagements 2010
Task author: Michał Pilipczuk
Memory: 64 MB
http://main.edu.pl/en/archive/pa/2010/pol

The Byteland Army plans to conduct the greatest military exercise in its history this weekend. The exercise will take place on the training ground in Northern Bytetown. The officers of the Byteland Army know this training ground perfectly; they do not, however, know the assignments. This is why they have requested your help, recruit!

Your superiors know exactly where the strategic spots on the training ground are located. During the exercise they will be asked multiple times to capture various areas of the training ground. One of the most crucial decisions will be the allocation of forces and determining the strength needed to capture a particular area—this strength should be proportional to the number of strategic spots in the attacked area. Your task is to determine for each area, represented as a polygon with vertices at the strategic spots, the number of strategic spots lying strictly inside the interior of the area.

Input

In the first line of the input two integers, n and m, are given ($3 \le n \le 1000$, $1 \le m \le 100{,}000$), specifying respectively the number of strategic spots on the training ground and the number of queries. The strategic spots are numbered from 1 to n.

The next n lines give descriptions of the strategic spots. In the i-th line two integers, x_i and y_i ($-10^9 \le x_i, y_i \le 10^9$), are given, specifying the coordinates of the i-th strategic spot. No three strategic spots are collinear.

The next m rows contain the descriptions of the m queries. Each description begins with an integer k_j ($3 \le k_j \le n$), specifying the number of vertices of the polygon. It is followed by k_j different integers from the interval $[1,n]$, which specify the identifiers of strategic spots that are subsequent vertices of the polygon. All the given polygons will be simple (i.e. without self-intersections), and their vertices are specified in clockwise order. The sum of all the numbers k_j does not exceed 1,000,000.

Output

Your program should write m lines to the output. The j-th line should contain one integer, being the number of strategic spots in the interior of the polygon described in the j-th query.

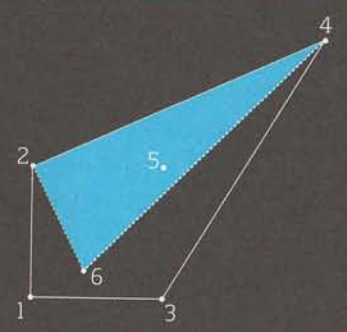
Example

For the input data:

```
6 4
0 0
0 5
5 0
11 10
5 5
2 1
4 1 2 4 3
4 1 2 5 3
3 6 2 4
3 1 2 6
```

the correct result is:

```
2
1
1
0
```



Explanation of the example: The points in the figure depict strategic spots, while the numbers next to the circles are their identifiers. The figure shows the areas from the first (continuous lines) and third (dashed lines, filled in blue) queries.

/ Solution

From the area of a polygon to the number of points inside

Assume that we are given a polygon $P = A_1A_2A_3...A_k$, where the points A_i are listed in clockwise order. There are several methods of evaluating the area of P, based on the idea of expressing this value as the sum of areas of simpler figures. Probably the best known one uses trapezoids: we traverse the consecutive sides of the polygon in a clockwise order and, for each side, add or subtract the area of the trapezoid below it, depending on whether we are increasing or decreasing the x-coordinate along that side. It is not hard to verify that all the parts of the plane are added in this sum exactly the same number of times as subtracted, except for the polygon P itself, which is added exactly one more time than it's subtracted (see figure 1a).

In the solution of our problem it will be more convenient to use a similar technique, but based on triangles with a common vertex (figure 1b). We consecutively add or subtract the areas of triangles OA_iA_{i+1} (the indices behave in a cyclic manner), depending on whether the orientation of the triangle is clockwise or counterclockwise (O is the center of the coordinate system). In this way we express $\mu(A_1A_2A_3...A_k)$ as $\sum_{i=1}^{k}\mu(OA_iA_{i+1})$, where μ is the signed area of a polygon, i.e. taken with a positive sign if its orientation is clockwise, and with a negative sign otherwise. As we know how to find the areas of triangles easily, we can compute the area of P in linear time. Note that the point O can be chosen arbitrarily; from now on we take $O = A_1$.

In our problem we would like to count the number of given points from S (strategic objects) inside the polygon instead of its area. However, we could simply treat the function μ in our formulas not as the area of a figure, but the number of points in the interior—everything should still work! In this way, if we are able to construct a data structure that can answer queries of the form, "How many points from S are inside the triangle with vertices $X, Y, Z \in S$?" then by computing the same sum as before we should get the number of points inside the polygon. Are we sure? Well, in order to prove it formally, we need to show that every point from S lying inside the polygon is counted exactly once in the sum, whereas all the points outside the polygon, including its vertices, are counted zero times.

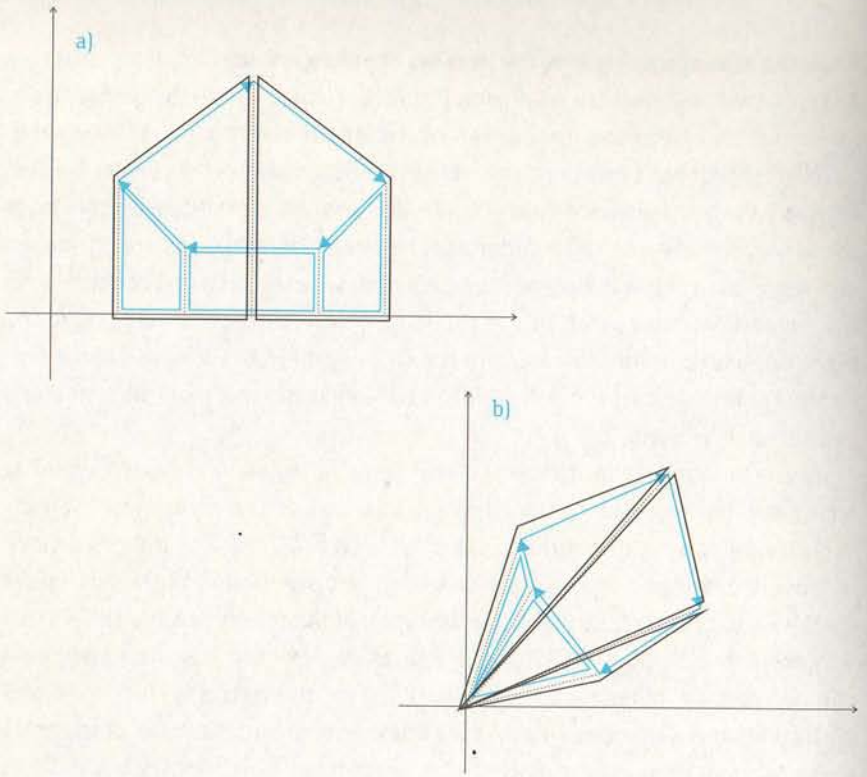


Figure 1: *Computing the area of a polygon using trapezoids and triangles.*

Let us consider some $X \in S$ and look at the ray $\overrightarrow{OX}$. Imagine a stickman that starts a journey at X and moves along the ray in the opposite direction to O. If X is inside P, then the stickman begins inside P, and after getting sufficiently far he will always be outside P, so he has to leave P exactly one time more than he enters P. In the sum, the point X is counted positively for every side through which the stickman leaves P, and negatively for every side through which he enters P. This means that each X lying inside P will be added exactly one time more than it will be subtracted (see figure 2a). Let us note that the stickman will never travel via a vertex of P; this would mean that three points from S—O, X and the considered vertex—were collinear.

Similarly, it can be proved that every X lying outside P is added and subtracted the same number of times as the stickman starts outside P and also

finishes outside P (see figure 2c). However, there is a problem for X being a vertex of P, i.e. $X = A_i$ for some i. If, directly after leaving X, the stickman finds himself outside P, then the point X is indeed added and subtracted the same number of times (see figure 2b). However, if we in fact enter P at X, then the point X is counted exactly once (see figure 2d), when it should be counted zero times. Fortunately, this is not a big problem: every such vertex that is counted one extra time can easily be recognized from the mutual positions of incident sides. This means that, by using an additional scan through the perimeter of P, we can compute this additional contribution to the sum.

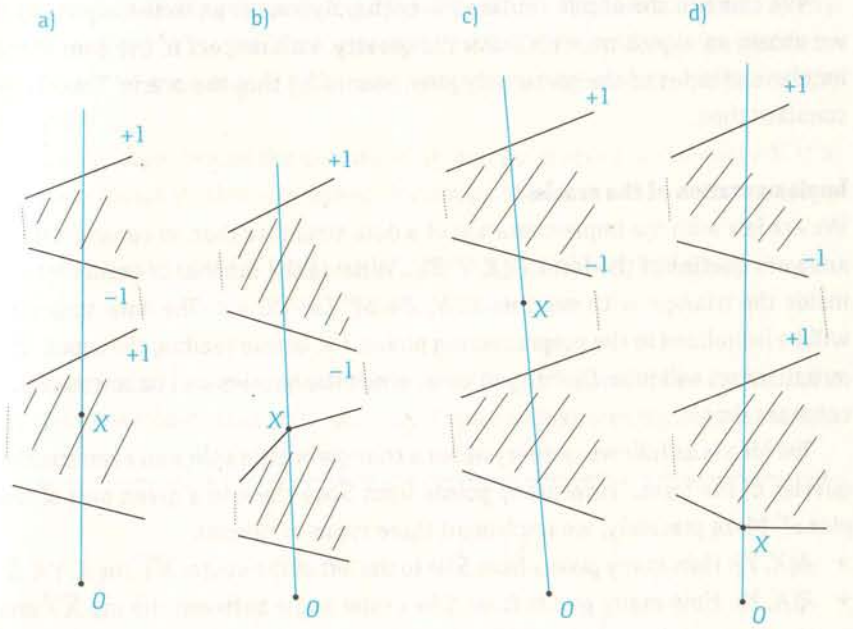


Figure 2: *Cases of alignment of point X with respect to P.*

Using a subroutine, which for any three points $X, Y, Z \in S$ that form a triangle oriented clockwise computes the number of points from S lying in its interior, we can easily implement the whole algorithm. This subroutine will be called the *oracle*.

```
Algorithm ArmyTraining(P = A_1A_2A_3...A_k, oracle T)
    result := 0
    for i := 2 to k − 1 do
        if triangle A_1A_iA_{i+1} is oriented clockwise then
            result := result + T(A_1, A_i, A_{i+1})
        else
            result := result − T(A_1, A_{i+1}, A_i)
    for i := 2 to k do
        if ray A_1A_i enters the interior of P in A_i then
            result := result − 1
    return result
```

We can run the above routine for each polygon given in the input. Thus we obtain an algorithm with linear complexity with respect to the sum of the numbers of sides of the given polygons, assuming that the oracle T works in constant time.

Implementation of the oracle

We are left with the implementation of a data structure that, in constant time, answers queries of the form $T(X, Y, Z)$: "What is the number of points from S inside the triangle with vertices $X, Y, Z \in S$?" Let $|S| = n$. The data structure will be initialized in the preprocessing phase, i.e. before reading the input. The initialization will take $O(n^2 \log n)$ time, while the queries will be answered in constant time.

The idea is as follows: a query about a triangle will be split into even simpler queries of the form, "How many points from S are there in a given part of the plane?" More precisely, we implement three types of queries:

- $A(X, Y)$: How many points from S lie to the left of the vector $\overrightarrow{XY}$, for $X, Y \in S$?
- $R(X, Y)$: How many points from S lie in the angle between the ray $\overrightarrow{XY}$ and the horizontal ray with origin in X and directed to the right, for $X, Y \in S$?
- $B(X, Y, Z)$: How many points from S lie in the angle between the rays $\overrightarrow{XZ}$ and $\overrightarrow{XY}$, for $X, Y, Z \in S$?

In the queries R and B we consider the angle between the rays constructed by clockwise traversal from the first to the second. In the results we omit the points lying on the selected lines and/or rays. Also, we assume that $X \neq Y \neq Z \neq X$. See figure 3 for an illustration.

Let us note that the query $B(X, Y, Z)$ can be trivially implemented via the query $R(X, Y)$. We have $B(X, Y, Z) = R(X, Z) - R(X, Y) - 1$ if $R(X, Z) > R(X, Y)$, and $B(X, Y, Z) = (n - 2) + R(X, Z) - R(X, Y)$ otherwise. The answers to queries A and R we will simply precompute in two-dimensional arrays of size $O(n^2)$.

For each point $X \in S$ we sort all the remaining points $Y \in S$ with respect to the angle between the ray $\overrightarrow{XY}$ and the horizontal ray with X as the origin, directed to the right. Let us denote the list of sorted $S \setminus \{X\}$ as L_X. The values $R(X, Y)$ can be computed very easily: if Y has index i in the list L_X (indices begin with zero), then $R(X,Y) = i$. Therefore, we can simply scan through the list L_X and complete one row of the array of values of R.

Memorizing the answers for A queries is only slightly more difficult. We traverse the list L_X using a pointer Y, while maintaining also a second pointer Y' that points to the farthest (in the angular distance) point on L_X that still counts in the query $A(X, Y)$, or on Y if no such point exists. When incrementing the pointer Y we simply push Y' as far as possible. Now we can observe that $A(X, Y) = B(X,Y,Y') + 1$ if $Y \neq Y'$ and $A(X,Y) = 0$ otherwise, so we can memorize the answer for the query $A(X,Y)$ using precomputed answers for queries $R(X,Y)$ and $R(X,Y')$.

Having implemented the queries A, B, we can proceed to the query T. If X, Y, Z form a triangle that is oriented clockwise, then

$$2(n-3) + T(X, Y, Z) = B(X, Z, Y) + B(Z, Y, X) + B(Y, X, Z) + \\ + A(X, Y) + A(Y, Z) + A(Z, X).$$

This formula can easily be verified by considering how many times each point in S is counted on both sides of the equality. Figure 4 illustrates the right-hand side.

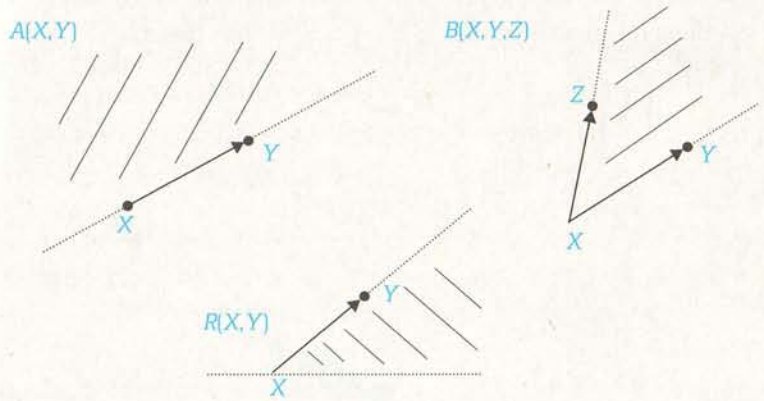


Figure 3: *Queries A, R, B.*

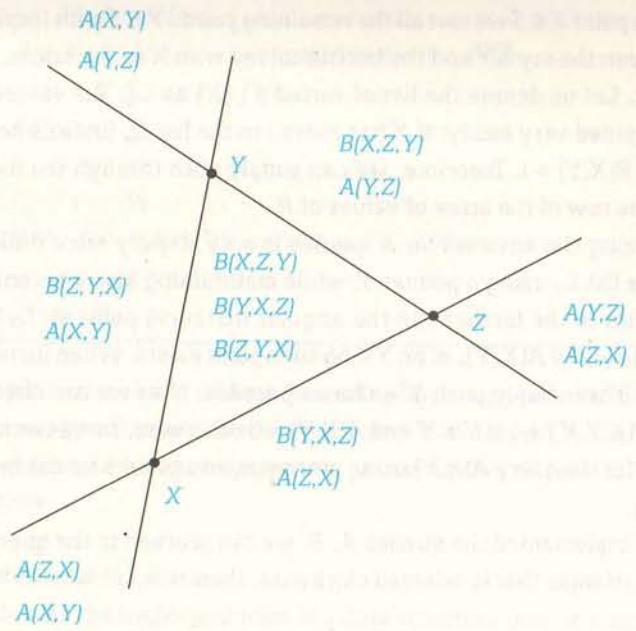
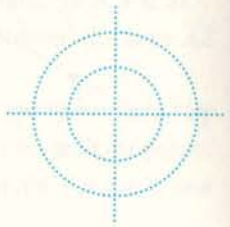
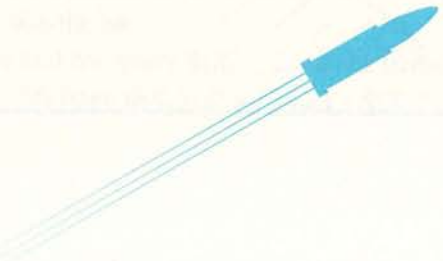


Figure 4: *Expressing query T in terms of queries A, B.*

In conclusion, we can memorize the answers for queries A, R in time complexity $O(n^2 \log n)$, since we sort all but one points from S n times in total. After that, we can answer the queries $T(X, Y, Z)$ in constant time. If we use this implementation of the oracle, then the presented algorithm runs in time $O(n^2 \log n + K)$, where K is the total number of sides of polygons given in the input.





Riddle

Contest: Algorithmic Engagements 2010
Task authors: Marek Cygan, Marcin Pilipczuk, Michał Pilipczuk, Jakub Wojtaszczyk
Memory: 128 MB
http://main.edu.pl/en/archive/pa/2010/zag

The evil sorcerer Voldebyte has imprisoned the brave knight Bytter the Bold in his tower. As is his custom, Voldebyte has promised to free Bytter if he can solve one of his (as yet unsolved) riddles. Bytter, unfortunately, has managed to kill Voldebyte's pet dragon, and came close to killing Voldebyte himself, so Voldebyte has decided that a really tough riddle is in order. This is the riddle posed by Voldebyte to Bytter:

Byteland is divided into k counties, in which there are n towns altogether. Additionally, some pairs of towns are connected by bidirectional roads. I would like to choose a town in each county to be its capital in such a way that, for each road, at least one of its endpoints is a capital town. Is this possible?

Help poor Bytter to save himself and solve the riddle for him.

Input

In the first line of the input there are three integers: n ($1 \le n \le 1{,}000{,}000$), specifying the number of towns in Voldebyte's riddle; m ($0 \le m \le 1{,}000{,}000$), specifying the number of roads; and k ($1 \le k \le n$), specifying the number of counties. The towns are numbered from 1 to n.

In the next m lines there are m pairs of integers a_i, b_i ($1 \le a_i, b_i \le n$, $a_i \ne b_i$); the i-th pair specifies a road connecting towns a_i and b_i. No pair of towns is connected by more than one road.

In the next k lines there are descriptions of subsequent counties. The j-th line begins with an integer w_j ($1 \le w_j \le n$), specifying the number of towns in the j-th county. Then w_j integers are given, specifying the (distinct) numbers of the towns in the j-th county. The sum of all the numbers w_j is equal to n.

Output

If the solution to the riddle is negative, your program should write a single line containing the word NIE (Polish for *no*) to the output.

Otherwise, your program should write two lines. The first should contain the word TAK (Polish for *yes*); the second should describe the solution. The second line should contain exactly k integers. The i-th of these integers should be the number of the town that should be selected as the capital of the i-th county.

If there are multiple correct solutions, your program may output any of them.

Example

For the input data:

```
6 5 2
1 2
3 1
1 4
5 2
6 2
3 3 4 2
3 1 6 5
```

the correct result is:

```
TAK
2 1
```

whereas for the input:

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

the correct output is:

```
NIE
```


/ Solution

Evil Voldebyte gave Bytter a map of Byteland, on which the towns are divided between counties. Bytter's task is to choose exactly one town from each county as its capital in such a way that, for every road connecting two towns, at least one of these towns is chosen as a capital of its county.

Let us consider how to model this situation. For the i-th town of Byteland we introduce a Boolean variable x_i that is true if this town is a capital of its county, and is false otherwise. In this language it is not hard to express the constraints given by the sorcerer:

- If the towns with indices i and j are connected by a road, then at least one of them must be the capital of its county; this constraint has the form $x_i \vee x_j$.
- In each county at most one town can be the capital. Equivalently, for every pair of towns within the same county, these towns cannot both be the capital at the same time. Therefore, we introduce a constraint $\neg x_i \vee \neg x_j$ for every pair of distinct indices i, j, such that the i-th town and the j-th town are in the same county.

Let us remark that such a representation of the problem allows for choosing none of the towns as the capital of some county. But if Bytter finds such a solution, then he will not spoil anything by simply making any town the capital in this county.

Hence, Voldebyte's riddle has a solution if and only if there exists such an evaluation of variables x_i that satisfies every introduced constraint. Let us write a Boolean formula

$$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_l$$

where C_j are the introduced constraints which henceforth will be called *clauses*. Each clause is a disjunction of two *literals*, i.e. occurrences of a variable or of its negation. This means that φ is a *conjunction of disjunctions of literals*, while every clause contains at most two literals. We say that the formula is in 2-CNF: it is in *conjunctive normal form* and 2 denotes the bound on the number of literals in a clause.

Checking satisfiability—i.e. whether there is an evaluation of variables that makes the formula true—for formulas in CNF is one of the oldest problems in theoretical computer science. For 3-CNF the problem is already NP-complete,

which means that we do not expect it to be solvable in polynomial time; however, for formulas in 2-CNF there exists a clever algorithm that checks satisfiability in linear time with respect to the number of variables and clauses. This algorithm is described in most of the classical books on graph algorithms, as well as on several algorithmic web pages.

We just need to check the size of the formula obtained by translating the description of the riddle. We have one variable per town. The number of clauses of the first type, i.e. those that encode roads between towns, is exactly the same as the number of roads. Unfortunately, in order to encode the constraints of the second type, we introduce one clause per pair of towns within the same county. The number of such pairs can be quadratic, so the algorithm will run in $O(n^2)$ time complexity. With one million towns in the input, this solution could not achieve a full score; we need something faster.

Speedup via compactification

The main problem in the presented approach is an overly expensive method of expressing the information that, in each county, at most one town can be a capital. Maybe we could encode it using only a linear number of clauses? In fact, it is possible; however, we will pay for a smaller number of clauses by increasing the number of variables.

The idea of the compactification is as follows: we will introduce new variables that encode whole groups of towns. Let us consider one county, containing towns with indices $i_1 < i_2 < \cdots < i_p$. For every prefix and every suffix of this sequence we introduce a variable, which intuitively says, "in this prefix (suffix) there is a capital." Formally, for every $j = 1,2,\ldots,p$ we introduce variables y_{i_j} and z_{i_j}, which will satisfy the following properties:

- if among towns $i_1,i_2,\ldots,i_j$ there is a capital, then y_{i_j} is true;
- if among towns $i_j,i_{j+1},\ldots,i_p$ there is capital, then z_{i_j} is true.

Conditions imposed on variables y_{i_j} and z_{i_j} can easily be encoded via clauses containing two variables. The key observation is as follows: a capital is inside some prefix (suffix) if and only if it is on its end or in the prefix (suffix) shorter by one town. Therefore,

- for each $j = 1,2,\ldots,p$ we introduce a clause $x_{i_j} \Rightarrow y_{i_j}$;
- for each $j = 1,2,\ldots,p-1$ we introduce a clause $y_{i_j} \Rightarrow y_{i_{j+1}}$;
- for each $j = 1,2,\ldots,p$ we introduce a clause $x_{i_j} \Rightarrow z_{i_j}$;
- for each $j = 1,2,\ldots,p-1$ we introduce a clause $z_{i_{j+1}} \Rightarrow z_{i_j}$.

Let us note that in the formula we can use implication, as the clause $a \Rightarrow b$ is equivalent to $\neg a \vee b$.

Observe that in this way we have encoded only one direction of implication; it could happen that the variable y_{i_j} is true even though in the corresponding prefix there is no capital. We only know that if the capital is actually there then y_{i_j} is true. Fortunately, this turns out not to be a problem.

We are left with encoding the uniqueness of capitals using variables y_{i_j} and z_{i_j}. For each $j = 1,2,\ldots,p-1$ we introduce a clause $\neg y_{i_j} \vee \neg z_{i_{j+1}}$. If there were two capitals in a county, say with indices i_j and $i_{j'}$ for $j < j'$, then the introduced clause would not be satisfied for the index j, as both y_{i_j} and $z_{i_{j+1}}$ would need to be true. On the other hand, if a county has at most one capital, say in the town with index i_j, then we can set $y_{i_{j'}}$ to true for all $j' \geq j$ and to false for $j' < j$, and symmetrically for variables $z_{i_{j'}}$. Thus we satisfy all the introduced clauses, as for every partition of the sequence $i_1,i_2,\ldots,i_p$ into a prefix and a suffix, in at least one of the parts the index i_j does not occur.

We perform this construction for every county and add clauses encoding the roads just as in the previous section. Let us check the size of the resulting formula:

- we have three variables per town—in total $3n$ variables;
- for each road we introduced one clause—in total m clauses;
- for each county we introduced $5p - 3$ clauses, where p is the number of towns within the county—in total, we have constructed at most $5n$ clauses.

Hence, the constructed formula φ has $3n$ variables and at most $5n + m$ clauses, so the linear-time algorithm for checking satisfiability of formulas in 2-CNF should be able to produce the answer within the time limit.

And what is the problem really about?

Let us view the map of Byteland as an undirected graph (V,E), in which every town is represented by a vertex and two vertices are adjacent if and only if the corresponding towns are connected by a road. Moreover, the vertices of the first county will be colored with color 1, second with color 2, etc. Now Voldebyte asks for a subset of vertices S such that each edge has at least one end in S. Such a set is called a *vertex cover* of the graph. In the sorcerer's riddle one has to find a vertex cover that contains exactly one vertex in each color. This is called the *colored* version of the problem.

In general, the problem of determining whether a given graph admits a vertex cover of size k is NP-complete; this means that it seems implausible that it can be solved in polynomial time. The complexity of computing a small vertex cover is also considered in the description of the problem *Guilds* in this book; a simple branching algorithm with running time $O(2^k(n + m))$ is described there. We

now present how to obtain an algorithm with slightly worse time complexity, $O(e^k(n+m))$, which moreover will be randomized, but which will also be a lot cooler! The main idea is to use the colored version of the problem, solvable in linear time, as a subroutine.

The described algorithm will be *randomized with one-sided error*. This means that, during the computation, the program can make decisions based on random coin tosses. If the graph does not admit a vertex cover of size at most k, then the algorithm must always give a negative answer. However, if a vertex cover exists, then the algorithm can either find it or make an error and return a negative answer, but the probability of such a mistake must be at most $\frac{1}{2}$. Observe that by making 20 such tests consecutively, we can reduce the probability of a mistake to a minute chance.

Our algorithm will perform a simple test multiple times in hopes that, at some moment, it will be able to "pinpoint" the solution. In each test we draw at random a coloring of vertices with k colors by independently assigning each vertex each color with the same probability; then, using the linear-time algorithm for the colored version, we check whether there exists a vertex cover that uses one vertex of each color. Therefore, in each test we count on the event that all the vertices from the solution will be assigned different colors, so the algorithm for the colored version will be able to extract the solution quickly.

Algorithm RandomizedVertexCover($(V,E),k$)
  for $t := 1$ **to** $\lceil e^k \rceil$ **do**
    choose a coloring of V with k colors at random
    invoke the algorithm for the colored version
    if the algorithm for the colored version has found a solution **then**
      return the solution
  return *no*

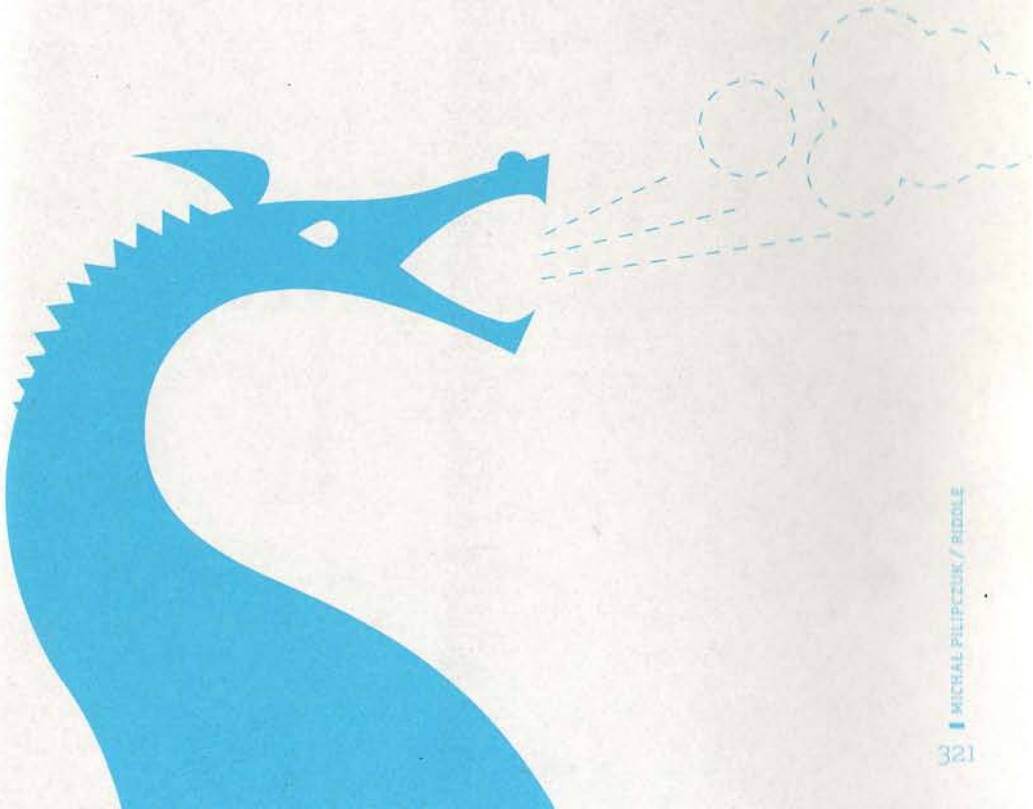
Clearly, if the algorithm has found a solution then it is a feasible vertex cover of size k. It remains to prove that if the graph admits some vertex cover of size k, then the algorithm finds an example solution with probability at least $\frac{1}{2}$. Let us fix some solution S and consider the probability that all the vertices from S are assigned different colors. It is easy to see that this probability is equal to $\frac{k!}{k^k}$, as S can be colored using distinct colors in $k!$ ways, while there are k^k colorings of S in total. Now we use the inequality $\frac{k!}{k^k} \geq \frac{1}{e^k}$ to infer that the probability that, during a single test, the solution is colored distinctly is at least $\frac{1}{e^k}$. Hence, with

at least this probability the subroutine for the colored version will be able to find a solution (maybe a different one from S) in linear time.

Now let us bound the probability that in each of the $\lceil e^k \rceil$ tests the algorithm does not find a solution. The probability of a mistake in a single test is at most $1 - e^{-k}$, so, as the tests are independent, the computation will be erroneous with a probability of at most $(1 - e^{-k})^{e^k}$. We can use the inequality $1 - x \leq e^{-x}$, which holds for all $x \in [0,1]$, to infer that the probability of an error is at most $e^{-e^{-k}e^k} = e^{-1} < \frac{1}{2}$.

Finally, let us remark that the algorithm invokes a subroutine running in $O(n+m)$ time at most $\lceil e^k \rceil$ times, so the total time complexity of the algorithm is $O(e^k(n+m))$.

The technique of randomly coloring a graph in order to make a solution distinguishable in polynomial time is called *color coding*, and was introduced by Alon, Yuster and Zwick in the early nineties. Nowadays, it is a classical method in the design of parameterized and exact exponential time algorithms. Let us also remark that the algorithm can be *derandomized* at the cost of worse running time. More precisely, based on the described method, one can construct a program that does not use randomness but that has a constant larger than e in the complexity bound.



JAKUB RADOSZEWSKI

Jakub Radoszewski is a Ph.D. student of computer science at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. His research interests focus on text algorithms and combinatorics. He is the chairman of the jury of the Polish Olympiad in Informatics and Algorithmic Engagements, an editor of booklets of the Polish Olympiad in Informatics, and an editor of a Polish popular science monthly, *Delta*. In 2011 he was the chairman of the jury of the Central European Olympiad in Informatics and the Polish Collegiate Programming Contest. As a student, together with his team, he reached second place in the Polish Collegiate Programming Contest and the ACM-ICPC Central European Regional Contest. He is a multiple semifinalist of TopCoder Open and finalist of Google Code Jam and Facebook Hacker Cup. He enjoys playing tennis, table tennis and badminton.

/ Fuel

Contest: Algorithmic Engagements 2011
Task author: Jacek Tomasiewicz
Memory: 128 MB
http://main.edu.pl/en/archive/pa/2011/pal

In the good old days, all n towns in Byteland were connected by a dense network of two-way roads. The king of Byteland decided to reduce the number of roads, and asked his Chief Computer Scientist for advice. Now, as a result of this decision, Byteland has only $n - 1$ two-way roads connecting pairs of towns in such a way that there is exactly one route between any two towns in Byteland. All the roads are of the same length.

Equipped with a car whose tank holds exactly enough fuel to drive on m roads in Byteland, Byteasar has decided to organize a trip that maximizes the number of visited towns. He can start his trip in any of the towns in Byteland, and, similarly, his trip can end anywhere—not necessarily back at the starting town. In maximizing the number of visited towns, Byteasar is allowed to drive multiple times on the same road, in the same or the reverse direction. Your task is to help Byteasar by finding the maximum number of towns that can be visited on one full tank of fuel.

Input

The first line of the input contains two integers, n and m ($2 \le n \le 500{,}000$, $1 \le m \le 200{,}000{,}000$), where n is the number of towns in Byteland (each numbered uniquely from $\{1,\ldots,n\}$), and m is the number of roads that can be traveled on one tank of fuel.

The next $n - 1$ lines describe Byteland's road network. Each of these lines contains two integers, a and b ($1 \le a, b \le n$), indicating that towns a and b are connected with a two-way road.

Output

The output consists of one line containing exactly one integer: the maximum number of towns that can be visited on one full tank of fuel.

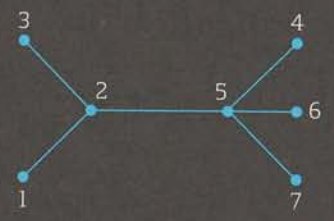
Example

For the input data:

7 6
1 2
2 3
2 5
5 6
5 7
4 5

the correct result is:

5



Explanation of the example: Byteasar can visit a maximum of five different towns. There are several different routes that visit five towns for this example input, including $4 \to 5 \to 7 \to 5 \to 6 \to 5 \to 2$ and $3 \to 2 \to 1 \to 2 \to 5 \to 6 \to 5$.

Solution

I do not like hard problems. I am a person who likes to get his hands dirty at work, so instead of thinking about a solution for hours I much prefer to alternate phases of thinking and coding when tackling a problem. If I keep analyzing a problem for a long period of time, and I feel that I have not made any real progress towards the solution, I simply get bored with it and find something more productive to do.

I certainly do not like trivial problems, though; I learn nothing from solving such problems. Also, when solving such a problem, I feel like a monkey trained to hit a keyboard as fast as possible. This type of problem includes tasks that can be solved with a straightforward application of a well-known technique or a classical algorithm, even if this technique or algorithm requires writing many lines of very smart code.

I do not care for problems whose solutions require only abstract ideas: ideas that may or may not occur in one's mind arbitrarily. I maintain that, for contestants with similar knowledge and abilities, the time it takes to solve such a task depends mostly on luck. This introduces a considerable amount of randomness to a competition, especially if the idea seems intuitively correct, but is really hard to prove—though note that such a proof is not required during a competition.

It appears that I do like my own problems, however! A few years ago I used to devise lots of new problems. Unfortunately, nowadays I do not have enough time to look around and constantly search for interesting new ideas on which to base problems.

Regardless of what I have just said, the problems that I have chosen to describe in this book are *not* ones that I wrote myself.

What I consider a perfect problem is neither too easy nor too difficult. Solving it should require creativity—so that a contestant can enjoy the magic moment of a discovery—and also the ability to apply (in a smart way) important classical tools—known techniques and algorithmic bricks. Such a problem combines pure intellectual activity with a challenging task for an algorithms engineer. If, additionally, the formulation of the problem contains a neat story with a natural background, I will certainly remember such a problem for a long period of time.

For the purposes of this book, I have chosen two problems that, in my opinion, satisfy all the above requirements and, moreover, have made an especially good impression on me. Let us start with the easier one, the task *Fuel*.

Solution

The story from the task statement translates into the following graph-theory problem: we would like to visit nodes of a given tree in the most efficient way. More precisely, we are looking for a walk of a given length that visits as many different nodes as possible. By a *walk* we mean a path that may contain multiple occurrences of the same node. Sometimes such a sequence of nodes is also called just a path; however, we will stick to the term walk to avoid confusion with a simple path in which no node repetitions are allowed.

As in the task statement, by n and m we denote the number of nodes in the tree and the given length of the walk. As soon as we start thinking about the problem, we can immediately make several basic observations. First of all, from our point of view the best type of walk is a path without repeating nodes; that is, the aforementioned simple path. If m is small enough, there is a chance that we will be able to find a simple path of length m. In order to describe all such clearly optimal walks, it is sufficient to find the longest simple path in the tree, the so-called *diameter* of the tree. Denote by d the length of the diameter; if $m \le d$, the correct result is $m+1$.

Thus we have identified the most optimistic case. Now let us consider the problem from a different perspective. What if we are interested in a really long walk? If m is large enough with respect to n, we may expect to find a walk that visits all the nodes of the tree. This, surely, takes place if $m \ge 2 \cdot (n-1)$—in this case the desired walk can be constructed by a depth-first search (DFS) traversal of the tree in which each edge is visited exactly twice: once in each direction.

Let us try to state a precise condition for the existence of a walk that visits all the nodes of a tree. Consider the shortest such walk, P. Assume that it starts in the node u and ends in the node v. If we added a path from v to u to P, we would obtain a walk that visits all the nodes of the tree and ends at its starting node. As we have already observed, the shortest such walk is the DFS traversal of the tree, of length $2 \cdot (n-1)$. Hence, the shortest possible walk P can be obtained as a DFS traversal starting in such a node u and ending in such a node v that u and v are endpoints of a diameter of the tree. The length of this walk is $2 \cdot (n-1) - d$.

We have already shown how to solve the problem if either $m \le d$ or $m \ge 2 \cdot (n-1) - d$. What can we do in the remaining cases? Let us consider a walk of length $m \in (d, 2 \cdot (n-1) - d)$ that leads through as many different nodes of the tree as possible (see figure 1). Denote the number of nodes visited in this walk by k. It turns out that this value is the key to the construction of the final algorithm.

Indeed, let us remove all the nodes of the tree that are not present in the walk. What remains is a subtree of the initial tree. Our walk visits all the nodes of this subtree, so, due to the previous considerations, it satisfies $m \ge 2 \cdot (k-1) - d'$, where d' is the diameter of this subtree. Hence, $k \le (m + d' + 2)/2$. In order to maximize the value of k we choose a subtree of maximal diameter, that is, a subtree containing a diameter of the initial tree. Then, $d' = d$. We obtain a walk that visits exactly $\lfloor (m+d+2)/2 \rfloor$ tree nodes.

Using the parameters n, m and d one can state the final algorithm as follows:

- If $m \le d$, return $m+1$.
- If $m \ge 2 \cdot (n-1) - d$, return n.
- If none of these cases holds, return $\lfloor (m+d+2)/2 \rfloor$.

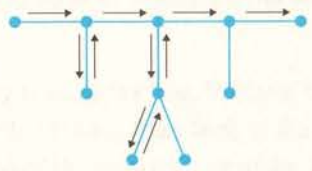


Figure 1: An example of an optimal walk for $n = 10$, $d = 4$, $m = 10$.

What is still missing is an efficient algorithm for finding the diameter of a tree. Here I could actually conclude the whole description, since a linear time algorithm solving this problem is folklore knowledge, or I could leave it as an exercise for the reader. However, since the most straightforward approach may often not be the simplest one, in the remaining part of the description I will show my favorite algorithm for this problem, which I learned only after several years of dealing with algorithms.

Diameter of a tree

What is the simplest algorithm for finding the diameter of a tree? I am quite sure that it is as follows:

- Choose an arbitrary node of the tree; denote it by u.
- Walk from u to the furthest node of the tree; denote that node as v.
- Walk from v to the furthest node of the tree; denote that node as w.
- The path from v to w is the diameter of the tree.

The only nontrivial operation here is walking to the furthest node of a tree. This operation is very easy to implement if we know the distances from u (v respectively) to all the nodes of the tree. These distances can be computed using any tree search algorithm; e.g. a BFS or DFS algorithm.

It is much more interesting to know why this algorithm actually works. To answer this question it is convenient to present the tree as "hanging by its diameter". Let x and y denote the endpoints of a diameter (initially unknown to us). Imagine that the tree edges are strings of unit length and the tree nodes are knobs. Let us hold the node x in one hand and the node y in the other hand, and pull the strings in the diameter connecting x and y. How long are the strings that hang from the respective nodes of the diameter? Obviously, no strings can hang from x and y. From a direct neighbor of x or y only a string of length 1 can hang, since otherwise the tree would contain a path longer than its diameter. Similarly, from neighbors of neighbors of x and y, only strings of length 2 can hang, and so on (see figure 2).

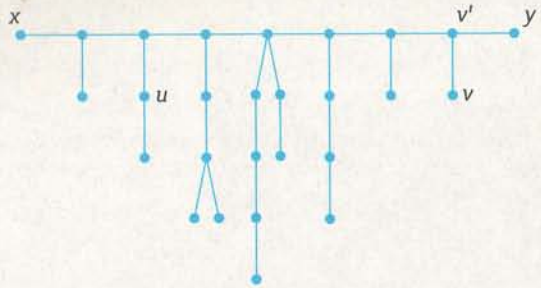


Figure 2: *A tree hanging by its diameter. A sample location of the nodes u and v from the algorithm is presented.*

Now, u is an arbitrary node of the tree. Without loss of generality, assume that u is located not further from x than from y. Clearly, the node v from the algorithm must be located on the opposite side of the diameter, either at the end of one of the hanging strings or in the node y. In the former case, the distance from v to the corresponding node on the diameter, v', must be equal to the distance from v' to y (e.g. in figure 2, v could not be the node located one unit left from the node denoted as v in this figure). In both cases, the path from v to x is a diameter of the tree, hence the path from v to the furthest node w is also a diameter of the tree. This proves the correctness of the algorithm.

/ Ploughing

Contest: 13th Polish Olympiad in Informatics
Task author: Marcin Kubica
Memory: 64 MB
http://main.edu.pl/en/archive/oi/13/ork

Byteasar, the farmer, wants to plough his rectangular field. He can begin by ploughing a slice from any of the field's edges, then he can plough a slice from any unploughed area's edges, and so on, until the whole field is ploughed. After the ploughing of every successive slice, the still-unploughed field has a rectangular shape. Each slice has a span of 1, and the length and width of the field are the integers n and m.

Unfortunately, Byteasar has only one puny and frail nag (horse) at his disposal for the ploughing. Once the nag starts to plough a slice, it will not stop until the slice is completely ploughed. However, if the slice is too much for the nag to bear, it will die of exhaustion, so Byteasar has to be careful. After every ploughed slice, the nag can rest and regain its strength. The difficulty of certain parts of the field varies, but Byteasar is a good farmer and knows his field well; hence, he knows the difficulty of ploughing every part.

Let us divide the field into $m \times n$ unitary squares called *tiles*. We identify them by their coordinates (i, j), for $1 \le i \le m$ and $1 \le j \le n$. Each tile has its own ploughing-difficulty—a nonnegative integer. Let $t_{i,j}$ denote the difficulty of ploughing the tile whose coordinates are (i, j). For every slice, the sum of ploughing-difficulties of the tiles constituting the slice may not exceed a certain constant k, lest the nag dies.

A difficult task awaits Byteasar: before ploughing each successive slice, he has to decide which edge of the field he will plough, so that the nag will not die. On the other hand, he would like to plough as few slices as possible.

Task

Write a program that:

- reads the numbers k, m and n from the input file, as well as the ploughing-difficulty coefficients,
- determines the best way to plough Byteasar's field,
- writes the result to the output file.

Input

There are three positive integers in the first line of the input file, separated by single spaces: k, m and n ($1 \le k \le 200{,}000{,}000$, $1 \le m, n \le 2000$). In the following n lines there are the ploughing-difficulty coefficients. The $(j+1)$-th line contains the coefficients $t_{1,j}, t_{2,j}, \ldots, t_{m,j}$ ($0 \le t_{i,j} \le 100{,}000$), separated by single spaces.

Output

Your program should write one integer to the output file: the minimum number of slices required to plough the field while satisfying the given conditions. Since we care for animals, we guarantee that the field can be ploughed according to the above rules. But remember, saving the nag is up to you!

Example

For the input data:

```
12 6 4
6 0 4 8 0 5
0 4 5 4 6 0
0 5 6 5 6 0
5 4 0 0 5 4
```

the correct result is:

```
8
```

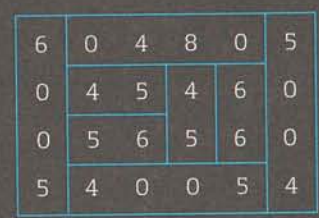


Figure: *The illustration shows the optimal ploughing pattern for the field in the example.*

/ Solution

This problem is much harder than the task *Fuel*. I consider it the most elegant problem from the Polish Olympiad in Informatics. It satisfies all the requirements from the introduction to the previous description, but that is not the only reason why I have chosen this task.

For several years now I have been looking at competition problems not only from a contestant's point of view but also, even more often, from a jury member's perspective. Problems that allow the contestants to be differentiated are invaluable in all "olympiad in informatics"-style contests. An important category of such problems is those that admit several natural solutions with different time complexities. When solving the task *Ploughing*, one arrives at solutions with time complexity $O(N^5)$, $O(N^4)$, $O(N^3)$ and finally $O(N^2)$. The respective solutions require an increasing amount of conceptual effort: applying dynamic programming, a greedy approach and a very simple amortized cost analysis—a so-called sliding window technique (or, alternatively, a smarter greedy algorithm). What an awesome combination!

I remember the day when Marcin Kubica gave me this task. He came up with it while mowing his lawn. As I was walking in a park and thinking about the possible solutions, each time I was able to reduce the time complexity by a factor of N I was more and more amazed with this task. At that time I was a rookie member of the Jury of the Polish Olympiad in Informatics. It is thanks to problems like this one that I have decided to participate in the organization of the Olympiad for a longer period of time.

Dynamic programming

Let us proceed with the problem analysis. We are given a rectangular board divided into $m \times n$ cells, each cell filled with a nonnegative integer. We would like to remove all the cells of the board. The only operation we may perform is to remove a slice of the board from any of its edges—that is, a peripheral row or column. Such an operation is allowed only if the sum of numbers written in the cells to be removed does not exceed a specified parameter k. Our task is to remove all the cells of the board with the minimum number of operations. We assume that the columns are numbered from 1 to m from left to right (the first coordinate) and the rows from 1 to n from top to bottom (the second coordinate). We state that $N = n + m$.

To obtain the first solution, let us make the following basic observation: after an arbitrary sequence of operations the board still has the shape of a rectangle. More precisely, it is a subrectangle of the initial board. This allows us to apply a standard approach based on dynamic programming.

A state in our dynamic programming is a quartet of numbers (i_1,j_1,i_2,j_2) representing a rectangle whose upper-left corner is (i_1,j_1) and lower-right corner is (i_2,j_2). In a single operation we can transition from such a state to each of the states: (i_1+1,j_1,i_2,j_2), (i_1,j_1+1,i_2,j_2), (i_1,j_1,i_2-1,j_2) and (i_1,j_1,i_2,j_2-1), provided that the corresponding operation can be performed; that is, that the sum of elements in the i_1-th column, the j_1-th row, the i_2-th column and the j_2-th row of the rectangle, respectively, does not exceed k. The result associated with a state is the minimum number of operations required to remove all the cells of the rectangle. The final state of the dynamic programming is $(1,1,m,n)$, and the initial states are all empty rectangles, e.g. rectangles for which $i_1 = i_2 + 1$ and $j_1 \le j_2$ and symmetric cases. In this classical approach we have $O(N^4)$ states and at most four transitions from a single state.

A straightforward implementation of this approach yields $O(N^5)$ time complexity, since $O(N)$ time is required to check which transitions are valid. To obtain $O(N^4)$ time complexity, we need to be able to compute in $O(1)$ time the sum of numbers written in the part of the i-th column located between the j_1-th and the j_2-th row, and, similarly, the sum of the part of the j-th row located between the i_1-th and the i_2-th column. Thus we are left with a one-dimensional problem of finding the sum of given contiguous subsequences in a number sequence $a_1,\ldots,a_p$ (that represents the numbers in a given row or column of the board). A simple solution to this problem uses the so-called sequence of partial sums, i.e. sums of the form $s_i = a_1 + \cdots + a_i$. Once we compute the partial sums, each $a_i + \cdots + a_j$ can be computed in constant time with a single subtraction $s_j - s_{i-1}$. This enables us to answer the requested queries in $O(1)$ time after $O(N^2)$ time preprocessing of partial sums in all rows and columns of the board. Thus we obtain a solution with $O(N^4)$ time complexity.

Greedy approach

Clearly our first solutions did not attempt to gain a serious insight into the problem. When we hold a hammer, everything around us looks like a nail. In order to make an improvement, we need to look for some properties of the solution. Let us start by asking how big the result can be.

Each operation reduces by 1 exactly one dimension of the rectangle. We finish when one of the dimensions drops to 0. If we are lucky enough, we will end up

performing $\min(n, m)$ operations. On the other hand, after at most $n + m - 1$ operations the board will already be empty. Thus we can conclude that the range of possible results is not so wide.

As a byproduct of the above analysis we obtain the useful corollary that one of the dimensions must eventually be reduced to 0, and this can be obtained only using slices that reduce this dimension by 1. Hence, in the whole solution we remove either m vertical slices or n horizontal slices. From now on, we assume that in the solution we remove m vertical slices (as the alternative case is symmetric). Under this assumption, the goal is to remove as few horizontal slices as possible (see figure 1). Let us just note that, having made this assumption, we are no longer sure that a valid solution can still be obtained.

So far our analysis has been only of a structural nature. The time has come to introduce a bit of greediness to the solution. We know that sooner or later we will remove m vertical slices. Consequently, if it is possible to remove a vertical slice at some point, it pays off to perform this action instantly. Unfortunately, there are cases when no vertical slice can be removed from the board. Then, a horizontal slice must be removed. But which of the two possibilities should we choose? Apparently no obvious greedy criterion can be stated here. Therefore we will apply a "less greedy" approach—we will check both possible options.

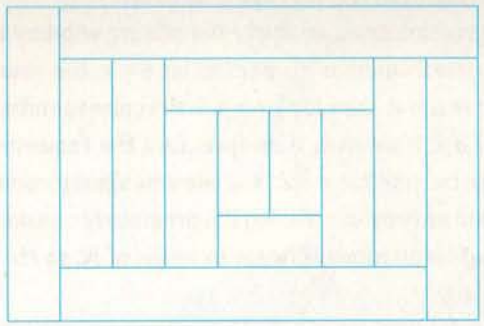


Figure 1: We remove m vertical slices.

Assume that, apart from the m vertical slices, in our solution we remove a upper and b lower horizontal slices ($a + b < n$). With this assumption in mind, we can afford a completely greedy approach. Each time we are able to remove a vertical slice, one of the a upper horizontal slices or one of the b lower horizontal slices, we can simply perform this action, since we would have done it sooner or later anyway. Note that this greedy routine exits with a success if and only if there exists a solution that removes m vertical slices, at most a upper horizontal slices and at most b lower horizontal slices.

Let us analyze the complexity of this solution. First of all, after the parameters a and b have been selected, the slice removal can be simulated really efficiently. Indeed, in each step we choose just one of four possible slices and remove it. Checking whether a slice is valid is performed in constant time using the partial-sums trick described in the previous section. In conclusion, the cost of a single simulation is $O(N)$, so the time complexity of the whole solution is $O(N^3)$.

Final tweaks

It is easy to see where the previous solution performs in a suboptimal way. It is doubtful whether the whole simulation can be performed in time sublinear in N; however, the number of simulations performed seems rather excessive. Indeed, there are a few ways of improving this process.

The first idea is to apply a binary search. We set the value of the parameter a and look for the smallest such b that the problem can be solved by removing at most a upper horizontal slices, at most b lower horizontal slices and, obviously,

m vertical slices. Finding the minimal value of b requires performing $O(\log N)$ simulations, each of which works in $O(N)$ time. We obtain an $O(N^2 \log N)$ time algorithm, which is not too far from being optimal.

To get an optimal solution, we apply the sliding window technique. Let us note that if, for a given value of the parameter $a = x$, the smallest value of the second parameter is $b = y$, then for $a = x + 1$ the corresponding value of b does not exceed y. For $a = 1$ we can, therefore, find the requested b by applying an exhaustive search, and for $a = 2, 3, \ldots$ we can simply consider consecutive values of b obtained only by decreasing the previously considered values. Here the total number of simulations is linear in terms of N, so the whole algorithm runs in $O(N^2)$ time.

Same time complexity can be obtained with a smarter implementation of the initial greedy approach. It suffices to introduce just one of the auxiliary parameters, e.g. the parameter a. Then, according to the greedy approach, in the simulation we can freely remove any vertical slices and at most a upper horizontal slices. Only if we cannot remove any slice of the aforementioned form do we choose to remove a lower horizontal slice. This modified simulation procedure still runs in $O(N)$ time, so the whole algorithm is quadratic in terms of N. And this solution to the task *Ploughing* is the one I admire the most.

Exercise

We conclude with an exercise for the reader. Note that the majority of the solutions described here do not work correctly if the board is allowed to contain negative numbers. Which solutions fall into this category? What are the corresponding counterexamples? How should this phenomenon be explained?

WOJCIECH RYTTER

Wojciech Rytter is a professor of computer science and the initiator of the algorithms section at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. He is an author and co-author of over 100 publications and several textbooks on algorithmics.

His scientific interests are centered on the design and analysis of algorithms and data structures for parallel computing, discrete mathematics, graph theory and text algorithms as well as formal languages and automata theory. Wojciech Rytter is also interested in algorithms for heavily compressed objects that do not require decompression.

Wojciech Rytter is one of the co-founders of the Polish Olympiad in Informatics launched in 1993, and has been involved in this event since its inception. He is the author of a few dozen olympic programming tasks.

/ Canoes

Contest: 4th Polish Olympiad in Informatics
Task author: Wojciech Rytter
Memory: 32 MB
http://main.edu.pl/en/archive/oi/4/kaj

We are organizing a canoe tour. Canoes can be hired at the harbor. Canoes are all alike. A canoe can take at most two persons. The sum of weights of these persons may not exceed the fixed maximum weight. We want to pay as little as possible, so we should try to place all the participants of our tour in the minimum number of canoes.

Task

Write a program that:

- reads from the input the maximum weight of a canoe crew, the number of participants of the tour and the weight of each participant,
- computes the minimum number of canoes that should be hired to accommodate all the participants of the tour according to the rules,
- writes the result to the output.

Input

In the first line of the input there is one integer w ($80 \le w \le 200$), specifying the maximum weight of a canoe crew. In the second line there is one integer n ($1 \le n \le 30{,}000$), specifying the number of participants of the tour.

Each of the following n lines contains one integer within the range $[5,w]$, representing the weights of the participants.

Output

In the first line of the output one integer should be written: the minimum number of canoes that should be hired.

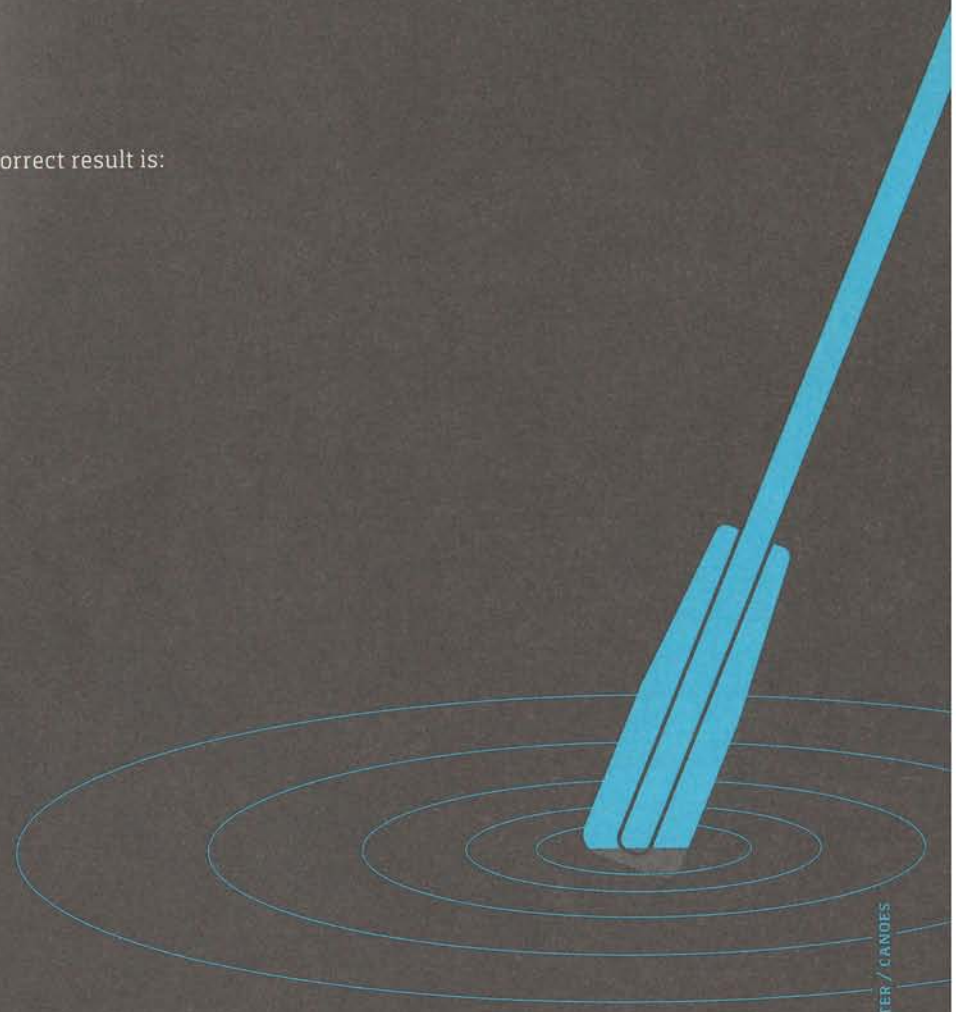
Example

For the input data:

```
100
9
90
20
20
30
50
60
70
80
90
```

the correct result is:

```
6
```

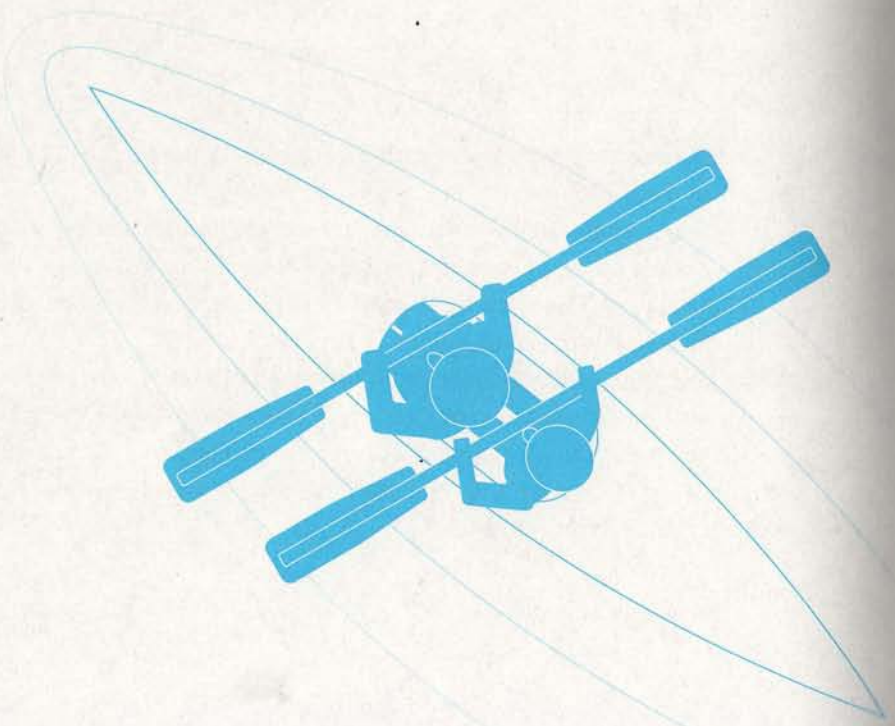


/ Solution

This is a simple problem that appeared in the Polish Olympiad in Informatics. A fast algorithmic solution is almost trivial, but not uninteresting. The algorithm is designed using a straightforward greedy approach:

Algorithm Greedy (informally)
  while there is still any person on the bank of the river **do**
    the heaviest of them is loaded into the next canoe with the person
    of the smallest weight, if they fit together;
    otherwise, only the heaviest one is loaded
  return the number of canoes used

In fact, we can place the heaviest person with anybody with whom he fits in a single canoe.



Why is such a strategy correct? Consider the following fact, which implies correctness immediately.

Fact. There exists an optimal solution (configuration) in which the heaviest person, say Mr. H, is in the same canoe with the lightest person, say Mr. L, or alone.

Proof. There are two possibilities in a fixed optimal configuration:

1. Mr. H sails by himself: obviously we can take this configuration and our fact is self-evidently true.
2. Mr. H sails with some person X. Then, X can exchange places with Mr. L, and we get a configuration that satisfies our requirements. □

The greedy algorithm can be written in the pseudocode as follows:

Algorithm Greedy
  {Input: array $weight[0..n-1]$, capacity of the canoe MAX}
  {Assumptions: $weight[i] \le MAX$ for $i = 0, 1, \ldots, n-1$}
  sort the array of weights in non-increasing order
  $i := 0$; $j := n-1$
  while $i \le j$ **do**
    if $weight[i] + weight[j] \le MAX$ **then**
      $j := j - 1$
    $i := i + 1$
  return i

The efficiency depends on the sorting method; we have chosen bucket sort. Thus, the whole algorithm has linear time complexity.

The simplicity of the solution disappears if the problem is slightly changed:

1. How should we count the number of optimal configurations?
2. What happens if we allow three people into the same canoe? How, then, do we efficiently compute the minimum number of canoes?

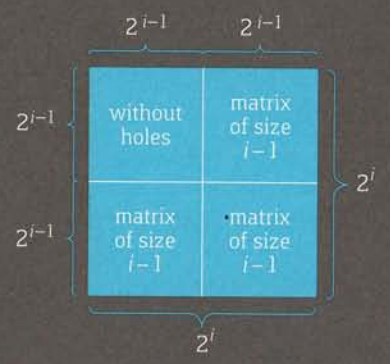
We suggest finding a polynomial time algorithm for the case where the weight array is a permutation of the numbers $1, \ldots, n$ (MAX can be arbitrary). Observe that, in this case and for (at most) two-person canoes, there is exactly one solution if $n \le MAX \le n+1$, but for $MAX = 2n$ there are exponentially many solutions. What about $MAX = n+2$?

/ Painter's *Studio*

Contest: 5th Polish Olympiad in Informatics
Task author: Wojciech Rytter
Memory: 32 MB
http://main.edu.pl/en/archive/oi/5/mal

The Painter's Studio is preparing the mass production of paintings. Paintings are going to be made with the aid of square matrices of various sizes. A matrix of size i consists of 2^i rows and 2^i columns. There are holes at the intersections of some rows and columns. A matrix of size 0 has one hole.

For $i > 0$, a matrix of size i is composed of four squares of size $2^{i-1} \times 2^{i-1}$. Look at the following figure:



Both squares on the right side and the bottom-left square are matrices of size $i-1$. The top-left square has no holes. Pictures are constructed in the following way. First, we decide on the values of three nonnegative integers n, x, y. Next, we take two matrices of size n, place one of them onto the other and shift the upper one by x columns right and y rows up. We place such a pattern on a white canvas and cover the common part of the matrices with yellow paint. In this way, we get yellow stains on the canvas in the places where the holes in both matrices align.

Example
Consider two matrices of size 2.

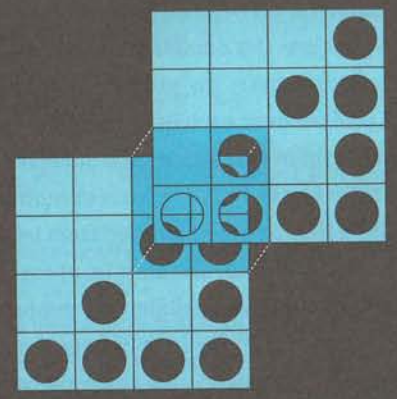


Figure: The upper matrix was shifted by 2 columns right and 2 rows up. There are three places where holes align.

Task
Write a program that:
- reads from the input the sizes of two matrices and the numbers of columns and rows by which the upper matrix should be shifted,
- computes the number of yellow stains on the canvas,
- writes the result to the output.

Input
There is one integer n ($0 \le n \le 100$) in the first line of the input, specifying the size of the matrices used for the production of paintings. In the second line there is one integer x, and in the third line one integer y, where $0 \le x, y \le 2^n$. The integers x and y are the numbers of columns and rows by which the upper matrix should be shifted.

Output
The first and only line of the output should contain the number of stains on the canvas.

Example

For the input data:	the correct result is:
2 2 2	3

/ Solution

For this task we use the general conceptual technique of *dynamic programming*. The solution of a given problem is a combination (in our case, sum of values) of several subproblems. In this case, the problem is to compute the size of the overlap of two squares shifted over each other: let $N((x,y),n)$ be the *number* of common holes in the overlap of both matrices, the second matrix shifted by (x,y) over the first one. Denote by M_n the $2^n \times 2^n$ matrix containing holes as described in the task statement. The columns are numbered from left to right and the rows from bottom to top, starting from zero. The shift (x,y) corresponds in the matrix to the position (x,y) called the *active position*. The subproblems correspond to overlaps of matrices of shape $2^i \times 2^i$, for $i \le n$, where the number i is called the *rank* of the problem (and of the matrix). The main point is the dependence of overlaps of rank i on a constant number of overlaps of rank $i-1$.

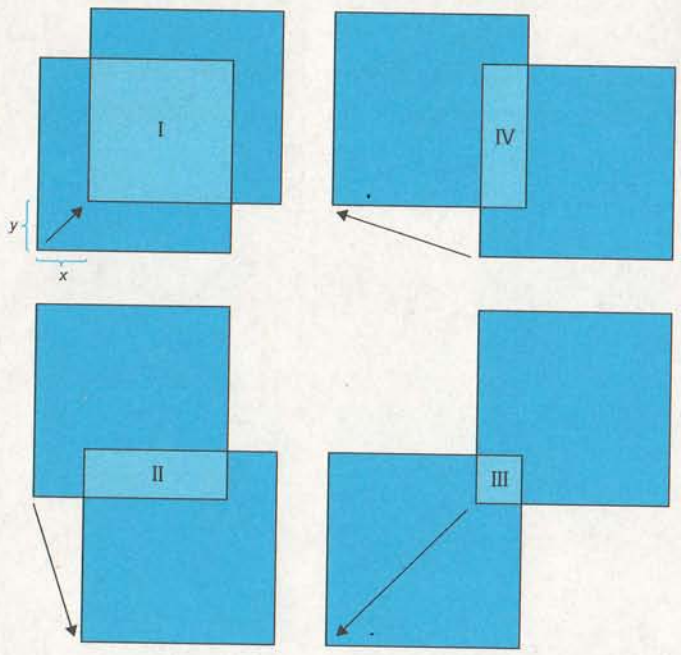


Figure 1: *The common parts of type I, II, III and IV of two matrices shifted one over the other. The arrows correspond to shifts $(x,y)+\beta_k(i)$, for $k=1,2,3,4$. $W_k(i)$ is the number of common holes in the corresponding common part.*

Define:

$$\beta_1(i)=(0,0),\quad \beta_2(i)=(0,-2^i),\quad \beta_3(i)=(-2^i,-2^i),\quad \beta_4(i)=(-2^i,0);$$

$$W_k(i)=N((x,y)+\beta_k(i),i),\quad \text{for } k=1,2,3,4.$$

Instead of computing a single value we want to compute the 4-tuple

$$W(i)=[W_1(i),W_2(i),W_3(i),W_4(i)].$$

In other words, we compute the number of common holes in the overlapping parts I, II, III, IV shown in figure 1.

The main idea is to express *in a simple way* the 4-tuple $W(i)$ in terms of a 4-tuple $W(i-1)$, corresponding to the matrix of the smaller rank. The 4-tuple $W(i)$ is located with respect to the active position (x,y), while the new 4-tuple is considered with respect to a new active position, $(x',y')=reduce(x,y)$. The coordinates of the position (x',y') are the coordinates of (x,y) with respect to the quarter of the matrix M_i in which (x,y) is located. The operation *reduce* is computable in constant time.

The value of the new 4-tuple corresponding to the matrix of rank $i-1$ is similarly determined by the location of the active position (x',y') in this matrix, as before.

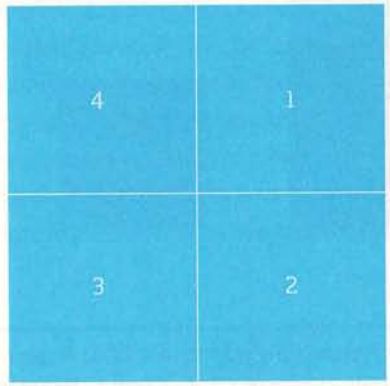


Figure 2: *Numbering of quarters.*

Hence, knowing four numbers related to objects of rank $i-1$, we want to compute in constant time four numbers related to objects of rank i, and can write

$$W(i) = \Phi(W(i-1), x, y).$$

The crucial point is to find the dependency Φ of $W(i)$ with respect to $W(i-1)$. The form of Φ depends crucially on the number of the quarter of the matrix in which the position (x,y) is located. Hence we switch to a quartile representation of positions in matrices. Let us number the quarters of the matrix as in figure 2.

In this representation, the operation of reducing (x,y) to (x',y') has an especially simple form. We describe how to compute the quartile representation $\lambda = address(x,y)$.

Initially, λ is an empty sequence. The initial matrix is of rank n, so its shape is $2^n \times 2^n$. Its active position belongs to one of four quarters. Let us write the number p of this quarter at the beginning of the address: in other words, $\lambda := p \cdot \lambda$. Let us remove the other quarters of the matrix. We get a new matrix of a smaller rank, and the active position is now located in a possibly different way with respect to the beginning of this new matrix; its new coordinates are (x',y'). We repeat this process until we arrive at a 1×1 matrix (see figure 3).

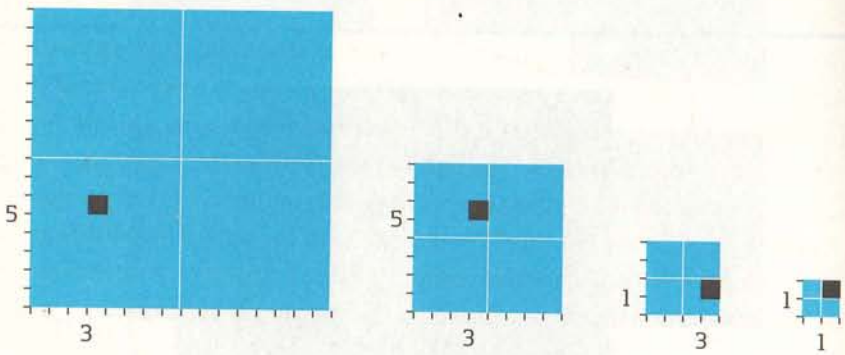


Figure 3: *The sequence of consecutive positions of* $(3,5)$ *in quarters of decreasing ranks. This position is successively in quarters* 3, 4, 2 *and* 1. *Hence* $address(3,5) = (1,2,4,3)$ *in the matrix of rank* 4. *The addresses (quartile representations) of successive positions result from cutting off the last digit.*

The sequence of consecutive numbers of quarters written in reverse order is the quartile address of (x,y) denoted by $address(x,y)$.

We now show the dependence of $W(i)$ on $W(i-1)$ for the case when (x,y) is in the first quarter of the matrix of rank i. In this case, the position $(x',y') = reduce(x,y)$ is of the form:

$$(x-2^{i-1}, y-2^{i-1}).$$

Let us express the numbers $W_1(i)$, $W_2(i)$, $W_3(i)$, $W_4(i)$ in terms of $W_1(i-1)$, $W_2(i-1)$, $W_3(i-1)$, $W_4(i-1)$. It is convenient to represent this dependence by 4×4 tables, called dependency tables. We introduce our first dependency table:

$$A_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix},$$

$$\begin{aligned}
W_1(i) &= a_{11} \cdot W_1(i-1) + a_{12} \cdot W_2(i-1) + a_{13} \cdot W_3(i-1) + a_{14} \cdot W_4(i-1), \\
W_2(i) &= a_{21} \cdot W_1(i-1) + a_{22} \cdot W_2(i-1) + a_{23} \cdot W_3(i-1) + a_{24} \cdot W_4(i-1), \\
W_3(i) &= a_{31} \cdot W_1(i-1) + a_{32} \cdot W_2(i-1) + a_{33} \cdot W_3(i-1) + a_{34} \cdot W_4(i-1), \\
W_4(i) &= a_{41} \cdot W_1(i-1) + a_{42} \cdot W_2(i-1) + a_{43} \cdot W_3(i-1) + a_{44} \cdot W_4(i-1).
\end{aligned}$$

In other words, the rows of the table are the multiplicities of components of $W(i-1)$. The k-th row of A_j is the sequence of dependency coefficients of $W_k(i)$ from $(W_1(i-1), W_2(i-1), W_3(i-1), W_4(i-1))$, assuming that (x,y) is in the j-th quarter of the current matrix. The index of the table (in this case it is 1, since we are considering the first quarter) corresponds to the number of the quarter containing the active position. We can see here the useful properties of the quartile system. When the active position is in the first quarter, it is easy to see, by considering the mutual placements of the quarters of both matrices, that:

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example 1. We show how the first row of A_3 is computed (by hand). In this case, (x, y) is in the third quarter. The common part consists of three smaller parts of type I, and one of each of the types II, III and IV (in the sense of overlaps of matrices of rank $i-1$); see figure 4. Hence the first row of A_3 is $[3,1,1,1]$. Other rows of A_3 are computed by shifting the overlapping matrix of rank i in three remaining directions over the first matrix of rank i, as in figure 1.

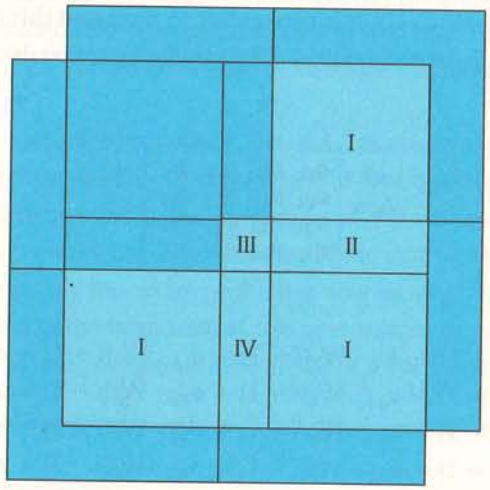


Figure 4: *The first row of A_3 is $[3,1,1,1]$; this means that if the left-bottom corner of the upper matrix is in the first quarter, then we have three smaller overlaps of type I, and one of each of the types II, III and IV, corresponding to matrices of smaller rank. (The parts which are not labeled contain no holes, and can be ignored.)*

In a similar way, we construct *by hand* the tables A_2, A_3, A_4, thus obtaining

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 3 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 3 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad A_4 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Example 2. Assume (x, y) is in the fourth quarter. According to A_4, the computation of $(W_1(i), W_2(i), W_3(i), W_4(i))$ consists in executing:

$$W_1(i) := W_1(i-1) + W_4(i-1); \quad W_2(i) := W_1(i-1) + 3 \cdot W_2(i-1) + W_3(i-1);$$

$$W_3(i) := W_3(i-1) + W_4(i-1); \quad W_4(i) := 0.$$

To simplify the notation, we introduce the operation $\otimes$, which transforms one 4-tuple into another using a table:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \otimes \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix},$$

$$\begin{aligned} r_1 &= v_1 \cdot a_{11} + v_2 \cdot a_{12} + v_3 \cdot a_{13} + v_4 \cdot a_{14}, \\ r_2 &= v_1 \cdot a_{21} + v_2 \cdot a_{22} + v_3 \cdot a_{23} + v_4 \cdot a_{24}, \\ r_3 &= v_1 \cdot a_{31} + v_2 \cdot a_{32} + v_3 \cdot a_{33} + v_4 \cdot a_{34}, \\ r_4 &= v_1 \cdot a_{41} + v_2 \cdot a_{42} + v_3 \cdot a_{43} + v_4 \cdot a_{44}. \end{aligned}$$

Using all these notations, the pseudocode of our algorithm is as follows.

```
Algorithm Compute-N((x,y),n)
    compute the quartile address of (x,y)
    W(0) := [1,0,0,0]
    for i := 1 to n do
        k := the i-th digit of the quartile address
        W(i) := A_k ⊗ W(i-1)
    return W_1(n)
```

We set the initial values:

$$W_1(0) = 1 \quad \text{and} \quad W_2(0) = W_3(0) = W_4(0) = 0.$$

It is easy to check that they generate correct values of $W(1)$ for all four possible shifts (x,y) in the 2×2 matrix. We have:

$$address(x,y) = 1 \Rightarrow W_1 = [1,0,1,0], \quad address(x,y) = 2 \Rightarrow W_1 = [1,0,0,1].$$

$$address(x,y) = 3 \Rightarrow W_1 = [3,0,0,0], \quad address(x,y) = 4 \Rightarrow W_1 = [1,1,0,0].$$

Example 3. For the 16×16 matrix and the shift $(3,5)$ (compare with figure 3), the correct answer is 14. The algorithm computes it as follows:

initialization:	$address(3,5) = (1,2,4,3)$
iteration 1:	$A_1 \otimes [1,0,0,0] = [1,0,1,0]$
iteration 2:	$A_2 \otimes [1,0,1,0] = [1,0,1,2]$
iteration 3:	$A_4 \otimes [1,0,1,2] = [3,2,3,0]$
iteration 4:	$A_3 \otimes [3,2,3,0] = [14,2,3,0]$
termination:	$output = 14$

KRZYSZTOF STENCEL

Krzysztof Stencel is an associate professor at the Institute of Informatics, University of Warsaw. He is interested in databases and software engineering. He is an author of numerous publications, textbooks and translations.

As a contestant, he has taken part in algorithmic competitions only once. In 1989 he represented Poland during the first International Olympiad in Informatics. However, he has been much more involved in such competitions as a judge and organizer. Between 1993 and 2008 he was the chairman of the jury of the Polish Olympiad in Informatics. He also chaired the jury of several international contests: Central European Olympiad in Informatics (1997, 2004), Baltic Olympiad in Informatics (2000, 2001, 2008) and International Olympiad in Informatics (2005). In 2011 he was the President of the Central European Olympiad in Informatics.

In 2008 Krzysztof Stencel began solving programming tasks on the famous contest server of University of Valladolid for fun. Between September 19, 2011, and the time of writing this text, he has been the leader in the rankings, with over 3000 algorithmic problems solved.

/ Triangles

Contest: 1st Polish Olympiad in Informatics
Task author: Piotr Chrząstowski
Memory: 16 MB
http://main.edu.pl/en/archive/oi/1/tro

A finite, at least three-element set A of rational-length line segments is given. We want to examine whether a triangle may be created from every three segments in A.

Input

The input contains a sequence of lengths of all line segments in the set A. Every segment length is a rational number written in the form a/b, where numerator and denominator are positive integers not greater than 9999. Consecutive lengths in this sequence are separated by single spaces.

Output

Your program should give one of the following answers: TAK (Polish for *yes*) if a triangle may be formed from every three line segments of A, or NIE (Polish for *no*) if some three line segments of A cannot be used to create a triangle.

Examples

For the input data:

13/10 1/2 6/5 11/6 9/7 3/5 9/7 13/10 9/5 8/5

the correct result is:

NIE

whereas for the input:

1/2 3/5 2/3 4/7 1/1 4/6

the correct output is:

TAK

Explanation of the first example: One cannot create a triangle from line segments of lengths 6/5, 3/5 and 9/5.

/ Triangles 2

Contest: 2nd Polish Olympiad in Informatics
Task author: Piotr Chrząstowski
Memory: 32 MB
http://main.edu.pl/en/archive/oi/2/tro

In a finite sequence of positive integers representing the lengths of line segments, we want to find three numbers such that a triangle can be created from segments of such lengths.

Task

Write a program that examines whether there exist three line segments—whose lengths are supplied in the input—from which one can create a triangle. If so, the program should write the lengths of those three segments in the output. If no such set of three segments exists, the program should write one word, NIE (Polish for *no*), in the output file.

If there exist multiple such sets of three line segments from which triangles may be created, then your program should find and write only one of them (the choice of set is arbitrary).

Input

In the input there is a finite sequence of at least three positive integers not greater than 1,000,000,000, terminated by the number 0. Each number is written on a separate line. These integers are lengths of line segments, and the number 0 denotes the end of the data.

Output

In the output there should be either one word, NIE, or three line segment lengths chosen from the input, from which one can create a triangle. The lengths are separated by single spaces.

Examples

For the input data:

105
325
55
12555
1700
0

the correct result is:

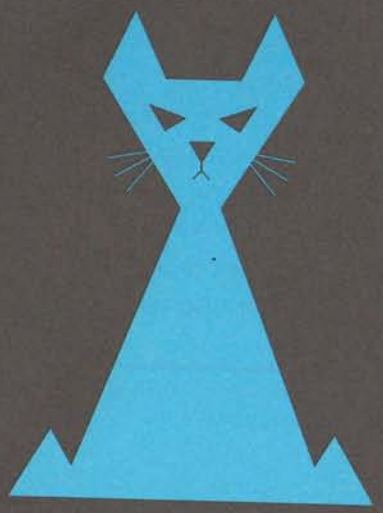
NIE

whereas for the input:

250
1
105
150
325
99999
73
0

the correct output is:

250 105 150



Solution

Triangles

The lengths of the sides of each proper triangle on the Euclidean plane satisfy so-called *triangle inequality*. If the sides of a triangle have lengths a, b, c and c is not smaller than the others, then it is true that

$$a + b > c. \tag{1}$$

This inequality is so common that it is hard to imagine basing upon it an interesting problem for programming contests. Surprise! Piotr Chrząstowski managed to prepare two interesting problems based solely on triangle inequality for the 1st and 2nd Polish Olympiads in Informatics. Personally, I am not a problem setter at all; I have set problems neither for the national Olympiad nor for Algorithmic Engagements. For years I have been a judge of programming contests, however. Nowadays, my hobby is to solve problems on the famous server of the University of Valladolid. I have solved a significant number of them; yet, my ability to set new problems is nil. This makes my esteem for problem setters very high: especially for those like Piotr Chrząstowski, who can make a nice contest problem out of almost nothing. They are like brilliant poets who can find subject matter under every stone, while we ordinary people can only trample the stones underfoot.

The input to the task from the 1st Olympiad is a sequence of segments of rational lengths. The problem consists in checking whether, from each three of those segments, one can form a proper triangle in Euclidean geometry. Note that—fortunately—only in the 1st Olympiad were contestants requested to check the correctness of the input data. If they had been faulty, the program would have printed the message NONSENSE. It was a nonsense indeed, and in subsequent Olympiads a contestant could assume that input data were correct. This rule is currently commonly adopted in all programming contests all over the world*.

Let us return to our problem. We do not have to check each trio of segments; it would be too expensive. Instead, we initially compute the length of the longest segment c and the lengths of the two shortest segments a and b ($a \le b$). Then we check whether they satisfy the inequality (1). If not, we have a trio of segments

* That is why the task statement presented in this book has been modified to remove the requirement to check the validity of the input data.

that do not form a proper triangle. However, if this inequality holds, all other trios also form correct triangles. In order to prove it, we pick three arbitrary segments a_1, b_1, c_1 from the input sequence. We will assume that $a_1 \le b_1 \le c_1$. Since a and b are the shortest and c is the longest, then $a \le a_1$, $b \le b_1$ and also $c \ge c_1$. Thus, using (1) we know that:

$$a_1 + b_1 \ge a + b > c \ge c_1.$$

Therefore, the trio a_1, b_1, c_1 satisfies the triangle inequality. Since the choice of a_1, b_1, c_1 was arbitrary, we know that all trios of segments form proper triangles.

The algorithm to solve this problem is now obvious: read the data; find the two shortest segments and the longest segment; then check if they satisfy (1) and output the answer. This means linear time and constant space consumption.

Can you believe that no contestant presented a program that scored the maximum number of points? Why not? Because we have forgotten the mouse traps in the input data. They are rational numbers with numerators and denominators up to 9999. "So what? Only 10^4?" you ask. When comparing two rational numbers with such denominators, they can reach only 10^8 and comfortably fit in a 32-bit signed integer. This is what our diabolic problem setter counted on! Eventually, we have to compute the answer. Assume that the smallest numbers are $\frac{n_1}{d_1}$ and $\frac{n_2}{d_2}$, while the largest is $\frac{n_3}{d_3}$. We have to check whether

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} > \frac{n_3}{d_3}.$$

After we multiply both sides by $d_1 d_2 d_3$, we get

$$n_1 d_2 d_3 + n_2 d_1 d_3 > n_3 d_1 d_2.$$

Input data are from the range 1 to 9999. Thus we can obtain numbers of magnitude 10^{12}, which is too large for 32 bits!

Twenty years ago, tricking the contestants in this way was even easier! A machine word usually contained only two bytes, and the same was also true of the standard integer type. Thus, it was very easy to overlook the magnitude of the values to be processed. Tests were chosen in such a way that precision errors in the conversion of rational to floating point (*real* or *double*) caused wrong answers. No student had yet heard about epsilons needed to compare floating

point values. This is no surprise. Even today, many programmers still believe in the perfect precision of such data types.

As you already know, there was no happy ending to this story. One student identified the problem with data types and implemented his own arithmetic to cope with it. Unfortunately, instead of computing maximum and minimum values on the fly, he initially loaded all the data into main memory. This could not succeed, since the input data and memory limits were set up to prevent such solutions from getting full points. Thus, this contestant also had to settle for less than the maximum score. So who's laughing at triangle inequality now?

Triangles Reloaded

The triangular problem from the 2nd Olympiad in Informatics is dual. Given a sequence of lengths of segments, you are to find a trio of them that forms a proper triangle; otherwise, your program has to declare the nonexistence of such a trio. Again, we do not have to check all possible trios. Let us sort our segments in increasing order. A triangle can be formed if, in the sorted sequence $a_1, \ldots, a_n$, there exist three neighboring items a_k, a_{k+1}, a_{k+2} such that:

$$a_k + a_{k+1} > a_{k+2}. \tag{2}$$

Indeed, assume there are three elements of our ordered sequence a_r, a_s, a_t such that $a_r + a_s > a_t$ and $r < s < t$. Note that $a_{s+1} \le a_t$ and $a_r \le a_{s-1}$. Thus:

$$a_{s-1} + a_s \ge a_r + a_s > a_t \ge a_{s+1}.$$

Therefore it holds that $a_{s-1} + a_s > a_{s+1}$, so a_{s-1}, a_s, a_{s+1} are three neighboring items of the sequence that satisfy the triangle inequality. We have proven that if the input sequence contains three segments that form a triangle, then in the sorted input sequence there will be three appropriate neighboring segments. In order to solve the problem, we sort the input data and check whether the inequality (2) holds for any k. The algorithm consumes $O(n \log n)$ time and linear space.

It remains only to collect our pay check, but to our surprise they say that they will pay only for constant time complexity (of course, in this case the space complexity must also be constant). Have we missed something? Yes! The range of input data. The lengths of the segments are positive integers up to 10^9. If there are no trios forming proper triangles, in the sorted input sequence each item (except the first two) must be no smaller than the sum of its two predecessors. (Who called up Fibonacci's spirit again? Anyone still not convinced about *The Da Vinci*

Code?) Among all possible input sequences that have no trio forming a triangle, the least ascending is the following:

1, 1, 2, 3, 5, 8, 13, 21, 34,...

This sequence increases very quickly, and its 45th element is already greater than 10^9: its value is 1,134,903,170. Thus it is enough to read only 45 first input numbers, sort them and find a trio that satisfies (2) or announce its nonexistence. The time needed by this algorithm is thus constant; we do not have to read all the input data.

The contestants of the 2nd Olympiad solved this problem much better than the previous one. Almost half of them got full marks. Nearly everyone noted that the length of the sequence without any triangle was limited; the estimations varied from 43 to 16,000. That lower limit was in fact too small, and was caused by neglecting initial repeating values. Of course, the input data contained a test case where the 44th item closed a triangle and made such a solution lose points. That upper limit was too big, since the time limit disallowed the reading of so much data. Remember that it was almost twenty years ago, and computers were significantly slower than today. Nowadays, in many problems you can still find a warning like: the input data is huge, use faster input/output methods.

Triangles Revolutions

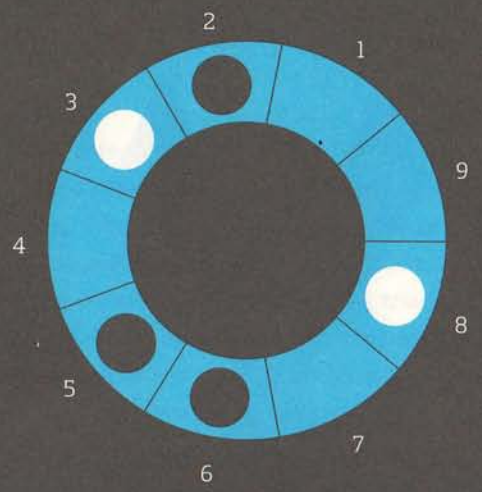
In 2009 three former contestants of the Polish Olympiad in Informatics founded Codility. This company offers automatic interviewing facilities for businesses that hire programmers. One of the Codility tests is almost the same as the second triangle problem discussed above; the only difference is that the answer is only YES or NO, depending on whether one can form a triangle from input items or not. The Codility servers now have 3,500 solutions to this problem, submitted by IT professionals from 85 countries. Only 4.5% of them have noticed that it is sufficient to check only trios of neighbouring items in the sorted sequence. About 54.5% of candidates have been unable to submit a fully correct solution, even if we consider very slow programs. Unfortunately, so far none of the programmers tested by Codility have noticed that if the number of segments were big enough (here: 45), the answer to the problem would always be positive.

On the other hand, programmers can be lucky with this problem. They can present an optimal solution without even noticing the aforementioned limit. It is enough to read numbers one by one and register them in a sorted sequence. If, at each number, the program checks whether the number forms a triangle together with its neighbours, it will lead to the optimal solution. We already know that such a procedure cannot last for too long, and either answers that a triangle cannot be formed or finds such a triangle in the 45th step at the latest. Unfortunately, programming contests require only correct and fast programs; no-one cares for proofs of correctness or complexity. It is only effectiveness that counts.

/ Circular *Game*

Contest: Algorithmic Engagements 2009
Task author: Tomasz Idziaszek
Memory: 64 MB
http://main.edu.pl/en/archive/pa/2009/gra

In the *Circular Game*, the board consists of m fields arranged on a circle and numbered from 1 to m. On the board there are b white and c black pieces, at most one on each field. Two players are playing the game: the white player and the black player. Starting with the white player, the players perform their moves on the board alternately. A move consists of moving a piece of the player's color any number of free fields forward or backward. For instance, for the board depicted below, the white player can move the piece from field 3 to field 4 or the piece from field 8 to any of the fields 7, 9 and 1.



If a player can perform no move in his turn, he loses. Knowing that both players play optimally, check who will win the game. It is possible for neither player to win (the game never ends).

Input

The first line of the input contains one integer t representing the number of boards to be considered. The following lines contain descriptions of the respective boards, each of which consists of three lines. In the first line there are three integers, m, b and c ($1 \le m \le 10^9$, $1 \le b, c$), separated by single spaces and specifying the length of the board, the number of white pieces and the number of black pieces. In the second line there is an increasing sequence of b integers (in the range 1 to m) representing the positions of the white pieces. In the third line there is an increasing sequence of c integers (in the range 1 to m) representing the positions of the black pieces. The total number of pieces in all boards does not exceed 10^6.

Output

Exactly t lines, containing answers for the consecutive boards, should be written to the output. The answer is always a single character: B, C, or R, depending on whether the white player wins (B), the black player wins (C) or the game never ends (R).

Example

For the input data:

```
3
9 2 3
3 8
2 5 6
6 2 2
5 6
2 4
7 1 1
3
4
```

the correct result is:

```
C
B
R
```

/ Solution

Friends and relations of Nim

Consider the classic game of *Nim*. Pieces in this game are divided into k heaps of size $n_1, n_2, \ldots, n_k$. Two players take turns, one after the other, and in each turn the player has to remove some positive number of pieces from one of the heaps. The player who collects the last piece wins. If we express the sizes of heaps as binary numbers, and at each position we count ones and all these counts are even, then the game is lost for the player whose turn it is. Namely, if the opponent plays perfectly, our poor player will be doomed. More formally, if the bitwise *xor* of numbers $n_1, n_2, \ldots, n_k$ is zero, our opponent has a winning strategy. Hence, everything depends on the value of the following expression (the symbol $\oplus$ denotes the bitwise xor):

$$n_1 \oplus n_2 \oplus \cdots \oplus n_k.$$

We can also permit the addition of new pieces to heaps, and under a certain condition we will still have a game equivalent to Nim. This condition is a limit for the number of reverse moves (addition of pieces) performed by players. Note that if a player has a losing position, he or she can try adding some pieces somewhere in order to avoid the loss. Unfortunately, the opponent will easily annihilate these efforts simply by removing the added pieces. Since the number of additions is limited, the losing player cannot prolong this procedure forever and will eventually lose. We have just described *Northcott's Game*. It can also easily be explained using a chessboard. In each of its columns we place two checkers: white and black. Checkers can move arbitrarily within their initial column, with a condition that they cannot jump over one another. The distance between two checkers corresponds to the size of a heap. Moving a checker towards the enemy position is like removing pieces from Nim heaps. Moving backward is equivalent to the addition of pieces. However, the board is finite; thus it is impossible to retreat endlessly. Northcott's Game is therefore a variant of Nim.

Analysis of the Circular Game

Wait a moment! Black and white checkers? Moving, not jumping? This is almost our *Circular Game*! Easy! There are some differences; notably, the problem statement allows for the game to be tied. Of course, there are also some contest problems, whereby some answers are just baits for incautious contestants—e.g. one is either to find a sequence or answer NO if no such sequence exists. After the contest, during the discussion of the problem, it emerged that such a sequence always exists—resulting in much wailing and gnashing of teeth! However, that's not a part of this solution. Note that if a player has a row of at least three pieces without any opponent piece between them, and there is at least one free field among them, the player cannot lose as it is always possible to move the pieces inside such a structure. If both players have such a strong formation, the game will be tied.

Partition into blocks. Let us formalize the problem a little. A maximal sequence of a player's pieces, arranged such that there is no opponent piece between any of them, shall be called a *block* and denoted as follows:

$$\langle d, n, l \rangle,$$

where d is the length of the block (the number of fields between its extreme pieces inclusive), n is the number of pieces and l is the length of the free space after the block (see figure 1). The state of a game can be depicted as the sequence of blocks on the board. The rotational symmetry of the board allows any block to be taken as the first one. The similarity to Northcott's Game is now apparent. A player moving their pieces can reduce or enlarge the space around their blocks. As in Northcott's Game, a move shrinking our blocks never leads to victory, since we cannot perform it forever. If all blocks consist of two pieces, they can be used to conquer territory (i.e. to collect pieces from heaps); our game simply reduces to Northcott's Game and eventually Nim. If the board contains k such blocks with gaps $l_1, l_2, \ldots, l_k$, the result of the game will depend on the value of the following expression:

$$l_1 \oplus l_2 \oplus \cdots \oplus l_k. \quad (1)$$

If it is zero and our opponent plays faultlessly, we will lose. Otherwise, we have a winning strategy.

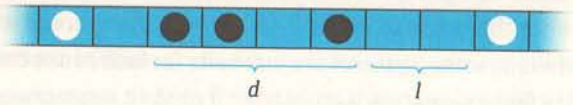


Figure 1: *Sample block for $d = 4$, $n = 3$, $l = 2$.*

Strong blocks. However, there may be blocks with some other number of pieces than two. Let us start with more abundant blocks, and suppose that every block on the board has at least two pieces. The key role is played by blocks with more than two pieces whose length is bigger than their number of pieces. We will call them *strong blocks*. A player who has a strong block cannot lose, since pieces can always be moved inside such a block. Moreover, that player will be the winner if the opponent has no strong block and cannot form one. A strong block is, in fact, like a tank: we can use it to crush any weaker squad of the enemy. In order to foretell the result of the game, we count the number of strong blocks possessed by each player and the number of strong blocks that can be formed in one move. We can form a strong block if a non-strong block with a least three pieces has free space before or after itself.

1. If Black has a strong block, or at least two moves forming such a block, it cannot lose. Thus if White has a strong block, or a move to form one, the game will be tied. Otherwise Black loses.
2. Assume Black has no strong block and no move to form one. If White has a strong block or a move to form one, White wins. Otherwise, the result depends on the Nim-value (1).
3. Now assume that Black has no strong block, but can make exactly one move to form such a block. White moves first, thus has the option to block this move. If White has a strong block, it will prevent Black from forming a strong block and thus win. If White is able to form a strong block, it can form one to assure a tie, or prevent the creation of the Black strong block in order to pursue victory. Therefore, we check the result of the game after this prevention move (technically we reverse the colors of the pieces). Note that each player can have only one such move (other possibilities are considered in the previous two points), so the recursive descent will be performed at most twice.

Singular blocks. Now it remains to discuss blocks containing single pieces. We will call them *singular*. Their fate depends on the surrounding blocks. If the board contains only singular blocks, and each player can move, the game will be tied. Singular blocks are weak, so they are not able to conquer the territory. It can also happen that, in the initial board, White cannot move or can block the only possible move of Black. Such a game of course is not tied.

If singular blocks lie between two non-singular blocks of the opponent's color, they are surrounded in a deadly formation. The owner of these two opposite blocks can easily squeeze what is inside them. We can thus remove such singular blocks. Remember also to remove the space they occupy, because, although deceased, they will always engage the field in which they appear. However, we can achieve the same goal simply by reversing the color of late singular blocks. Consider the following group of blocks:

$$\langle d_i, n_i, l_i \rangle \langle d_{i+1}, n_{i+1}, l_{i+1} \rangle \cdots \langle d_{i+k}, n_{i+k}, l_{i+k} \rangle, \tag{2}$$

where $n_i > 1$, $n_{i+k} > 1$ and $n_{i+j} = 1$ for each $j = 1, 2, \ldots, k-1$.

If k is even, extreme blocks belong to the same player. We replace this group of blocks by the following block:

$$\left\langle \sum_{j=i}^{i+k} d_j,\ n_{i+k} + \sum_{j=i}^{i+k-1} (n_j + l_j),\ l_{i+k} \right\rangle.$$

This operation is associative. Therefore, if we can perform a number of such reductions, their order will not matter.

If k is odd, the singular blocks are inside non-singular blocks of different players. The elimination of these single blocks requires an additional observation. For the player who controls the left end of the group, it only pays to move pieces rightward. For the other player, it pays to move pieces in the opposite direction. The pairs of neighboring pieces will change into fighting pairs of Northcott's checkers. The group of blocks (2) can be thus transformed into the following sequence:

$$\langle d'_i, n'_i, l'_i \rangle \langle d'_{i+1}, n'_{i+1}, l'_{i+1} \rangle \cdots \langle d'_{i+k}, n'_{i+k}, l'_{i+k} \rangle,$$

where for each $j = 1, 2, \ldots, k-1$ it holds that

$$d'_{i+j} = n'_{i+j} = 2$$

and

$$l'_{i+j} = \begin{cases} 0 & \text{if } j \text{ is odd,} \\ l_{i+j} & \text{if } j \text{ is even.} \end{cases}$$

The solution

At last we know how to deal with blocks containing any number of pieces. Therefore we can formulate the solution to the problem. It consists of the following steps:

1. the computation of the initial partition into blocks,
2. the check whether White has at least one move or can block the single possible move of Black,
3. the elimination of singular blocks according to the two last prescriptions,
4. the analysis of existing and potential strong blocks,
5. if strong blocks do not resolve the game, the computation of the value of the expression (1) for remaining gaps between blocks.

All these steps can be performed in linear time and linear space with respect to the number of pieces on the board.

Epilogue

A similar task, but on a smaller (up to 60 fields) and linear (non-circular) board, was used in 1998 at ACM-ICPC South Pacific Regional in New Zealand. I could not find the exact results of that contest on the Internet; however, it appears that no-one solved the problem. It is indeed not easy; nonetheless, it can easily be transformed into the problem discussed above. It is enough to add a field at each end of the board and put there a piece of the same color as the extreme piece from this side. For example, if the leftmost piece is white, we will add a field on the left with a white piece. After this operation we glue together the ends of the board to make it circular. The resulting circular game is equivalent to the original linear game.

It could be interesting to ask for the result of the game on other shapes of board: A planar graph? An arbitrary graph? A tree? A cactus? A Euro 2012 ball? To be continued at the next ACM-ICPC in Poland...

WOJCIECH ŚMIETANKA

Wojciech Śmietanka is a fifth-year student of computer science at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. His scientific interests include algorithmics, number theory and combinatorics. As far as programming competitions are concerned, Wojciech Śmietanka was a finalist of TopCoder Open in 2010 and in the same year qualified for the Google Code Jam finals. However, he did not participate in the latter after all, as he was doing an internship at Google at the time. He has been taking part in team programming competitions since 2007. Together with his team, he has won the Polish Collegiate Programming Contest three times and the ACM-ICPC Central European Regional Contest twice. He also represented the University of Warsaw at the ACM-ICPC World Finals in 2011 and 2012. His non-scientific hobbies include history, economics, dance, volleyball and basketball.

/ Byteland

Contest: 1st Junior Polish Olympiad in Informatics
Task author: Wojciech Śmietanka
Memory: 32 MB
http://main.edu.pl/en/archive/oig/1/baj

King Byteasar the Great is the ruler of the rich and mighty country of Byteland. There are n towns in the country. King Byteasar would like to improve the infrastructure of Byteland, and has ordered the royal architects to prepare plans for building highways across the country. He has received m proposals, each of which is described by three numbers, p, k, w, where p and k are the towns at the endpoints of the highway, and w denotes the cost of building this highway. Each highway is bidirectional and does not visit any town except for its endpoints.

The king would like to choose the highways so as to create at least one way of traveling between any pair of towns, possibly by visiting several towns along the way. Byteasar would also like to build the highway network as cheaply as possible.

Task

Write a program that:

- reads from the input the number of towns, the number of proposed highways and the descriptions of those highways,
- finds out, for each highway, whether a highway network exists that contains this highway and satisfies Byteasar's requirements,
- writes the results to the output.

Input

The first line of the input contains two integers: the number of towns, n, and the number of proposed highways, m, separated by a single space and satisfying the conditions: $2 \le n \le 7000$, $1 \le m \le 300{,}000$. Each of the following m lines contains three space-separated integers, p, k, w, describing the proposed highway. Here p and k denote the numbers of the towns at the endpoints of the highway, and w is the price of the highway ($1 \le p, k \le n$, $1 \le w \le 100{,}000$).

Output

The output should consist of m lines. The i-th of those lines should contain a word, TAK (Polish for *yes*) or NIE (Polish for *no*), depending on whether a highway network exists that contains the i-th highway from the input and satisfies Byteasar's requirements. You can safely assume that at least one highway network satisfying Byteasar's requirements exists.

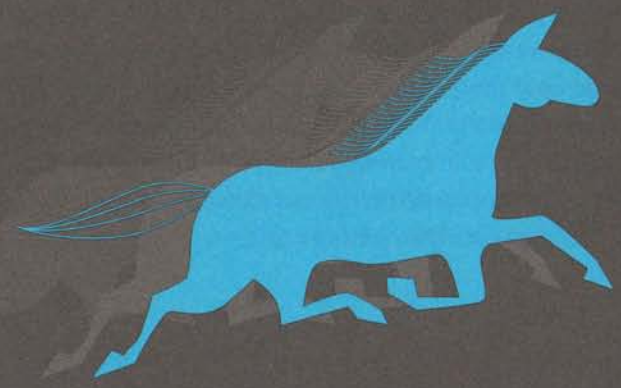
Example

For the input data:

```
6 10
1 2 2
1 6 1
1 5 3
4 1 5
2 6 2
2 3 5
4 3 4
3 5 4
4 5 4
5 6 3
```

the correct result is:

```
TAK
TAK
TAK
NIE
TAK
NIE
TAK
TAK
TAK
TAK
```



Solution

In the description of the solution, it is convenient to use graph theory language. An undirected graph with weighted edges is given, and we would like to tell, for each edge, whether there exists a minimum spanning tree that contains this edge. In the following, we show two different approaches to this task. We use the same symbols as the task statement: n denotes the number of cities and m the number of highways.

Author's solution

At the start we build the minimum spanning tree using Kruskal's algorithm. It follows trivially that, for every edge of this tree, there is a minimum spanning tree that contains this edge. Hence, we get the result for $n-1$ edges. But what about the remaining $m-(n-1)$ edges?

For each of these edges, we would like to know whether it can belong to a minimum spanning tree. We first present the easiest solution. Let us focus on an edge ab. We build the minimum spanning tree by taking this edge and constructing the rest of the tree with Kruskal's algorithm. Then we can check whether the weight of the resulting tree is the same as the weight of the minimum spanning tree obtained previously. Since all edges can be sorted only once at the start, this algorithm can be implemented to run in $O(m^2 \log^* n)$ time. However, this is too slow.

We can do better. Observe that edges that were rejected by Kruskal's algorithm would be also rejected by the modified algorithm. If we compare the run of Kruskal's algorithm—in which we inserted edge ab at the beginning—with the ordinary run, then in corresponding steps, the current spanning tree (or, rather, spanning forest) is "more connected" in the former. What does "more connected" mean? It means that if two vertices are connected in the second of these runs, then they are also connected in the first. Thus we can limit our attention only to edges from the minimum spanning tree.

As a result, when we take an edge ab and construct the rest of the tree, we add only edges that belong to the (optimal) minimum spanning tree. This observation allows us to give an alternate description of our problem. We are given the minimum spanning tree and insert an edge ab, thus creating a cycle. We need to check whether we can remove an edge from this cycle, so that the remaining tree still has minimal cost.

Of course, we can only remove one edge and it has to belong to the cycle; any other deletion would disconnect the tree. Which edge should we delete? Obviously, it is best to remove the most expensive one. Thus, for every edge ab that does not belong to the minimum spanning tree, we check whether its weight w is equal to the weight of the heaviest edge on the path that connects a and b in the tree.

If every edge on the path connecting a and b has weight lower than w, then ab does not belong to any minimum spanning tree. Otherwise, if the heaviest edge on the path had weight w, it could be exchanged with ab without affecting the total cost. The above condition can be checked in $O(n)$ time. Hence, the algorithm runs in $O(mn)$ time, which is still too slow.

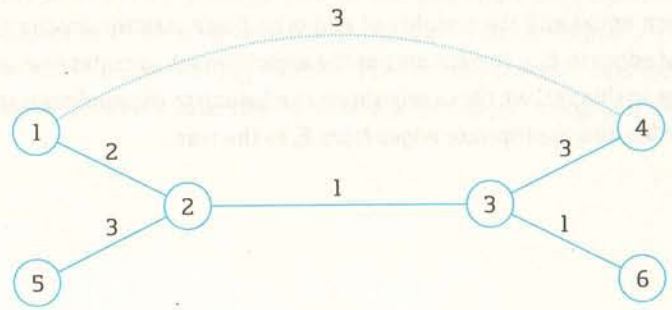


Figure 1: The dotted edge from 1 to 4 has the same weight as the most expensive edge on the tree path from 1 to 4. This means that this edge can replace the edge connecting 3 and 4 in the minimum spanning tree without changing the total cost.

We can improve this further. Observe that the costliest operation, performed $m-n+1$ times, is traversing the tree in order to find the maximum weight of an edge on a particular path. We can do this more efficiently. Start from a vertex v and compute the heaviest edges on paths from v to all other vertices. Having done that, we can easily test whether an edge whose endpoint is v can belong to the minimum spanning tree. This would be the case if the maximum weight of the path connecting its two endpoints were equal to the weight of the edge itself.

We need $O(n^2)$ total time for computing the heaviest edges on all paths in the tree. In addition, we use constant time to perform the check for each edge and $O(m \log m)$ time to run Kruskal's algorithm. The total time is thus $O(n^2 + m \log m)$.

The best solution from the contestants

The contestants were more creative than the author of this task, and came up with an interesting idea: an edge is in some minimum spanning tree if there exists a run of Kruskal's algorithm that includes this edge in a minimum spanning tree.

Kruskal's algorithm may have many different runs. This is caused by the fact that edges of equal weight can be put in any order by the sorting algorithm. It can easily be seen that an edge is in some minimum spanning tree if it is selected by a run of Kruskal's algorithm, in which the edge is in the first position among edges of equal weight.

Let us partition all edges into sets $E_1, E_2, \ldots, E_k$, where inside each E_i the edges' weights are equal and the weights of edges in E_i are strictly smaller than the weights of edges in E_{i+1}. In each step of the algorithm we consider one set E_i. For every edge in this set, we check whether it can belong to the minimum spanning tree. Then we add appropriate edges from E_i to the tree.

```
Algorithm Byteland
  T := ∅
  partition all edges into sets E_i
  for i := 1 to k do
    for ab ∈ E_i do
      if vertices a and b are not connected in T then
        ab belongs to some minimum spanning tree
      else
        ab does not belong to any minimum spanning tree
    for ab ∈ E_i do
      if a and b are not connected in T then
        T := T ∪ {ab}
```

In this way we can get to know, for each edge, whether it belongs to some minimum spanning tree. If we use a union-find algorithm to implement a disjoint-set data structure that checks whether vertices are connected in T, we obtain an algorithm that runs in $O(m \log m)$ time. We can also achieve $O(m \log^* m + W)$ time complexity where W is the maximum of the edges' weights in the graph.

/ Cakes

Contest: Algorithmic Engagements 2009
Task author: Jakub Łącki
Memory: 256 MB
http://main.edu.pl/en/archive/pa/2009/cia

Masters of baking from all over the country have come to this year's Bytean confectioners' convention. During the convention there will be a competition for the best cake baked by a team of three confectioners. Each team of three confectioners may submit at most one cake. However, each contestant can be a member of multiple teams. Unfortunately, some confectioners do not like each other and would not wish to bake cakes together.

It is known how many decagrams of flour each individual contestant requires to bake a cake. A three-person team needs as much flour as is required by the team member with the highest requirements. How much flour will be used during the competition, provided that each team of three confectioners who all like each other will bake exactly one cake?

Input

The first line of the input contains two integers, n and m ($1 \le n \le 100{,}000$, $1 \le m \le 250{,}000$), separated by a single space and specifying the number of confectioners at the convention and the number of pairs of them that like each other. The participants at the convention are numbered from 1 to n. The second line contains n integers p_i ($1 \le p_i \le 1{,}000{,}000$) separated by single spaces and specifying the requirements of the respective confectioners for flour (in decagrams). The following m lines contain data about the pairs of contestants that like each other. Each of these lines contains two integers, a_i and b_i ($1 \le a_i, b_i \le n$, $a_i \ne b_i$), separated by a single space, and represent pairs of confectioners a_i and b_i who like each other. We assume that all pairs of participants of the convention, other than the ones listed in the input, would not wish to bake cakes together. Each pair of confectioners appears at most once in the input.

Output

The first and only line of the output should contain a single integer: the quantity of flour that will be used by all teams in total, in decagrams.

Example

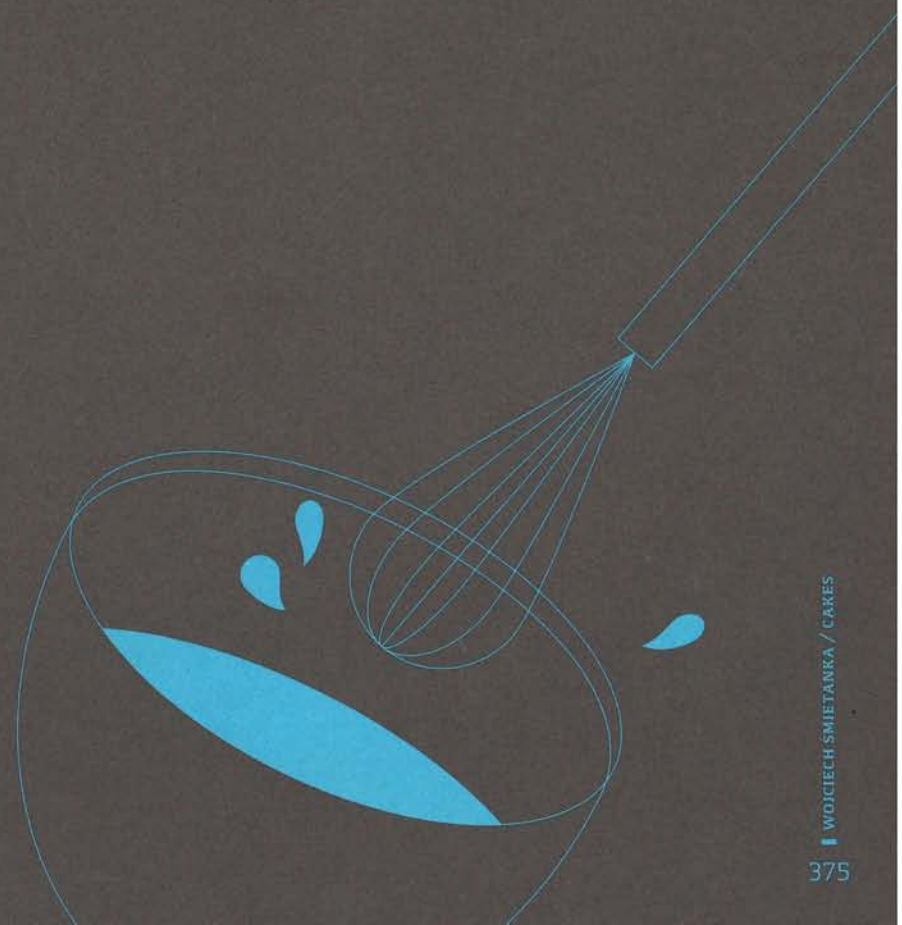
For the input data:

```
5 7
1 5 3 4 2
1 2
2 3
5 2
4 3
3 1
1 4
5 1
```

the correct result is:

```
14
```

Explanation of the example: The following three-person teams: (1,2,3), (1,2,5) and (1,3,4) require 5, 5 and 4 decagrams of flour to bake the cakes. In total, $5 + 5 + 4 = 14$ decagrams of flour are required.



/ Solution

In this task it is convenient to think in terms of graph theory. Each confectioner is a vertex in a graph, and for each pair of confectioners who like each other there is an undirected edge connecting them. We would like to find all trios of vertices in which each pair of vertices is connected with an edge—we will call such trios *triangles*.

Let us think of the simplest solution. For each trio of vertices we can check whether they form a triangle. Such a solution works in $O(n^3)$ time, which is too long for the time limit. However, by making some cunning observations, this can be greatly improved.

Solution

In the simplest solution we perform many wasteful operations, so let us try to prune it. If we choose vertex i as one of the vertices of a triangle, then the remaining vertices j and k have to be neighbours of i. If the degree of a vertex i is equal to d then there are $\frac{d(d-1)}{2}$ pairs of neighbours to consider. We would like d to be as small as possible. Observe that after considering vertex i, we can erase it as well as all its incident edges. How should we choose i? It seems reasonable to consider the vertex with the lowest degree. It turns out that the degree of such a vertex is always quite low.

Lemma. In any simple graph with n vertices and m edges, there exists a vertex of degree lower than or equal to $\sqrt{2m}$.

Proof. Suppose that the degree of every vertex in the graph is greater than $\sqrt{2m}$. Then $n < \sqrt{2m}$, as otherwise the sum of degrees would be greater than $\sqrt{2m} \cdot \sqrt{2m} = 2m$, which is impossible. Since there are fewer than $\sqrt{2m}$ vertices, none of them can have a degree greater than $\sqrt{2m}$: a contradiction. □

Thus, the solution can be summarized as follows. We choose the vertex with the lowest degree, and for each pair of its neighbours we check if they are connected. Then we erase the chosen vertex. We continue this process until we have deleted all vertices.

Observe that we go through all pairs of edges incident to the chosen vertex. From the lemma we infer that each edge would belong to at most $\sqrt{2m} - 1$ such pairs. Hence, we can bound the total number of pairs considered by $m \cdot \sqrt{2m}$. Each of these pairs can be used to construct a triangle, but for this we need the third edge. How can we check whether such an edge exists? At the start, we sort adjacency lists of all vertices and then use a binary search to perform every check in $O(\log n)$ time. This gives a solution that works in $O(m\sqrt{m} \log n)$ time.

A different and faster approach

Let us consider what happens if we direct the edges. If we do this in such a way that there are no directed cycles of length 3, then each triangle has exactly one vertex with two out-edges. In such a case the triangle can be considered from the point of view of this vertex. Moreover, we would like the out-degrees of vertices to be not too big, as checking all pairs of out-edges of a vertex requires time that is quadratic in the number of its out-edges.

Observe that in any undirected graph there are at most $\sqrt{2m}$ vertices of degree greater than $\sqrt{2m}$. Let us order all vertices from left to right in ascending order of their degrees and direct all edges from left to right. Note that the first $n - \sqrt{2m}$ vertices (from the left) have out-degrees lower than $\sqrt{2m}$, as their degrees had been lower than $\sqrt{2m}$ even before the edges were directed. The last $\sqrt{2m}$ vertices also have out-degrees lower than $\sqrt{2m}$, since there are at most $\sqrt{2m} - 1$ vertices to the right of them. In addition, there are no cycles of length 3 in such a graph, as there are no cycles at all.

The out-degree of every vertex of the graph is now at most $\sqrt{2m}$. We iterate through all edges. For an edge $a \to b$ we compute the intersection of adjacency lists of a and b. This intersection is exactly the list of vertices that can complement a triangle with a and b. If we first sort the adjacency lists of all vertices, computing the intersection of two lists takes time that is linear with regard to their total length—in this case it is bounded by $O(\sqrt{m})$. As every edge is processed in $O(\sqrt{m})$ time, the time complexity of the entire algorithm is $O(m\sqrt{m})$.

TOMASZ WALEŃ

Tomasz Waleń works at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, where in 2010 he earned his Ph.D. degree in algorithms in computational biology. His scientific research is centered on various aspects of algorithmics, with special focus on text algorithms.

In 1997 Tomasz Waleń won the 4th Polish Olympiad in Informatics and a bronze medal at the International Olympiad in Informatics. He is the first author of SIO—an electronic web-based system that has been supporting the online facilitation of programming competitions in Poland for the past decade. The system makes it possible for solutions to competition tasks to be evaluated in real time. SIO has facilitated hundreds of competitions, including the 2005 and 2008 International Olympiad in Informatics as well as Central European Regional Contests in Budapest, Warsaw and Wrocław. Tomasz is one of the founders of the software company Codility.com.

In his spare time, Tomasz enjoys mountain trekking and photography.

/ Building *Blocks*

Contest: 15th Polish Olympiad in Informatics
Task author: Tomasz Waleń
Memory: 32 MB
http://main.edu.pl/en/archive/oi/15/klo

Byteasar loved to play with building blocks as a child. He used to arrange the blocks into n columns of random height and then organize them: Byteasar would choose a number k and try to arrange the blocks in such a way that some k consecutive columns would be of equal height. Furthermore, he always tried to achieve this goal in the minimum number of moves possible, where a single move consisted of:

- laying one block on top of any column (Byteasar had a huge box of spare blocks, ensuring that this move could always be performed), or
- removing the uppermost block from the top of any column.

However, Byteasar was never quite sure whether his sequence of moves was indeed optimal; therefore he has asked you to write a program that will help him solve the problem.

Task

Write a program that:

- reads the number k of consecutive columns, along with the initial setup of the blocks, from the input,
- determines the optimal solution (the shortest possible sequence of moves),
- writes out the solution to the output.

Input

In the first line of the input there are two integers, n and k ($1 \le k \le n \le 100{,}000$), separated by a single space. Each of the following n lines contains the height of some column; the $(i+1)$-th line contains the integer $0 \le h_i \le 1{,}000{,}000$—the height of the i-th column, i.e. the number of blocks it consists of.

Output

An optimal solution should be written to the output—i.e. such an arrangement of blocks that:

- contains k consecutive columns of equal height,
- can be obtained from the initial setup in the minimum possible number of moves.

The output should consist of $n + 1$ lines, each one containing a single integer. The number in the first line should be the minimum number of moves needed to get the desired arrangement. The $(i + 1)$-th line (for $1 \le i \le n$) should contain the number h'_i—the final height of the i-th column. If more than one optimal solution exists, output just one (chosen arbitrarily).

Example

For the input data:

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

the correct result is:

```
2
3
9
2
2
2
```


/ Solution

A simplified version of the task

Before we proceed with the solution to the original problem, let us consider its simplified version in which $k = n$. In this version we do not have to worry about choosing the part of the sequence of columns that we have to equalize. Nevertheless, the computation of the minimum cost is still not very obvious.

Of course, if we knew the height to which the columns should be equalized, the computation would be very simple. If this height was equal to h, we would just add blocks to columns for which $h_i < h$ to obtain the height h, and for columns with $h_i > h$ we would just remove the additional blocks.

Checking all possible values of h would, obviously, be too time-consuming, so we have to come up with a better method for finding the optimal height.

Let us assume that we have k columns of blocks of heights $x_1,\ldots,x_k$. Let $c_x(h)$ be the cost of equalizing all columns to obtain the height h. This function is described using the following formula:

$$c_x(h) = \sum_{i=1}^{k} |h - x_i|.$$

Example. Let us consider the configuration of blocks from the example in the task statement; that is, $x = (3,9,2,3,1)$. A plot of the function $c_x(h)$ for this sequence is shown in figure 1.

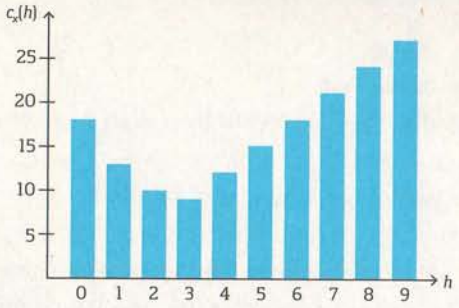


Figure 1: *A plot of the function $c_x(h)$ for the sequence $x = (3,9,2,3,1)$.*

Looking at the plot we can observe that the function $c_x(h)$ decreases, then attains its minimum, and then increases. For us it is important to determine the value of h for which the function $c_x(h)$ attains its minimum.

To find the optimal height we will take a closer look at the function $c_x(h)$. First of all, for $h = 0$ the value of $c_x(0)$ is:

$$c_x(0) = \sum_{i=1}^{k} x_i.$$

Let us now check how the function $c_x(h)$ behaves when its argument increases by 1. To obtain $c_x(h)$ from $c_x(h-1)$, we have to:

- add a single block to each column whose height is less than or equal to $h-1$,
- for each column of height greater than or equal to h, we avoid removing a single block (compared with the solution for $h-1$).

These observations can be transformed into the following formula:

$$\begin{aligned} c_x(h) &= c_x(h-1) + |\{x_i : x_i \le h-1\}| - |\{x_i : x_i \ge h\}| = \\ &= c_x(h-1) + |\{x_i : x_i \le h-1\}| - (k - |\{x_i : x_i \le h-1\}|) = \\ &= c_x(h-1) + 2 \cdot |\{x_i : x_i \le h-1\}| - k. \end{aligned}$$

Let $p_x(h)$ be the number of elements of the sequence x with value less than or equal to x; that is, $p_x(h) = |\{x_i : x_i \le h\}|$. Using this notation we can write $c_x(h)$ as:

$$c_x(h) = c_x(h-1) + 2p_x(h-1) - k = c_x(0) + \sum_{i=0}^{h-1} (2p_x(i) - k).$$

The function $p_x(h)$ is non-decreasing, therefore $c_x(h)$ decreases as long as the elements of the above sum are negative. This means that $c_x(h)$ attains its minimum for the height $h = H$ such that:

$$2p_x(H-1) - k < 0 \quad \text{and} \quad 2p_x(H) - k \ge 0. \tag{1}$$

The value of H that satisfies the inequalities (1) is always well-defined, since the range of the function $p_x(i)$ is $0,\ldots,k$ (for $i = -1,\ldots, \max\{x_j : 1 \le j \le k\}$). Hence, the function $2p_x(i) - k$, for the same domain, may attain values from $-k$ (for $i = -1$) to k (for the maximal value of i).

Once more, let us take a quick look at the inequalities (1). For the requested height H we would like to have in the sequence x:

- fewer than $\lceil k/2 \rceil$ columns of heights less than or equal to $H-1$, and
- at least $\lceil k/2 \rceil$ columns of heights less than or equal to H.

By assigning to H the height of the $\lceil k/2 \rceil$-th (in the order of increasing heights) column of the sequence x, we fulfill both conditions. This element is called the median of the sequence x.

Corollary. The function $c_x(h)$ attains its minimum for h equal to the $\lceil k/2 \rceil$-th (in the order of increasing values) element of the sequence x, where k is the length of the sequence x.

Searching for the optimal segment

From the corollary we can infer a draft algorithm that solves the general version of the task, in which we consider the whole sequence $h_1,\ldots,h_n$:

```
Algorithm ArrangeBlocks((h_1, ..., h_n), k)
    result := +∞
    for i := 1 to n + 1 − k do
        find the median H of the sequence x = (h_i, ..., h_{i+k−1})
        compute the optimal cost for the sequence x, that is, c_x(H)
        result := min(result, c_x(H))
    return result
```

The median of a k-element sequence can be computed in (expected) $O(k)$ time using the Hoare algorithm (unfortunately, the worst-case time complexity of this algorithm is $O(k^2)$). We could also use a more complicated (and not really practical) algorithm with worst-case $O(k)$ time complexity. The value of $c_x(H)$ can easily be computed in $O(k)$ time. This gives us a solution with total time complexity $O(nk)$. If k were a small constant, such a solution would be satisfactory. Unfortunately, in our problem the parameter k can be of magnitude $\Theta(n)$, and in such a case the time complexity of our solution is $O(n^2)$. This is clearly not sufficient, so we need to obtain a more efficient solution.

First, let us observe that the consecutive sequences x, for which we compute the median, are very similar. In the i-th iteration we process the sequence $(h_i, h_{i+1}, \ldots, h_{i+k-1})$, and in the $(i+1)$-th iteration, the sequence $(h_{i+1}, \ldots, h_{i+k-1}, h_{i+k})$. These two sequences differ only by two elements. Using this observation we can obtain a more efficient algorithm, as long as we can implement a data structure S that provides the following methods:

- add(y)—add an element y to the structure,
- remove(y)—remove the element y from the structure,
- median()—return the median of the sequence stored in the structure,
- minCost()—return the value of the function $c_x(median())$ for the sequence stored in the structure.

Using the structure S we can transform the draft algorithm into the following solution:

```
Algorithm ArrangeBlocks2((h_1, ..., h_n), k)
    S := ∅
    for i := 1 to k do
        S.add(h_i)
    result := S.minCost()
    for i := k + 1 to n do
        S.remove(h_{i-k})
        S.add(h_i)
        result := min(result, S.minCost())
    return result
```

Now it all reduces to an efficient implementation of this data structure. This is, however, a more sophisticated problem.

The presence of add and remove operations suggests that we can use some kind of a dictionary data structure; for example, red-black trees. And, in fact, if we add some extra information to the nodes of such trees, each of the requested operations on the structure S can be performed in $O(\log n)$ time. The complexity of the whole algorithm is then $O(n \log n)$. The main drawback of this solution is that it is quite inconvenient to implement. The implementation of the dictionary structures available in the common libraries (like set or multiset in the STL in C++) does not allow for the storing of additional information in the nodes of the tree. So, in order to use this approach, we have to implement the red-black trees by ourselves.

Luckily, we can implement the data structure using less complicated tools. We start the new construction with a more detailed analysis of the information that is necessary to compute the values of the function *minCost*. Assume that all the elements of the multiset S have been divided into: the median (*median*) and two multisets, *le* and *ge*, such that *le* contains elements that do not exceed the median, whereas *ge* contains elements that are at least equal to the median. Denote by *sumLE* and *sumGE* the sum of elements in the multiset *le* and *ge* respectively. Using this additional data, the value $c_x(median)$ can be obtained with the following formula:

$$c_x(median) = (|le| \cdot median - sumLE) + (sumGE - |ge| \cdot median). \qquad (2)$$

We add one more assumption: the multisets *le* and *ge* should be of equal or almost equal cardinality (there can be at most one extra element in the multiset *ge*). This means that the following condition should be an invariant of the algorithm:

$$|le| = \left\lceil \frac{|le| + |ge| + 1}{2} \right\rceil - 1. \qquad (3)$$

Let us notice that the location of elements equal to the median is not strictly determined. Such elements can be stored both in *le* and in *ge*; as long as the invariant is satisfied, we can choose the location arbitrarily. This time we can implement the multisets *le* and *ge* using a standard data structure from the STL; that is, the multiset container.

We have to show how to use the pair of multisets to implement the operations on the data structure S:

- *median*()—returns the stored median;
- *minCost*()—can be computed in constant time using the formula (2);
- *add*(y)—first we add the new element to one of the multisets *le* or *ge*: if $y \le median$, then we add y to *le*; in the opposite case we add y to *ge*.

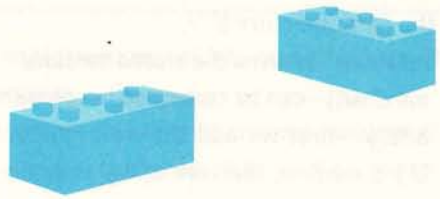
Unfortunately, after this insertion, the invariant (3) might not hold. To restore the invariant, we compare the cardinality of *le* and *ge*, and:

- if one element is missing in *le*, then we add an element equal to *median* to *le*, extract the minimal element from *ge* and save it as the new value of *median*;
- similarly, if one element is missing in *ge*, then we add an element equal to *median* to *ge*, extract the maximal element from *le* and save it as the new *median*.

In both cases we update *sumLE* and *sumGE* accordingly. If we choose the multiset container from STL, then adding, removing and extracting the minimal/maximal elements can be performed in $O(\log k)$ time.

→ *remove*(y)—if $y < median$, we remove the element y from *le*, and if $y > median$, we remove y from *ge*; it is also possible that $y = median$, in which case we remove y from the field *median*, and as the new *median* we set the minimal element from *ge*. In all cases we may have to restore the invariant, but this can be done using the same procedure as in *add*(y).

The whole solution works in $O(n \log k)$ time. From the theoretical point of view, the multisets *le* and *ge* could be implemented with simpler data structures; note that all the necessary operations are provided by the ordinary heap data structure. However, once again we encounter implementation obstacles caused by the limitations of standard programming libraries. Indeed, the algorithm requires the operation of removing an element (function *remove*(y)), and such an operation is not available in the heap implementation of priority_queue in STL.



Matching

Contest: Central European Olympiad in Informatics 2011
Task authors: Tomasz Kulczyński, Tomasz Waleń
Memory: 64 MB
http://main.edu.pl/en/archive/ceoi/2011/mat

As a part of a new advertising campaign, a big company in Gdynia wants to put its logo somewhere in the city. The company is going to spend its whole advertising budget for this year on the logo, so it has to be really huge. One of the managers has decided to use entire buildings as parts of the logo.

The logo consists of n vertical stripes of different heights. The stripes are numbered from 1 to n from left to right. The logo is described by a permutation $(s_1, s_2, \ldots, s_n)$ of numbers $1, 2, \ldots, n$. Stripe number s_1 is the shortest, stripe number s_2 is the second shortest etc., and finally stripe s_n is the tallest. The actual heights of the stripes do not really matter.

There are m buildings along the main street in Gdynia. To your surprise, the heights of the buildings are distinct. The problem is to find all positions where the logo matches the buildings.

Help the company and find all the contiguous parts of the sequence of buildings that match the logo. A contiguous sequence of buildings matches the logo if building number s_1 within this sequence is the shortest, building number s_2 is the second shortest, etc. For example, a sequence of buildings of heights 5, 10, 4 matches a logo described by a permutation (3,1,2), since building number 3 (of height 4) is the shortest, building number 1 is the second shortest and building number 2 is the tallest.

Input

The first line of the input contains two integers, n and m ($2 \le n \le m \le 1{,}000{,}000$). The second line contains n integers s_i, forming a permutation of the numbers $1, 2, \ldots, n$. That is, $1 \le s_i \le n$ and $s_i \ne s_j$ for $i \ne j$. The third line contains m integers h_i—the heights of the buildings ($1 \le h_i \le 10^9$ for $1 \le i \le m$). All the numbers h_i are different. In each line the integers are separated by single spaces.

Output

The first line of the output should contain an integer k: the number of matches. The second line should contain k integers—1-based indices of buildings that

correspond to the stripe number 1 from the logo in a proper match. The numbers should be listed in increasing order and separated by single spaces. If $k = 0$, your program should output an empty second line.

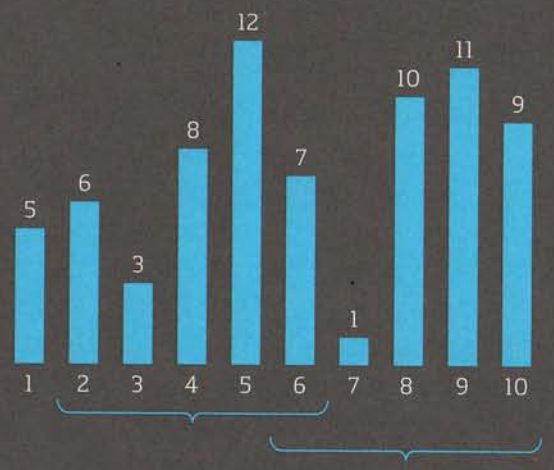
Example

For the input data:

5 10
2 1 5 3 4
5 6 3 8 12 7 1 10 11 9

the correct result is:

2
2 6



Explanation of the example: Both the sequences 6, 3, 8, 12, 7 and 7, 1, 10, 11, 9 match the logo described by the permutation (2,1,5,3,4). In particular, in the first sequence, building number 2 (of height 3) is the shortest, building number 1 (of height 6) is the second shortest, building number 5 (of height 7) is the third shortest, and so on.

Solution

This problem is a variant of the pattern-matching problem, which is a fundamental task in text processing.

In the pattern-matching problem we are given two strings, P and T, and we need to find all occurrences of the pattern P in the text T; i.e. find the positions of all contiguous parts of T that are equal to P. Several efficient algorithms for this problem are known: for example, the Knuth-Morris-Pratt (KMP) algorithm that runs in $O(|P| + |T|)$ time.

In this task we also have to find all occurrences of a pattern in a text; however, the definition of an *occurrence* is not so obvious as in the classical version of the problem.

How to compare sequences

Let us first observe that in this task the pattern is not given in the same format as the text. The input sequence $s_1,\ldots,s_n$ can be used to restore the pattern. This reconstruction is not unique; for one sequence s there could be several possible patterns. To make it unique, we can restrict the elements of the pattern to the range $1,\ldots,n$. For example, for the sequence $s = (2, 1, 5, 3, 4)$ the pattern p has the following form: $(2, 1, 4, 5, 3)$. In general, we have $p[s_i] = i$ for $i = 1,\ldots,n$.

Once we have settled on the canonical form of the pattern, we need to define when a part of the text (denote it by w) matches the pattern (denoted as p). The first and obvious condition is the equal lengths of the two sequences; that is, $|w| = |p|$. The second, more important condition is the identical ordering of elements in w and p, which can be expressed as:

$$w[i] < w[j] \Leftrightarrow p[i] < p[j] \quad \text{for } 1 \le i, j \le |w| = |p|.$$

If p matches w, we will denote it as $p \approx w$. For example, $(2,1,4,5,3) \approx (6,3,8,12,7)$.

Unfortunately this definition has a few disadvantages. Verification of a match is quite time-consuming. In the classical pattern-matching problem it requires linear time, and the verification of the relation $\approx$ directly (from the definition) requires quadratic time.

Luckily, based on a deeper analysis of the relation $\approx$, we can propose a more efficient method of verification.

For the pattern p let us define the following arrays:

$$LPred_p[i] = i - \text{maxarg}_j\{p[j] : j < i \text{ and } p[j] < p[i]\},$$
$$LSucc_p[i] = i - \text{minarg}_j\{p[j] : j < i \text{ and } p[j] > p[i]\}.$$

In the above formulas, maxarg_j and minarg_j denote the value of j for which the maximum and minimum, respectively, of the argument set is attained. For some indexes i the value of $LPred_p$ or $LSucc_p$ may be undefined: this happens when the argument set of the function maxarg/minarg is empty.

i	1	2	3	4	5
$p[i]$	2	1	4	5	3
$LPred_p[i]$	-	-	2	1	4
$LSucc_p[i]$	-	1	-	-	2

Figure 1: *The arrays $LPred_p$ and $LSucc_p$ for the pattern $p = (2, 1, 4, 5, 3)$.*

The arrays *LPred/LSucc* can be used to locate the positions of the predecessor/successor (as to the value) to the left of an element of the pattern.

Using those two arrays, we can significantly reduce the verification time. Now only two comparisons are required for each character of the text. The pattern p matches the text fragment w ($p \approx w$) if:

$$w[i - LPred_p[i]] < w[i] \text{ and } w[i] < w[i - LSucc_p[i]] \text{ for } 1 \le i \le |p| = |w|. \quad (1)$$

Indeed, if $p[j] = p[i] + 1$ (these are two elements of the pattern of adjacent values), then among the above comparisons a comparison between $w[i]$ and $w[j]$ will take place; more precisely, it will take place for the greater of the indices i, j. If for some index i the value $LPred_p[i]$ (or $LSucc_p[i]$) is undefined, the corresponding comparison is not performed. In this case we may assume that $LPred_p[i] = 0$ (or $LSucc_p[i] = 0$) and use a subacute inequality when checking the condition.

The above test can be performed in linear time. Now we have to show that the arrays *LPred/LSucc* can be computed efficiently. Both arrays can be computed in linear time by inspecting the pattern p from the right to the left:

```
Algorithm ComputeLPredLSucc(p)
    let L be a bidirected list of pairs (p[i],i) ordered by the values of p[i]
    for i := |p| downto 1 do
        let x be the element of the list L corresponding to the pair (p[i],i)
        let (a,l) and (b,r) be the predecessor and successor of x in the list L
        LPred_p[i] := i − l (or NULL, if x is the first element of L)
        LSucc_p[i] := i − r (or NULL, if x is the last element of L)
        remove x from the list L
    return (LPred_p, LSucc_p)
```

Now we have all the pieces needed to prepare the first solution. We can adapt the naïve pattern-matching algorithm to obtain a solution with $O(mn)$ running time (where n is the length of the pattern p and m is the length of the text h).

```
Algorithm FindMatchesSimple(h,p)
    (LPred_p, LSucc_p) := ComputeLPredLSucc(p)
    for i := 1 to |h|−|p| + 1 do
        check if p ≈ h[i, ..., (i + |p| − 1)] (using the condition (1))
```

According to the constraints from the problem statement, this solution is not efficient enough. We need to look for a faster algorithm.

How to find all occurrences efficiently

Fortunately, we can modify the Knuth-Morris-Pratt algorithm to fit our definition of a match. The KMP algorithm is a tricky improvement of the naïve pattern-matching algorithm—by precomputing the pattern to find a partial matches function, the algorithm can find all the matches efficiently, spending (on average) constant time on each character of text.

In the original KMP algorithm, the partial matches function is defined as follows:

$$\pi[i] = \max\{j : j < i \text{ and } p[1,\ldots,j] = p[(i-j+1),\ldots,i]\}.$$

A natural application of this definition to our model requires the relation = to be changed to ≈.

$$\pi'[i] = \max\{j : j < i \text{ and } p[1,\ldots,j] \approx p[(i-j+1),\ldots,i]\}.$$

i	1	2	3	4	5
$p[i]$	2	1	4	5	3
$\pi'[i]$	0	1	1	1	2

Figure 2: The values of π' for the pattern $p = (2,1,4,5,3)$.

The modified function π' can be computed using the same algorithm as the original function π. The only change that has to be made is the modification of the test "if a given part of the pattern can be extended by a single character". In the classical KMP algorithm, this test is trivial (we just compare two characters). Here, as in the verification of ≈, we can use information from the arrays *LPred* and *LSucc*. We deal with the undefined elements of *LPred/LSucc* just as before.

```
Algorithm ComputePartialMatchFunction(p, LPred_p, LSucc_p)
  π'[1] := j := 0
  for i := 2 to |p| do
    while j > 0 and not (p[i − LPred_p[j + 1]] < p[i] < p[i − LSucc_p[j + 1]]) do
      j := π'[j]
    if p[i − LPred_p[j + 1]] < p[i] < p[i − LSucc_p[j + 1]] then
      j := j + 1
    π'[i] := j
  return π'
```


Once we have computed the function π', we can find all the occurrences of the pattern efficiently. As in the previous case, we can modify the original KMP algorithm slightly, by changing those parts of the code that include character comparison.

```
Algorithm FindMatches(h, p)
  (LPred_p, LSucc_p) := ComputeLPredLSucc(p)
  π' := ComputePartialMatchFunction(p, LPred_p, LSucc_p)
  j := 0
  for i := 1 to |h| do
    while j > 0 and not (h[i − LPred_p[j + 1]] < h[i] < h[i − LSucc_p[j + 1]]) do
      j := π'[j]
    if h[i − LPred_p[j + 1]] < h[i] < h[i − LSucc_p[j + 1]] then
      j := j + 1
    if j = |p| then
      there is a match ending at the element i
      j := π'[j]
```

This solution runs in linear time. Here, only the two inner while-loops in the functions ComputePartialMatchFunction and FindMatches may look confusing. However, we can observe that each iteration of each of the loops causes a decrease of the counter j, and this variable is increased by $|p| - 1$ (or $|h|$ respectively) during the whole execution of the algorithm.

JAKUB WOJTASZCZYK

Jakub Wojtaszczyk graduated from the University of Warsaw with a Ph.D. degree in mathematics and a M.S. degree in computer science. He is currently a software engineer at Google. He is known for authoring programming tasks, some of which were used at the Polish olympiads both in informatics and mathematics, Algorithmic Engagements, ACM-ICPC World Finals, International Mathematical Competition and Google Code Jam. However, Jakub Wojtaszczyk also has some notable achievements as a contestant: he was a semifinalist of TopCoder Open in 2010, placed second in Algorithmic Engagements in 2011 and came third in the Internet Problem Solving Contest in 2011. Besides his technical interests, he is an active scout leader and an organizer of over twenty summer camps, more than ten of them abroad.

/ Party

Contest: 18th Polish Olympiad in Informatics
Task author: Jakub Wojtaszczyk
Memory: 64 MB
http://main.edu.pl/en/archive/oi/18/imp

Byteasar intends to throw a party. Naturally, he would like it to be a success. Furthermore, Byteasar is quite certain that, for it to be a success, it is sufficient that all the invited guests know each other. He is currently trying to come up with a list of friends he would like to invite.

Byteasar has n friends, where n is divisible by 3. Fortunately, most of Byteasar's friends know one another. Furthermore, Byteasar recalls that he once attended a party where there were $\frac{2}{3}n$ of his friends, and where everyone knew everyone else. Unfortunately, Byteasar does not quite remember anything else from that party... In particular, he has no idea which of his friends attended it.

Byteasar does not feel obliged to throw a huge party, but he would like to invite at least $\frac{n}{3}$ of his friends. He has no idea how to choose them, so he asks you for help.

Input

In the first line of the input, two integers, n and m,

$$(3 \le n \le 3000,\ \frac{\frac{2}{3}n(\frac{2}{3}n-1)}{2} \le m \le \frac{n(n-1)}{2}),$$

are given, separated by a single space. These specify, respectively, the number of Byteasar's friends and the number of pairs of his friends who know each other. Byteasar's friends are numbered from 1 to n. Each of the following m lines holds two integers separated by a single space. The numbers in the $(i+1)$-th line (for $i = 1, 2, \ldots, m$) are a_i and b_i ($1 \le a_i < b_i \le n$), separated by a single space, which specify that persons a_i and b_i know each other. Every pair of numbers appears at most once in the input.

Output

In the first and only line of the output, your program should print $\frac{n}{3}$ numbers, separated by single spaces, in increasing order. These integers should specify

the numbers of Byteasar's friends whom he should invite to the party. As there are multiple solutions, pick one arbitrarily.

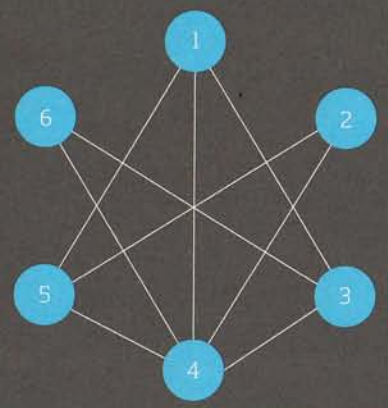
Example

For the input data:

6 10
2 5
1 4
1 5
2 4
1 3
4 5
4 6
3 5
3 4
3 6

the correct result is:

2 4



Explanation of the example: Byteasar's friends numbered 1, 3, 4, 5 know one another. However, any pair of Byteasar's friends who know each other—like 2 and 4, for instance—constitutes a correct solution; i.e. such a pair needs not be part of the aforementioned group of four.

/ Solution

As a problem author I like my problems to be appreciated. The situation that gives me the greatest satisfaction is when someone who does not know that I devised a particular problem starts telling me about it: "Man, what a cool problem I saw recently...!" I would like to present two problems for which exactly this situation occurred.

In both cases, the reason why the problem stuck in my interlocutor's memory was *surprise*—a hard-to-define moment in which some previously unsuspected fact became obvious, and a simple path appeared in what had previously been a tangled combination of questions. To give oneself the opportunity of experiencing this moment—regardless of whether it comes from one's own insight or from reading the solutions I provide—one has to try and tackle the problem oneself: to try to grasp, on one's own, where the real difficulty lies in the problem and spend some time trying to solve it. If you read the solutions without giving the problems an honest try, you will miss out on the opportunity of experiencing this magic moment.

On the other hand, if you have expended the effort on facing the problems yourself, welcome!

About the problem

This problem was created for the finals of the Polish Olympiad in Informatics in 2011. It can be expressed in the language of graph theory. For a given graph $G = (V, E)$ a set K of vertices is called a *clique* if any two vertices of K are connected by an edge. If n denotes the number of vertices of the input graph, we know that there is a clique of cardinality at least $\frac{2}{3}n$ in the graph, while our job is to find a clique of cardinality at least $\frac{1}{3}n$.

It is natural to wonder why the problem statement is so suboptimal: if we know that a clique containing two thirds of the vertices exists, why doesn't the problem ask us to find it? The short answer is that finding a clique of size $\frac{2}{3}n$ is very difficult; I will expand on this later. As a matter of fact, it turned out that even finding the "small" clique proved pretty tricky for quite a few of the participants of the finals of the Olympiad...

One more piece of trivia about the problem statement... When creating a new programming task, it sometimes happens that the problem begins with a story, and an algorithm is discovered later on—this was the case with the second presented task, *Dice*. In this task the algorithmic problem, formulated in the language of graph theory, came first. I spent a bit of time trying to figure

out a story that would justify this atypical situation—we know a clique can exist, but we cannot actually put a finger on it. The moment the idea of a party from which you don't remember very much dawned on me, I knew I had the story I was looking for.

Solution

Consider any two vertices of the graph that are not connected by an edge. Regardless of the clique we are aiming to find, at least one of the two has to be rejected. Which one? It's hard to guess. So, in order not to make a mistake, we will just remove both of them from the graph.

What next? Well, we take some other two vertices not connected by an edge, and throw those two out as well. And so on, as the pseudocode below illustrates:

```
Algorithm Party(V,E)
    for all v ∈ V do
        take[v] := true {Initially all vertices are in the solution}
    for all v, w ∈ V do
        if vw ∉ E and take[v] and take[w] then
            take[v] := false
            take[w] := false
    return take
```

Of course, the real question is why this simple algorithm works (as opposed to a number of other, more or less simple, algorithms that do not). Let K denote the mysterious clique with $\frac{2}{3}n$ elements that we know to be somewhere out there. Each time we remove a pair of vertices from the graph, at most one of them belongs to K, and at least one of them does not. As there are $\frac{1}{3}n$ vertices outside K initially, we can remove a pair of vertices at most $\frac{1}{3}n$ times, so at the end we will have at least $n - 2 \cdot \frac{1}{3}n = \frac{1}{3}n$ vertices left, and each pair will be connected by an edge (otherwise we would have removed it). So the leftover vertices are indeed the clique we were searching for.

Testing

Luckily for me, the Polish Olympiad in Informatics is one of these contests in which the author of the problem is not responsible for creating the test data—"luckily" because this is one of these problems for which a plethora of creative, incorrect solutions exists, and the author of the test data should attempt to eliminate all of them (and, in order to do so, should think quite a few of them

up). Creating tests is actually quite an art, as can be seen in the case of this problem. Consider the following pseudocode:

```
Algorithm IncorrectParty(V,E)
    for all v ∈ V do
        reject[v] := false   {Initially we treat all vertices as candidates.}
    clique := ∅
    while |clique| < n/3 do
        for all v ∈ V do
            if not reject[v] then
                deg[v] := 0
                for all w ∈ V do
                    if (not reject[w]) and vw ∈ E then
                        deg[v] := deg[v] + 1   {We calculate the degree of all
                                                vertices}
        candidate := v0
        for all v ∈ V do
            if ((not reject[v]) and deg[v] > deg[candidate] and v ∉ clique)
                or reject[candidate] or candidate ∈ clique then
                candidate := v   {We search for the candidate with the
                                  highest degree}
        clique := clique ∪ {candidate}
        for all v ∈ V do
            if (v, candidate) ∉ E then
                reject[v] := true
    return clique
```

The idea here is to have a greedy algorithm—one by one we add the vertices with the highest degree to our clique. At first sight this does not look any better or worse than the solution that turned out to be the correct one—so maybe this one is also correct?

To verify that it is *not*, we should find a graph for which this solution will not work. We begin by creating the clique K of size $\frac{2}{3}n$ that our graph has to contain. We add a vertex v_0—this is the vertex we want our algorithm to add to the clique in the first step. As the clique vertices have degree at least $\frac{2}{3}n - 1$, because they are all adjacent, we have to give v_0 a larger degree; say $\frac{3}{4}n - 1$. Suppose it neighbours all the vertices from outside K and as many vertices from K as are needed to sum up to $\frac{3}{4}n - 1$ (it will give $\frac{5}{12}n$ vertices from K).

We have quite a few vertices left, so we will prepare v_1 for our algorithm. At this point we have not rejected around $\frac{1}{3}n$ vertices from outside K from the graph, and around $\frac{5}{12}n$ vertices from K—so let us set the degree of v_1 to be equal to $\frac{1}{2}n$: a bit more than $\frac{5}{12}n$. We achieve this, again, by connecting it to $\frac{1}{6}n+1$ vertices from K. This guarantees all the vertices from outside of K and that the above algorithm chooses v_1 in the second step.

Now, if all the vertices outside K have only the two edges we already created (to v_0 and v_1), the next step of the algorithm will take something from K into its clique, and eventually construct a clique with $\frac{1}{6}n+3$ vertices—too few. So this algorithm is indeed incorrect.

Unfortunately, the job of the test setter is far from done at this point. What if we calculated the degree taking into account all vertices, and not only the ones left in the graph? Or—a nastier case—what if we improved the algorithm by removing all vertices of degree below $\frac{1}{3}n$ (or $\frac{2}{3}n$) from the graph at the beginning (they are of no use to us anyway)? This will pass the test created above. It is, however, still incorrect—although I will leave the creation of a graph to show this as an exercise for the reader.

A bit of background

NP-hardness. The majority of readers will probably have heard about NP-hard problems; this is a class of algorithmic problems for which a polynomial-time algorithm is very unlikely to exist (and also connected to a number of prizes, if somebody does actually devise one). One of the best-known NP-hard problems is to find the largest possible clique in an arbitrary graph. This serves as a partial explanation of why we ask for a "suboptimal" clique in the problem.

Some of you might doubt this; does the fact that we *know* there exists a large (in our case, of size $\frac{2}{3}n$) clique in the graph not help us? After all, we know from experience that searching for something becomes easier if we know that it is actually there to be found... This is not the case here, however, and some easy reasoning proves it. Say we have an algorithm A that allows us to find a clique of cardinality k in a graph, assuming we know such a clique exists. Moreover, assume A is polynomial, i.e. for some polynomial p we have the guarantee that A will find the clique in at most $p(n)$ operations. We will show how to convert this algorithm into one that answers the question whether there is a clique of size k in it. Assume we have a graph G that we know nothing about. Let us run A on G and see what happens. Of course, anything can happen—A can crash, loop, work for a very long time or return some random junk: we have almost no guarantee as to what will occur. Why

"almost"? Well, we do know that if no clique of size k exists, A will not find it. It might *claim* to find it, but we will easily be able to verify that what A found is not, in fact, a clique of size k.

So, if A does end and return something that is not a clique of size k, we know that no such clique was there to be found (that is the assumption about A—if there were anything to find, it would find it; so if it returned junk, there wasn't anything to find). If A works for longer than $p(n)$ steps or crashes, we can also safely claim that there is no clique of size k. So indeed, whatever happens when we run A on an arbitrary graph does tell us whether the graph contains a requested clique. Formally, we turned an algorithm for finding a clique, knowing that it exists, into an algorithm that checks whether a clique exists. Let us add, for clarity, that if A is known to run in polynomial time, but p is unknown, we will not be able to pinpoint the requested polynomial-time algorithm checking for a clique, but we will be able to prove that one exists. Reasonings like the one presented above are called *reductions*, and are standard tools for proving that various problems are NP-hard.

I encourage the reader to find a reduction in the other direction—to show that, given an algorithm that can only answer *whether* there is a clique of size k in a graph, we can also create an algorithm that will *find* it if it exists.

Approximation. The NP-hardness of the clique problem is quite widely known. A somewhat less well known fact is that the clique problem is also *hard to approximate*. For many NP-hard problems, seeing no chance of finding an effective algorithm to give the optimal solution, computer scientists settle on some approximation of the optimum—a solution that we can guarantee is not too far away from the optimal one.

It turns out that the approximation of the clique problem is actually also pretty hard. What does that mean, exactly? For example, the following problem is NP-hard (meaning that it's unlikely to have a polynomial time algorithm):

Problem 1. Given a graph $G=(V,E)$, find a clique in this graph that is at most two times smaller than the largest one.

A wary reader is ready to protest at this point: isn't that exactly the problem we just solved a page or two ago? Well, not exactly. The difference is that in the *Party* problem we know that the optimal size is at least $\frac{2}{3}n$. On the other hand, it turns out that the difficult cases for the general approximation problem begin when the largest clique is rather small, e.g. $\frac{1}{20}n$. This small difference (and my initially futile attempts to grasp the precise meaning of the statement "it is NP-hard to approximate a clique") is the source of the *Party* problem.

An interested reader might like to know that the problem of approximating the clique is far harder than what has been stated above. One can start by replacing the constant 2 with any other constant, and the problem will remain NP-hard. But that's not all! Even the following problem is NP-hard:

Problem 2. Given a graph $G = (V, E)$ with n vertices, find a clique in this graph that is at most $\sqrt{n}$ times smaller than the largest clique in G.

Here, again, the square root can be replaced by any power of n of exponent smaller than 1. The proof of this fact is complex and requires quite a lot of theory (although it is also quite beautiful, and rather well explained in Irit Dinur's paper, "The PCP theorem by gap amplification").

/ Dice

Contest: Algorithmic Engagements 2009
Task author: Jakub Wojtaszczyk
Memory: 32 MB
http://main.edu.pl/en/archive/pa/2009/kos

Jack claims to have telekinetic powers. He has made a bet with Mike that he will throw a die n times and get exactly k pips in total. Jack's die has six faces, numbered from 1 to 6, and is perfectly symmetrical. Mike does not believe in Jack's telekinetic powers (in general, being a tenacious rationalist, he does not believe in telekinesis), but he is concerned that Jack could win the bet simply by luck. Therefore, he would like to know the chance (as a percentage) of such an unlucky event occurring, and has asked you to write a program that will help him solve this problem.

Input

The first line of the input contains a single integer t ($1 \le t \le 20$) specifying the number of test cases. Each of the following t lines contains a description of one test case in the form of two integers, n_i and k_i ($1 \le n_i, k_i \le 10^6$), separated by a single space.

Output

To the output t lines should be written, containing the answers to respective test cases. The answer to each test case is the probability (as a percentage) that Jack wins the bet, rounded *down* to the nearest integer.

Example

For the input data:

```
1
1 6
```

the correct result is:

```
16
```

Solution

This problem appeared in the second round of Algorithmic Engagements and went rather well—not surprisingly, as it is not mindblowingly hard. I came up with it while teaching a *Probability* class about the Central Limit Theorem—which will be described in more detail later on.

Solution

Suppose for a moment that n is bounded from above by 100 or 1000. This would turn out to be a very standard problem in dynamic programming. Let us see—what is the probability of getting exactly k pips on n dice? If $n = 0$, then either $k = 0$ and we have a probability of 100%, or $k \neq 0$ and the probability is zero.

Now we move on to the slightly more interesting case of $n > 0$. Let us throw the first $n - 1$ dice and see what happens. If the total number of pips is smaller than $k - 6$ or larger than $k - 1$, we do not have a chance to get exactly k pips after the last throw. Otherwise, if the number of pips in the first $n - 1$ throws is between $k - 6$ and $k - 1$, we have exactly one chance in six to succeed. Thus if $P(n, k)$ denotes the probability of getting k pips in n throws, we see that

$$P(n,k) = \frac{\sum_{i=1}^{6} P(n-1, k-i)}{6}.$$

Moreover, if $k < n$ or $k > 6n$ then $P(n, k) = 0$.

Thus if we take, say, $1 \leq n \leq 1000$, we can calculate all the 6,000,000 values $P(n, k)$ for $1 \leq n \leq 1000$ and $1 \leq k \leq 6000$ as follows:

```
Algorithm DiceUpTo1000(N,K)
    SetToZeros(P)
    P[0][0] := 100   {The array P holds exact values, not rounded to the
                      nearest percent}
    for n := 1 to 1000 do
        for k := 1 to 6000 do
            for i := 1 to 6 do
                if k − i ≥ n − 1 then
                    P[n][k] := P[n][k] + P[n − 1][k − i] / 6
    return ⌊P[N][K]⌋   {Here we round downwards to the nearest percent}
```

For each feasible pair n, k we add at most six numbers, which gives 36,000,000 additions in total (and a similar number of other operations)—well in time in most contests.

Attempts to use this approach for the full range of n and k cannot succeed. We need to be somewhat smarter.

The key observation is that for large n there are so many different possible results that the probability of hitting a particular one is pretty small—likely to be smaller than 1%. So in this problem we will not need to calculate the exact value; we can just output zero. Having had this idea, we can do one of two things:

- We can take a piece of paper and try to calculate how big n has to be to guarantee that $P[n][k]$ is below 1% for all k; or
- We can simply run the program above and check whether 1000 is enough.

A programmer will naturally opt for the second solution. In just a few minutes we write a program that calculates $P[n][k]$ up to 1000 and learn that the furthest value of $P[n][k]$ that is at least 1% is $P[545][1908] \simeq 1.0002\%$. So we could have written the above program for $n \leq 545$, and if $n > 545$, we can output zero without a single calculation. The complete solution is as follows:

```
Algorithm Dice(N,K)
    if N > 545 then
        return 0
    else
        SetToZeros(P)
        P[0][0] := 100   {Exact values, again}
        for n := 1 to 545 do
            for k := 1 to 545 · 6 do
                for i := 1 to 6 do
                    if k − i ≥ n − 1 then
                        P[n][k] := P[n][k] + P[n − 1][k − i] / 6
    return ⌊P[N][K]⌋   {Rounding to the nearest percent}
```

A bit of background

The background here, as already mentioned, is intuition tied to the Central Limit Theorem (often abbreviated to CLT); it is also a pretty useful tidbit of knowledge. The Theorem itself states that when we add together a large number of independent variables (for instance, dice throws), the sums will approximate a *normal* (or *Gaussian*) *distribution*. Here we do not care exactly what this means (although

it does pay to know sometimes); we care how the CLT translates to knowledge about the highest n for which $P(n,k)$ is not smaller than 1%.

The key fact is that the probability of hitting the most probable outcome which is roughly the average—in this case $3.5n$, after rounding to an integer—is proportional to $1/\sqrt{n}$. The constant of proportionality in this case is roughly $1/\sqrt{6\pi} \approx 0.23$. Thus we would expect that $P(n, 3.5n) \approx 0.23/\sqrt{n}$. If we want this to drop below 1%, we solve $0.01 \approx 0.23/\sqrt{n}$, obtaining $n \approx 529$—a pretty good approximation of the real value of 545.

For applications in other problems, it is worth knowing that, while the exact constant of proportion depends on the type of variables we add (it decreases with the variety and spread of the variable; for instance, if we add pips on twenty-sided dice, it is about 0.07, while for a coin toss, which corresponds to a "two-sided die", it is about 0.56), the probability of hitting the most probable solution is always proportional to $1/\sqrt{n}$. This rough estimate often serves to tell us whether a given approach has any chance of success at all.

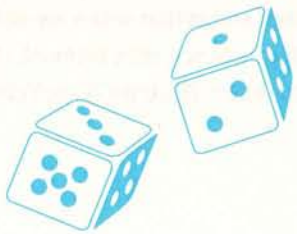
What if we had to count exactly?

In this problem we avoided the issue of calculating $P[n][k]$ for large n thanks to the low precision of the output that was required. It is worth considering whether we could solve this problem if the requested precision were the standard 10^{-6} required in similar problems.

We will not follow the solution to the end, but we will create a rough sketch that can be extended to a full solution (with enough work). We will use two observations to lead us to the solution.

First, notice that the recursive formula we used for $P(n,k)$ was not the only possible one. For example, instead of looking at the last die throw, we could consider the last three throws (assuming $n > 3$), and write

$$P(n,k) = \sum_{i=3}^{18} P(n-3, k-i) \cdot P(3,i).$$



In this formula and all the following formulas we assume that, for negative arguments, P equals 0. We will use yet another formula, obtained by splitting n into two equal halves. If n is even, then

$$P(n,k) = \sum_{i=n/2}^{3n} P(n/2, k-i) \cdot P(n/2, i).$$

Let us assume for a moment that $N = 2^l$ is a power of two. Then, instead of calculating the whole $P[n][k]$ array for $n \le 2^l$, we can just calculate $P[2^m][k]$ for $m \le l$—so instead of $O(N^2)$ values we calculate $O(N \log N)$.

If n is not even, we first calculate $P[n-1][k]$ for all k (using the equality above), and then we go to $P[n][k]$ by considering the last die.

Does this help? We need to calculate $O(N \log N)$ values of P—but unfortunately, calculating one of those values now takes $O(n)$ additions instead of six. So we could say we gained nothing (a more detailed analysis would also show that we did not really lose a log factor, as one might expect, but that is not really satisfactory). Time to pull a second trick out of our sleeves.

With enough algorithmic experience one can notice that the expression

$$P(n,k) = \sum_{i} P(n/2, k-i) \cdot P(n/2, i)$$

looks pretty familiar: it is the so-called *convolution*, and it can also be written as polynomial multiplication. The *Fast Fourier Transform* (FFT) algorithm allows one to calculate all the values of $P(n,k)$ (if the values of $P(n/2,i)$ are given for all i) in $O(n \log n)$ time, as opposed to the $O(n^2)$ one gets by applying the recursive formula directly. Thus, connecting the observation about half-splitting with the FFT, we obtain a solution that works in $O(N \log^2 N)$ time—which will run in time for $N \le 10^5$ (and could even work for $N \le 10^6$ if the time limits were lenient enough).

FILIP WOLSKI

Filip Wolski started participating in programming competitions in Seventh Grade, when Ryszard Szubartowski—a teacher at the Third High School in Gdynia, and now the president of the "Talent" Society—became his mentor. He has won four gold medals at the International Olympiad in Informatics (including first place in 2006) as well as a number of medals and a few victories in regional computer science and mathematics competitions (Baltic Olympiad in Informatics, Central European Olympiad in Informatics, Baltic Way). He has represented the University of Warsaw at the ACM-ICPC World Finals twice, winning the competition with his teammates Marek Cygan and Marcin Pilipczuk in Tokyo in 2007.

Being a water sports enthusiast, and especially passionate about scuba diving, Filip Wolski spent a year traveling and diving around the globe as an OWUSS European Rolex Scholar. He is also a scuba instructor.

His interests include innovative uses of informatics in society and economics. Recently Filip Wolski graduated from his joint program at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw with an M.S. degree in computer science and a B.S. degree in mathematics.

/ Two *Parties*

Contest: 12th Polish Olympiad in Informatics
Task author: Wojciech Guzicki
Memory: 32 MB
http://main.edu.pl/en/archive/oi/12/dwa

King Byteasar has decided to throw two great parties, and wants to invite to them every single inhabitant of Byteland. Of course, he wants to invite each inhabitant to exactly one of the parties. The king knows from his own vast experience that a person has a good time when there is an even number of their own friends at the party. Thus he has asked you to divide the country's inhabitants between the two parties in such a way that as many people as possible have an even number of friends at their party. A trivial division—i.e. one that leaves one party with no guests invited—is allowed. An acquaintanceship is a symmetric relation; i.e. whenever person A knows person B, person B also knows person A.

Task

Write a program that:

- reads from the input the number of Byteland's inhabitants and the descriptions of their acquaintances,
- splits the inhabitants into two parties in such a way that the number of people who have an even number of friends at their party is as large as possible,
- writes to the output a list of people who should be invited to the first party.

Input

In the first line of the input there is one integer N ($1 \le N \le 200$), specifying the number of inhabitants of Byteland. The inhabitants are numbered from 1 to N. In the following N lines there are the descriptions of the inhabitants' acquaintances. At the beginning of the $(i+1)$-th line there is an integer l_i ($0 \le l_i \le N-1$), specifying the number of friends of the i-th inhabitant. It is followed by l_i pairwise distinct numbers of the i-th inhabitant's friends. We assume that no inhabitant is his/her own friend. It follows that each acquaintance is written twice: if A and B know each other, then B appears on A's list of friends and A appears on B's list.

Output

In the first line of the output your program should write one integer M: the number of people who are to come to the first party. In the second line the M numbers of these people should be written. The rest shall come to the second party.

There are many correct results for this task; your program should write any one of them.

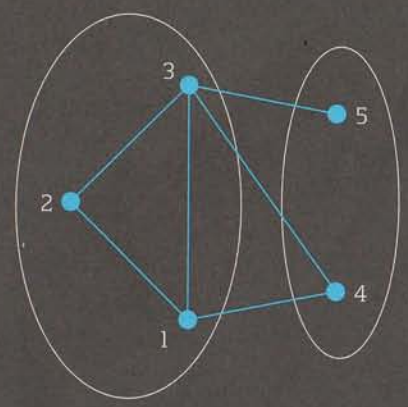
Example

For the input data:

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

the correct result is:

```
3
1 2 3
```



Explanation of the example: In the above example, everyone will have an even number of friends at the party.

/ Solution

Reading the problem statement immediately creates associations. Vertices having even degrees—that brings to mind the problem of determining whether the graph has an Euler cycle. On the other hand, splitting the graph into two parts reminds us of bipartite graphs—and, in fact, if the friendship graph in this task is bipartite, it is easy to split it into two parts so that no vertex has any friend whatsoever in its part, which solves our problem. Going further, determining whether a graph is bipartite involves searching for odd-length cycles in the graph. However, we will actually use a completely different technique to solve this task.

As it turns out, it is always possible to split Bytelanders so that they will all be happy! In other words, the set of vertices of any undirected graph can be split into two subsets, so that each vertex will have an even number of neighbors in its set. We will call such a division *good*. A subgraph in which each vertex has an even number of neighbors will be called a *party*. Given a division of a graph, we will also be talking about the *inner degree* of a vertex—that is, the number of its neighbors that belong to the same subset.

Our proof will use mathematical induction. Additionally, it will be constructive: it will show directly how to build an algorithm that solves this problem. For very small graphs it is easy to see that good division is always possible. It is obvious for a graph with only one or two vertices, and we could also manually check that this is also the case for each graph with three vertices. Let us assume, then, that our hypothesis is true for each graph of N vertices.

Now let us take a graph G with $N+1$ vertices. $G=(V,E)$, where V is the set of its vertices and E is the set of its edges. We will attempt to split the graph into two sets, V_1 and V_2, so that for each $v \in V_i$, set $\{w \in V_i : (v,w) \in E\}$ is of an even size.

If each vertex of G has an even number of neighbors, then it is easy: $V_1 = V$ and $V_2 = \varnothing$. Let us assume, then, that there is at least one vertex v with an odd number of neighbors. Denote by S the set of its neighbors (see figure 1).

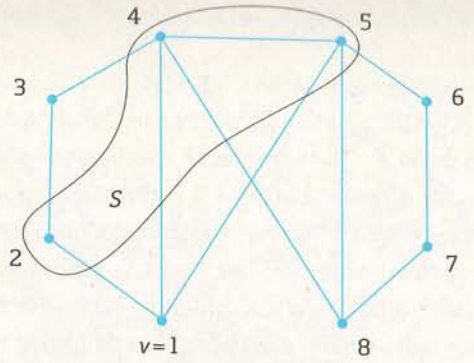


Figure 1: Graph G, in which we have chosen vertex v = 1 of odd degree. S is the set of its neighbors.

We now try to build a good division recursively. First we find a good division of a smaller graph, and on that basis we try to find a good division of the whole graph G. If the smaller graph is obtained by removing vertex v then, after finding a good division of the smaller graph, it would be nice if we could just add v to one of the parties to get a good division of G. If the whole division is to be good, v has to be added to a party containing an even number of elements of S. But if we did that, we would reverse the parity of the inner degrees of elements of S in that subset, and it would no longer be a party (see figure 2)!

Let us try to deal with this problem by correcting G a bit after removing v and before searching for a good division of the smaller graph. Let G' be that corrected graph. After we find a good division of G', in one of its two parties there is an even number of elements of S. We would like these elements to have an odd (instead of even) inner degree. Can we achieve that just by manipulating the edges? Most definitely! What we need to do is to "invert" edges between vertices of S in graph G'. This means that, for each pair of vertices in S, if there is an edge between them in G there is none in G', and if there is no edge in G, there is one in G'. In graph theory language, G' contains an edge supplement of an S-induced subgraph of G.

A vertex with degree s in a graph with n vertices will have degree $n-s-1$ in this graph's supplement, so the difference between these degrees is $|n-2s-1|$. Therefore, if n is even, then the parity of the degree changes, and if it is odd, it stays the same.

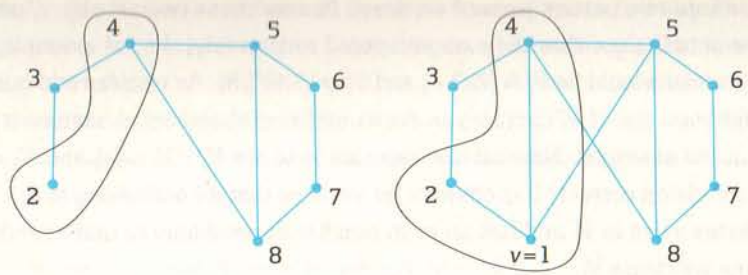


Figure 2: Graph G after removing vertex v and its good division (on the left). If we add v back again, then the marked subgraph is no longer a party.

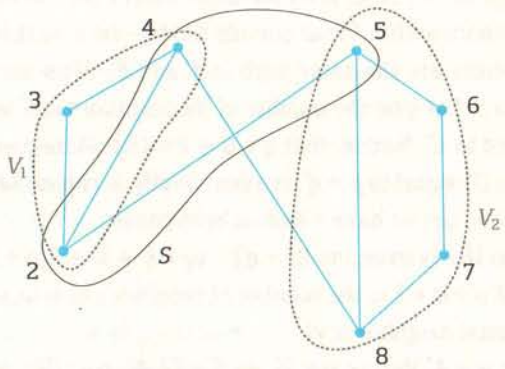


Figure 3: Graph G' created based on v's selection in G.

What happens to the parity of inner degrees of vertices in S? Exactly what we expect. The good division of G' splits S into two subsets S_1 and S_2, such that one of them has even size and the other has odd size. Assume that S_1 is the subset of odd size. Each element of S has an even inner degree in G'. Vertices that belong to S_1 will also have an even inner degree in G. Vertices that belong to S_2 will have an odd inner degree in G—but they become even again after adding v.

Let us make it more formal (see also figure 3). Let $G' = (V', E')$. Vertex set V' is created by removing v from G: $V' = V \setminus \{v\}$. Edge set E' is similar to E, except for edges whose endpoint is v (they are removed) and edges between vertices in S:

- If either u or w is not in S, then $(u, w) \in E' \Leftrightarrow (u, w) \in E$.
- If both u and w are in S, then $(u, w) \in E' \Leftrightarrow (u, w) \notin E$.

From our assumption we know that graph G', which has N vertices, can be split into two parties (sets of vertices). Denote these two sets by V_1' and V_2' (in the actual algorithm they are computed recursively). In our example, one such division would be $V_1' = \{2,3,4\}$ and $V_2' = \{5,6,7,8\}$. As v has an odd number of neighbors, one of V_i' contains an even number of them—let us assume it is V_1' (like in the example). Now all that remains is to set $V_1 = V_1' \cup \{v\}$ and $V_2 = V_2'$. Is this division correct? It is obvious for vertices that do not belong to S:

- Vertex v lies in V_1 and has an even number of neighbors in that set—this is how we chose V_1.
- Vertex $w \neq v$, which does not lie in S, has the same set of neighbors in this division as in the division of G', and that is—as follows from the induction—an even number. In the example, that would be vertices 3 or 6 in figures 1 and 3.

Vertices from S are split into two groups based on the division, $S_1 = S \cap V_1$ and $S_2 = S \cap V_2$. Vertex $w \in S_i$ has an even number of neighbors in V_i', but with regard to the edge set E'. So exactly what neighbors does w have within V_i? It might have neighbors within V_i but outside $S \cup \{v\}$—let p be their number. Note that these neighbors are the same both in E and E'. Now w might also have neighbors within S. Let q be the number of its neighbors in S_i with regard to E, and q' with regard to E'. Notice, that $q + q' + 1 = |S_i|$. Moreover, the number of neighbors of w in G', equal to $p + q'$, is even. Lastly, it is possible that, if $w \in S_1$, v is a neighbor of w. Let us have a look at both cases:

- If $w \in V_1$ then $|S_i|$ is even, so $(p + q') - (q + q' + 1) + 2(q + 1) = p + q + 1$ is also even and $p + q + 1$ is the number of neighbors of w in V_1 with regard to E (one additional neighbor is v).

  Example: Let $w = 4$. Vertex $w \in V_1$, so $S_i = \{2,4\}$, $p = |\{3\}| = 1$, $q = |\varnothing|$ (that is because, in G, vertex 4 is a neighbor of 1, 3, 5 and 8, of which only 5 is from S, but it is not in V_1) and $q' = |\{2\}| = 1$. Then $(p + q') - (q + q' + 1) + 2(q + 1) = 2$, and this is equal to $|\{1,3\}|$—the number of w's neighbors within V_1' with regard to E.
- If $w \in V_2$ then $|S_i|$ is odd, while $q + q'$ is even. Hence, $(p + q') - (q + q') + 2q = p + q$ is even as well, and $p + q$ is number of w's neighbors in V_2 with regard to E.

  Example: Let $w = 5$. Then $p = 2$, $q = q' = 0$ and $(p+q')-(q+q')+2q = 2$, and this is equal to $|\{6,8\}|$—the number of w's neighbors within V_2 with regard to E.

This ends the proof for graphs with $N + 1$ vertices—and, by induction, for any undirected graph.

A sample solution is shown in the pseudocode below. At the beginning, all entries in the *Active* array have to be set to **true**, whereas in array *Even* we assign **true** or **false** to each vertex, depending on whether it has an even number of neighbors. After running the algorithm, the set *Division* contains one of the two subsets of a good division of the graph.

```
Algorithm DivideGraph()
    v := 1
    while v ≤ N and (Even[v] or not Active[v]) do
        v := v + 1
    if v > N then
        Division := {w ∈ V : Active[w]}
        return
    S := ∅
    for w ∈ V ∧ Active[w] ∧ Neighbors[v, w] do
        S := S ∪ {w}
    if |S| mod 2 ≠ 0 then
        for w ∈ S do
            Even[w] := ¬Even[w]
    for (w, u) ∈ S × S ∧ w ≠ u do
        Neighbors[w, u] := ¬Neighbors[w, u]
    Active[v] := false
    DivideGraph()
    if |Division ∩ S| mod 2 = 0 then
        Division := Division ∪ {v}
```

Let N be the number of vertices of graph G. The function *DivideGraph* will be executed no more than N times (each time, one of the vertices gets removed). Its execution time is dictated by the loop that reverses the parities of vertices, which requires $O(N^2)$ steps. Therefore, the running time of the entire algorithm is bounded by $O(N^3)$.

/ Guesswork

Contest: 11th Polish Olympiad in Informatics
Task author: Wojciech Guzicki
Memory: 32 MB
http://main.edu.pl/en/archive/oi/11/zga

Byteasar has become a chronic gambler. He throws away all his money in the casino playing *Guesswork*. He claims that it is possible to devise a game system that allows him to beat the casino.

A single game of *Guesswork* consists of drawing nine consecutive real numbers belonging to (0, 1) using uniform distribution. Upon drawing each number, the player must immediately guess the position of the number (with respect to the standard order of numbers) among the nine to be chosen. Obviously, it is not possible to foresee the position—one has to take a guess. If nine positions are guessed correctly, the player wins; otherwise, the player loses. If, for instance, we guess that the first drawn number is the second biggest, and then draw two numbers that are smaller, we lose.

Task

Help Byteasar! Program the best guessing strategy you can. Write a program that plays multiple games of *Guesswork*, winning as many as possible. The greater the number of games won, the better the evaluation of your program will be. More precisely, for each game won the casino pays the player 423.99 bytean dollars, and for each game lost it collects 13.53 bytean dollars from the player. Your program shall play 10^6 games. The number of points you shall receive will equal the total profit divided by 10^4 and rounded to the nearest integer within [0, 100].

You are to program a module containing the following three functions:

- void inicjalizuj() (Polish for *initialize*)—this procedure shall be called only once, at the outset, before any game begins. You can use it to initialize your data structures.
- void nowa_rozgrywka() (Polish for *new game*)—this procedure will be called at the beginning of each game. You can use it to initialize the variables concerning a single game.
- int kolejna_liczba(double x) (Polish for *next number*)—this function receives a drawn number x as a parameter ($0 < x < 1$), with a precision of 12 decimal places. The function should calculate the position of x among the 9 numbers drawn in this round and return the outcome. If the game is lost, the function may return any integer from 1 to 9.

Example

A hypothetical interaction with the program could be as follows (the numbers returned by the function kolejna_liczba are given in italics; two games have been played.

GAME 1		GAME 2	
0.043319026120		0.164020610610	
1		*2*	
0.933801434041		0.205594263575	
8		*3*	
0.992050359372		0.042637391231	
9		*1*	
0.145189967093		0.147521628974	
4		*1*	Defeat
0.518803250649		0.946549875333	
6		*1*	
0.093583048537		0.772946216573	
2		*1*	
0.764309529654		0.152956276544	
7		*1*	
0.338653748790		0.539653928563	
5		*1*	
0.119437652934		0.552047936535	
3	Victory	*1*	

/ Solution

Guesswork is a very interesting problem—and not only for the gamblers among us. Much like the movie *21*, in which some American students were winning big in blackjack, Byteasar—as we learn from the problem description—intends to outplay casinos in *Guesswork*, using a special strategy. In this game, Byteasar's opponent—the dealer or the casino—does not have any strategy, so Byteasar's strategy is enough to determine the probability of his winning.

If we assume that, before each game, Byteasar picks a permutation of numbers 1 through 9 and uses its consecutive elements during the game, then the probability of his winning a single game would be $\frac{1}{9!} = 0.00000275...$; this means that, with each game, he would be losing 13.52879431... bytean dollars on average.

We can try to determine how good a strategy we need to come up with. If its probability of winning is p, then Byteasar's expected income in each round would be $423.99p - 13.53(1-p)$. Byteasar will be happy if this is positive, because that would make him a millionaire (someday). We, on the other hand, would like Byteasar to earn at least 10^6 bytean dollars after 10^6 games—that is, when we can expect the number of points scored by our solution to be close to 100. We are therefore expecting probability p to be at least

$$423.99p - 13.53(1-p) \geq 1 \Rightarrow p \geq 0.03320991...$$

Brief discussion about the strategy

Let us ponder first how we might represent our strategy. One thing to consider is which events are independent of each other and which are not. For example, in a game where we are to guess the score of the next dice roll, we could incorporate previous results into our strategy; but this would not affect our results. This is because the score of every roll is independent of the previous ones.

In our case, we get a number in the first move. We are to decide how big this number is in comparison with the other eight numbers that we do not know yet. Let us say that we pick an answer and claim that the first number is the i-th smallest.

We can start by computing the probability that we are right. Let $(x_i)_{i=1}^{9}$ be a sequence of numbers given to us by the casino. So far, we know only x_1. We would like to compute the probability that $i-1$ among $x_2,...,x_9$ will be smaller than x_1, and that the other numbers will be greater. First, let us pick those $i-1$ numbers. There are $\binom{8}{i-1}$ ways to pick $i-1$ numbers from a sequence of eight numbers. Once we pick them, for each of the numbers x_i (with $i = 2, 3,..., 9$) we can tell that we would like it to be either smaller or greater than x_1. The probability that a specific number is smaller than x_1 is x_1, whereas the probability of its being greater amounts to $1 - x_1$. Picking each of them is independent, so the probability that all of them will meet our requirements is $x_1^{i-1}(1-x_1)^{9-i}$. Therefore, the cumulative probability that our guess about x_1 is correct is $\binom{8}{i-1}x_1^{i-1}(1-x_1)^{8-i}$, or

$$\binom{n-1}{i-1}x_1^{i-1}(1-x_1)^{n-i} \quad \text{for } n = 9.$$

Let us generalize the *Guesswork* game now: we will say that n-guesswork is a game of *Guesswork*, but played on a sequences of *n elements*, instead of 9. We will also say that it is played on a segment (a, b), if *Guesswork*'s elements are picked with uniform distribution from segment (a, b) (in original *Guesswork*, $a = 0$ and $b = 1$). Note that n-guesswork played on (a, b) is equivalent to n-guesswork played on $(0, 1)$: it just boils down to scaling the elements you receive from the casino (x in the former corresponds to $\frac{x-a}{b-a}$ in the latter).

We now observe that the full description of a strategy is not necessary. All we need is an algorithm for figuring out the first move. Why is that? Assume that in the first move we get 0.4 and respond that it is the fourth element. Therefore we limit the space of games we can still win to those games in which the first element, equal to 0.4, is the fourth smallest one. Hence, all probabilities should now be computed within this game space. If we get 0.3 in the next move, then in general there is a positive probability that it is the fifth element of the whole sequence—but we have already guessed that 0.4 is the fourth one, and the probability of both these events happening together is 0. What we need here is conditional probability: the probability, that 0.3 in the second position is the i-th smallest number *under the condition* that 0.4 in the first position is the fourth smallest number.

Thus, the first move splits the rest of the game into two independent games:

- 3-guesswork played on (0,0.4),
- 5-guesswork played on (0.4,1).

Further on, if our guess was correct, then each of the following moves will take one of the independent games still being played and split it again into two (down to 0-guesswork).

Before we describe it more formally, let us introduce some notation. Recall that $(x_i)_{i=1}^{9}$ denotes the sequence of values given by the casino—at the i-th step we

know only x_j, for all $j \le i$. Let $(t_i)_{i=1}^{9}$ be the sequence of our responses—at the i-th step we are supposed to give element t_i. To make further considerations simpler, let us assume that we also have these sentinels: $x_0 = 0$, $t_0 = 0$, $x_{-1} = 1$, $t_{-1} = 10$.

Each time we receive a new element x_i, we should pick from $x_{-1},\dots,x_{i-1}$ the largest element smaller than x_i (call it x_s) and the smallest element greater than x_i (call it x_g). Thus, $x_s < x_i < x_g$. We have already decided on t_s and t_g values, so we have already assumed that, among elements $x_{\max(s,g)+1},\dots,x_9$, there are exactly $t_g - t_s - 1$ falling into segment (x_s, x_g). The first one of them is x_i. Now we should "ask" our strategy for the best first move in the $(t_g - t_s - 1)$-guesswork played on (x_s, x_g), with the first element equal to x_i; or, equivalently, for the best first move in the $(t_g - t_s - 1)$-guesswork with the first element equal $\frac{x_i - x_s}{x_g - x_s}$.

Simple strategy

We have already figured out that all we need from our strategy is the ability to tell us which number, in sorted order, is the first element x in n-guesswork, for a given $x \in (0,1)$. But we do not yet know how to compute that.

Let us start with the simplest strategy: divide $(0,1)$ into n equal segments. If a given x falls into the i-th segment, then our response is i. If we represent this strategy with S^0, it could be written as:

$$S^0(n,x) = \lceil xn \rceil.$$

This means that if we play n-guesswork and the casino gives us a number x, we say that it is the $\lceil xn \rceil$-th smallest number. We can also represent any strategy in a different way (this will prove useful later on). Let us fix n. Assume that, for $x < y$, we have $S^*(n,x) \le S^*(n,y)$. Strategies that do not fulfill this requirement seem to be less reasonable. Furthermore, the values of $S^*(n,x)$ are $1, 2, \dots, n$. Hence, we can fully specify $S^*(n,x)$ (for this n) by giving values $s^*_{n,0} = 0 \le s^*_{n,1} \le \dots \le s^*_{n,n-1} \le s^*_{n,n} = 1$ and saying that $S^*(n,x) = i \iff s^*_{n,i-1} < x \wedge x < s^*_{n,i}$. In case of S^0 we would have $s^0_{n,i} = \frac{i}{n}$. Observe that this description does not say what to do when the casino gives us one of the numbers $s^*_{n,i}$. This is not a problem—we can give any answer in such a case—as this happens with probability 0.

Before we find out whether this simple strategy is of any use, let us try to describe a slightly more complicated one. If we are to answer the question of which number in sorted order is a given number, why do we not respond with the one with the maximum likelihood? This is analogous to so-called maximum-likelihood estimation.

We already know how to compute the probability that the first element x is the i-th number in sorted order in n-guesswork:

$$p(n,x,i) = \binom{n-1}{i-1} x^{i-1}(1-x)^{n-i}. \tag{1}$$

In the example given before, i.e. $n = 9$ and $x = 0.4$, the right side is maximal for $i = 4$ (for $i = 3$ the right side equals 0.20901888, and for $i = 4$ it is 0.27869184).

Let us call this strategy S^1 and try to compute $s^1_{n,i}$. We already know that $s^1_{n,0} = 0$ and $s^1_{n,n} = 1$ for any n. How do we find the value of $s^1_{n,i}$ for $0 < i < n$? Notice that in $s^1_{n,i}$'s neighborhood strategy, S^1 always responds with either i or $i+1$. More precisely, for $x < s^1_{n,i}$ we have $S^1(n,x) = i$ and for $x > s^1_{n,i}$ we have $S^1(n,x) = i+1$. Additionally, we know that $p(n,x,i)$ is continuous. Therefore, $x = s^1_{n,i}$ must be a value for which $p(n,x,i) = p(n,x,i+1)$. Let us try to find such an x:

$$p(n,x,i) = p(n,x,i+1),$$

$$\binom{n-1}{i-1} x^{i-1}(1-x)^{n-i} = \binom{n-1}{i} x^{i}(1-x)^{n-i-1},$$

$$\frac{1-x}{n-i} = \frac{x}{i},$$

$$x = \frac{i}{n}.$$

Wow! It turns out that this "more complicated" strategy is in fact the same as 0-strategy. Or, in other words, the simple 0-strategy is equivalent to picking the answer with the greatest likelihood. Is this enough to beat the casino?

Strategy analysis

Let us try to compute probability P^*—the probability of winning in a game of *Guesswork*, assuming we are playing with some strategy S^*. Let $P^*(n)$ be the probability of winning n-guesswork in general, while $P^*(n,x)$ is the probability of winning n-guesswork if the first element given by the casino is equal to x. These probabilities depend directly on our strategy:

$$P^*(n,x) = p(n,x,S^*(n,x))P^*(S^*(n,x)-1)P^*(n-S^*(n,x)).$$

Below we will discuss two ways of computing these probabilities. One of them uses the Monte Carlo method to approximate them. The other is more precise, but requires a bit of integral calculus.

Monte Carlo. The Monte Carlo method is often used when we want to compute some values that might be difficult to obtain analytically. Instead of the exact result, we get an approximation by taking enough samples. In our case, we have already decided on a strategy, so we have all we need to write a program to play the game of *Guesswork* with strategy S^1.

What about the casino program? If we were at the competition, we could just use the program provided by the organizers. It would run our solution 10^6 times and tell us how many points we got for this execution. That should be sufficient to decide whether our solution is good enough to score many points.

On the other hand, outside the competition, we could write a simple program to perform as a dealer. This program would also query our solution. We could have it play 10^7-10^8 games and report the fraction of the games won. That fraction would be a good approximation of the value of $P^1(9)$.

Programming all this would be a good exercise for the reader. If you are impatient, though, have a look at the end of the next subsection to find out whether strategy S^1 would score any points...

Precise analysis. Independent of the choice of strategy, the probability of winning the game of guesswork can be described recursively:

$$P^*(n) = \int_0^1 P^*(n,x)dx,$$

$$P^*(n,x) = p(n,x,S^*(n,x))P^*(S^*(n,x)-1)P^*(n-S^*(n,x)).$$

We can make it a bit simpler. For that, we will utilize values $s_{n,i}$ that we have defined before:

$$\begin{aligned} P^*(n) &= \int_0^1 p(n,x,S^*(n,x))P^*(S^*(n,x)-1)P^*(n-S^*(n,x))dx \\ &= \sum_{i=1}^{n} \int_{s^*_{n,i-1}}^{s^*_{n,i}} p(n,x,i)P^*(i-1)P^*(n-i)dx \\ &= \sum_{i=1}^{n} P^*(i-1)P^*(n-i)\int_{s^*_{n,i-1}}^{s^*_{n,i}} p(n,x,i)dx. \end{aligned}$$

Now let us see if we can deal with this kind of integral:

$$\begin{aligned} \int_a^b p(n,x,i)dx &= \int_a^b \binom{n-1}{i-1} x^{i-1}(1-x)^{n-i}dx \\ &= \binom{n-1}{i-1}\int_a^b x^{i-1}\sum_{j=0}^{n-1}\binom{n-i}{j}(-x)^j dx \\ &= \binom{n-1}{i-1}\int_a^b \sum_{j=0}^{n-1}\binom{n-i}{j}(-1)^j x^{i+j-1}dx \\ &= \binom{n-1}{i-1}\sum_{j=0}^{n-1}\binom{n-i}{j}(-1)^j\int_a^b x^{i+j-1}dx \\ &= \binom{n-1}{i-1}\sum_{j=0}^{n-1}\binom{n-i}{j}(-1)^j\frac{b^{i+j}-a^{i+j}}{i+j}. \end{aligned}$$

We can compute that numerically with any precision we need. It is easy to see that $P^1(0) = P^1(1) = 1$. Now for further values we can use the following formula:

$$P^1(n) = \sum_{i=1}^{n} P^1(i-1)P^1(n-i)\int_{\frac{i-1}{n}}^{\frac{i}{n}} p(n,x,i)dx.$$

After we implement this, we find out that $P^1(9) = 0.03075546...$; so strategy S^1 is not sufficient in this problem. But what can we improve?

Better strategy

So far, we have had a very simple strategy: each time we pick the response with the greatest likelihood. This seems reasonable—the very same value we are maximizing is the one we use later when computing the probability of winning. Recall that it can be calculated from the formula:

$$P^*(n,x) = p(n,x,S^*(n,x))P^*(S^*(n,x)-1)P^*(n-S^*(n,x)).$$

We choose $S^1(n,x)$ so that $p(n,x,\ S^1(n,x))$ is maximal. But, as we look at it now... why not pick $S^*(n,x)$ to maximize the entire right-hand side?

Our mistake led us to pick the response with the greatest likelihood in the first step. Instead, we should be picking the response that will maximize our chance to win the entire game—and that includes further steps as well.

Here is an example: 3-guesswork with first element $x = 0.3$. Should we say it is the first or the second number in sorted order? It is more likely that it is the first one—the probability that the next two elements are larger is 0.49, while the probability that exactly one is larger is 0.42. In first case, though, we have one more decision to make later on—whether the second element is the second or the third smallest one. We can win this 2-guesswork with probability 0.75. So if we respond that 0.3 is smallest element of the sequence, then we win the whole game with probability $0.49 \cdot 0.75 = 0.3675$. On the other hand, if we respond that it is the second smallest number, then although that is true with a probability of only 0.42, there are no more choices to take later.

Let us call this strategy S^2. What are the values s^2? We compute them in a similar way to s^1. In strategy S^2 we are maximizing the value of $q(n,x,i) = p(n,x,i)P^2(i-1)P^2(n-i)$ by picking response i. Therefore we find the border value of $s^2_{n,i}$ by equating:

$$q(n,x,i) = q(n,x,i+1),$$

$$p(n,x,i)P^2(i-1)P^2(n-i) = p(n,x,i+1)P^2(i)P^2(n-i-1),$$

$$\frac{1-x}{n-i}P^2(i-1)P^2(n-i) = \frac{x}{i}P^2(i)P^2(n-i-1),$$

This eventually gives us

$$s^2_{n,i} = x = \frac{iP^2(i-1)P^2(n-i)}{iP^2(i-1)P^2(n-i) + (n-i)P^2(i)P^2(n-i-1)}.$$

In the S^1 strategy we computed P^1 values to discover whether it pays off to use it. Here, in turn, we need to compute the values in order to define the strategy. As with S^1, we will have $P^2(0) = P^2(1) = 1$, and for $n \geq 2$:

$$P^2(n) = \sum_{i=1}^{n} P^2(i-1)P^2(n-i)\int_{s^2_{n,i-1}}^{s^2_{n,i}} p(n,x,i)\,dx.$$

As in the previous analysis, we do not need integral formulas here. If, for a fixed n, we know $P^2(n')$ for each $n' < n$, then based on that we can compute $s^2_{n,i}$ (for all $0 \leq i \leq n$). This is enough to play n-guesswork with strategy S^2.

However, here we do not have to use integrals either. Instead, we can use the Monte Carlo method to approximate $P^2(n)$. This way we can precompute border values $s^2_{n,i}$ in advance, using the Monte Carlo method for consecutive values of n. If we incorporate them directly into our solution's code, we will not need to recompute them when our solution is playing the game.

Now we have everything we need to compute not only the strategy but also the probability of winning *Guesswork* when playing with strategy S^2: $P^2(9) = 0.03337533...$; this should already be enough to get 100 points.

How do we figure out the probability of getting the maximum score using strategy S^2? For that we need a bit of knowledge from probability theory. Say that after 10^6 games we win X of them (X is a random variable). We would like $423.99X - 13.53(10^6 - X) \geq 10^6$, which is equivalent to $X \geq 33210$. The probability of winning in a single game of guesswork is $p = 0.03337533...$. From the Central Limit Theorem we know that X has (approximately) normal distribution, with expected value $\mathbb{E}X = 10^6 p > 33375$ and standard deviation $\sigma_X = \sqrt{10^6 p(1-p)} < 180$. Therefore the probability of $\{X < 33210\}$ is less than:

$$\mathbb{P}(X < 33210) < \mathbb{P}(X - \mathbb{E}X < -165) < \mathbb{P}\left(\frac{X - \mathbb{E}X}{\sigma_X} < -\frac{165}{180}\right) =$$

$$= \mathbb{P}\left(\mathcal{N}(0,1) < -\frac{165}{180}\right) < \Phi\left(-\frac{165}{180}\right) < 0.18.$$

So by playing strategy S^2 we will get 100 points with a probability of at least 0.82.

Closing remarks

The problem *Guesswork* is pretty original. It first appeared on a trial day of the final stage of the Polish Olympiad in Informatics, so it was probably treated as an experimental problem by the Olympiad's Committee. Some of the competitors solved it correctly. Others finished without a full solution—e.g. they played *Guesswork* using the simple strategy described above. One of the competitors managed to totally beat the casino by figuring out future game progressions based on pseudo-random number generator variables.

I remembered this problem, years after the competition, while on a lengthy trip. After a long SMS exchange with an unknown number, the time came for me finally to guess which of several girls I had recently met was on the other end. Well trained by the Olympiad, I guessed it was the one I liked most.

Task index

Afternoon Tea / 216
Altars / 138
Ants / 18
Army Training / 307
Axes of Symmetry / 151
Barricades / 63
Building Blocks / 379
Byteland / 368
Cakes / 374
Canoes / 338
Cave / 264
Chocolate / 186
Circular Game / 360
Coding of Permutations / 91
Dice / 403
Fibonacci Sums / 193
Fishes / 31
Fuel / 323
Game of Tokens / 277
Guesswork / 416
Guilds / 293
Hashing / 209
Hyperclock / 25
Ice Skates / 251
Kangaroos / 220
Leonardo Numbers / 157
Matching / 387
Monkeys / 108
Mushrooms / 77
Painter's Studio / 342
Party / 395
Pilots / 41
Plotter / 116
Ploughing / 330
Questions / 176
Reconstruction of Byteland / 302
Riddle / 315
Ritual / 170
Rooks / 102
Shuffle / 272
Skiers / 53
Straight Lines / 281
Sums / 235
Superknight / 240
Sweets / 83
Termites / 123
Termites 2 / 256
The Search / 202
Travel Agency / 68
Triangles / 352
Triangles 2 / 353
Two Parties / 409
Window / 132

OFEK
OGÓLNOPOLSKA FUNDACJA
EDUKACJI KOMPUTEROWEJ

Ogólnopolska Fundacja Edukacji Komputerowej (OFEK) was established in Wrocław by Founding Members included in the notary deed of February 24, 1986. The Foundation was brought to life at the initiative of directors of Wrocław-based enterprises and institutions, especially Mr. Bogusław Lisowski, the first President of OFEK, and Mr. Andrzej Myszkier, the current President of OFEK. During the 25 years of its existence, the Foundation has been cooperating closely first with the Ministry of Education and Upbringing and its later equivalent—Ministry of National Education, thanks to which it was able to supply more than 8700 computer labs in schools across the country with over 91,000 computers and 15,000 printers.

OFEK has been fulfilling its statutory goal by coordinating and participating in training events for over 80,000 teachers. These activities have not only brought financial benefits but, as Mr. Lisowski predicted, they enabled Polish schools to catch up with the most modern schools in the world—turn-of-the-century schools that emerged in the midst of the information revolution.

Since the very beginning the main goal of the Foundation has been to provide computer education to people of all ages, especially the youth, by popularizing the use of computers for teaching and learning purposes.

The Foundation's activities largely contribute to:

- wider application of computers in didactic and development processes, thanks to which former students are able to make use of computers in everyday life once they have already graduated,
- society's engagement in activities supporting computer education,
- popularization of achievements in the field of science and technology that facilitate practical use of computers,
- continuous development of computer culture within the society.

The Foundation fulfills its goals by:

- raising financial means, computer equipment and other means as well as increasing their volume in the course of its own business activities,
- creating favorable conditions supporting the learning process, creative and productive efforts aiming at popularization of computer education in the society, especially among pupils and students,
- granting scholarships, organizing contests, exhibitions and olympiads in informatics,
- cooperating with local and national institutions and other parties interested in activities supported by the Foundation.

Contest tasks say a lot about the quality of a programming competition. They should be original, engaging and of different levels of difficulty. Finding a solution should cause the contestant to feel great satisfaction, whereas being unable to solve a given task should encourage an individual to broaden their knowledge and develop new skills. This book contains the best tasks from algorithmic and programming competitions organized or co-organized by the University of Warsaw, together with their exemplary solutions.

The selection of the tasks was undertaken by people who have played significant roles in the history of Polish algorithmic and programming competitions as their participants or organizers. All of the authors of texts presented in this book are closely affiliated with the Faculty of Mathematics, Informatics and Mechanics at the University of Warsaw, whether as former or current students or as academic staff.

Each of the tasks presented and discussed in this book was used during one of the following events: Polish Olympiad in Informatics, Junior Polish Olympiad in Informatics, Polish Olympiad in Informatics Training Camp, Central European Olympiad in Informatics, Polish Collegiate Programming Contest and Algorithmic Engagements.

ISBN 978-83-920897-2-8

Looking for a Challenge?

The Ultimate Problem Set from the University of Warsaw Programming Competitions

LOOKING FOR A CHALLENGE?